HILBERT (RELEASE 3): A MATLAB IMPLEMENTATION OF ADAPTIVE BEM

MARKUS AURADA, MICHAEL EBNER, MICHAEL FEISCHL, SAMUEL FERRAZ-LEITE, THOMAS FÜHRER, PETRA GOLDENITS, MICHAEL KARKULIK, MARKUS MAYR, AND DIRK PRAETORIUS

Abstract. The Matlab BEM library `HILBERT` allows the numerical solution of the 2D Laplace equation on some bounded Lipschitz domain with mixed boundary conditions by use of an adaptive Galerkin boundary element method (BEM). This paper provides a documentation of `HILBERT`. The reader will be introduced to the data structures of `HILBERT` and mesh-refinement strategies. We discuss our approach of solving the Dirichlet problem (Section 5), the Neumann problem (Section 6), the mixed boundary value problem with Dirichlet and Neumann boundary conditions (Section 7), and the extension to problems with non-homogeneous volume forces (Section 8). Besides a brief introduction to these problems, their equivalent integral formulations, and the corresponding BEM discretizations, we put an emphasis on possible strategies to steer an adaptive mesh-refining algorithm. In particular, various error estimators are discussed. Another notable feature is a complete and detailed description of our Matlab implementation which enhances the reader's understanding of how to use the `HILBERT` program package.

Table of Contents

Date: **29.05.2013**, 08.06.2012, 12.07.2010, 29.01.2010, 20.11.2009.
`HILBERT` — `http://www.asc.tuwien.ac.at/abem/hilbert/`.

1. Introduction

The boundary element method is a discretization scheme for the numerical solution of elliptic differential equations. On an analytical level, the differential equation, stated in the domain, is reformulated in terms of a certain integral equation called *representation theorem* or *third Green's formula*. For the Laplace equation on some bounded Lipschitz domain $\Omega \subset \mathbb{R}^2$, each solution of

$$-\Delta u = f \quad \text{in } \Omega \tag{1.1}$$

can explicitly be written in the form

$$u(x) = \widetilde{N} f(x) + \widetilde{V}\phi(x) - \widetilde{K} g(x) \quad \text{for all } x \in \Omega, \tag{1.2}$$

where $\phi := \partial_n u$ is the normal derivative and $g := u|_\Gamma$ is the trace of u on $\Gamma := \partial\Omega$. The involved linear integral operators read

$$\begin{aligned}
\widetilde{N} f(x) &:= -\frac{1}{2\pi} \int_\Omega \log|x-y|\, f(y)\, dy, \\
\widetilde{V}\phi(x) &:= -\frac{1}{2\pi} \int_\Gamma \log|x-y|\, \phi(y)\, d\Gamma(y), \\
\widetilde{K} g(x) &:= -\frac{1}{2\pi} \int_\Gamma \frac{(y-x)\cdot n_y}{|y-x|^2}\, g(y)\, d\Gamma(y),
\end{aligned} \tag{1.3}$$

where n_y denotes the outer unit vector of Ω at some point $y \in \Gamma$. Put differently, the solution u of (1.1) is known as soon as the Cauchy data $(\partial_n u, u|_\Gamma)$ are known on the entire boundary Γ.

If one considers the trace of u, the representation formula (1.2) becomes

$$g = u|_\Gamma = Nf + V\phi - (K - 1/2)g. \tag{1.4}$$

If one considers the normal derivative of u, the representation formula (1.2) becomes

$$\phi = \partial_n u = N_1 f + (K' + 1/2)\phi + Wg. \tag{1.5}$$

The two linear equations (1.4)–(1.5) are known as *Calderón system*. It involves six linear integral operators acting only on Γ: the simple-layer potential V, the double-layer potential K with adjoint operator K', the hypersingular integral operator W, and the trace N and the normal derivative N_1 of the Newton potential $\widetilde{N}$.

For the boundary element method, the Laplace equation with given boundary data is equivalently stated in terms of the Calderón system (1.4)–(1.5). This leads to a boundary integral equation formulated on Γ. This integral equation is solved numerically to obtain (approximations of) the missing Cauchy data. In a postprocessing step, the computed Cauchy data are then plugged into the representation formula (1.2) to obtain an approximation of the solution u of the differential equation.

Examples for this approach are given in the subsequent sections: In Section 5, we consider the Dirichlet problem, where $g = u|_\Gamma$ is known and where the unknown normal derivative $\phi = \partial_n u$ has to be computed. More precisely, we consider the weakly-singular integral formulation of

$$\begin{aligned}
-\Delta u &= 0 \quad \text{in } \Omega, \\
u &= g \quad \text{on } \Gamma.
\end{aligned} \tag{1.6}$$

In Section 6, we consider the Neumann problem, where the normal derivative ϕ is known and where the unknown trace g has to be computed. More precisely, we consider the hypersingular integral formulation of

$$\begin{aligned} -\Delta u &= 0 \quad \text{in } \Omega, \\ \partial_n u &= \phi \quad \text{on } \Gamma. \end{aligned} \tag{1.7}$$

Finally, in Section 7, we consider a mixed boundary value problem, where Γ is split into two disjoint parts Γ_D and Γ_N and where g is known only on $\Gamma_D \subset \Gamma$, whereas ϕ is known only on $\Gamma_N \subset \Gamma$. More precisely, we consider the so-called symmetric integral formulation of

$$\begin{aligned} -\Delta u &= 0 \quad \text{in } \Omega, \\ u &= g \quad \text{on } \Gamma_D, \\ \partial_n u &= \phi \quad \text{on } \Gamma_N. \end{aligned} \tag{1.8}$$

All of these integral formulations lead to first-kind integral equations with elliptic integral operators so that the Lax-Milgram lemma applies and provides existence and uniqueness of discrete solutions.

Whereas this documentation focusses on the implementation of the adaptive lowest-order BEM only, we refer to the literature for details on the analysis and the numerics of BEM: For instance, the analysis of boundary integral equations is completely presented in the monograph [25]. For the state of the art in numerical analysis, we refer to [30]. Fast BEM is discussed in [29]. These algorithms are, however, beyond the scope of MATLAB and consequently beyond the scope of `HILBERT`. Finally, an introductory overview on the analysis of elliptic boundary integral equations and the boundary element method is best found in [31].

1.1. What is `HILBERT`. Throughout, $\Gamma = \partial\Omega$ is the piecewise affine boundary of a polygonal Lipschitz domain $\Omega \subset \mathbb{R}^2$. Sometimes, Γ is partitioned into finitely many relatively open and disjoint boundary pieces, e.g. in a Dirichlet boundary Γ_D and a Neumann boundary Γ_N, i.e., $\Gamma = \overline{\Gamma}_D \cup \overline{\Gamma}_N$ and $\Gamma_D \cap \Gamma_N = \emptyset$.

Let $\mathcal{E}_\ell = \{E_1, \dots, E_N\}$ be a finite set of affine line segments $E_j \in \mathcal{E}_\ell$, i.e., there holds

$$E_j = [a_j, b_j] := \operatorname{conv}\{a_j, b_j\} \tag{1.9}$$

with certain $a_j, b_j \in \mathbb{R}^2$ with $a_j \neq b_j$. We say that $\mathcal{E}_\ell$ is a mesh (or triangulation) of Γ provided that $\Gamma = \bigcup_{j=1}^N E_j$ and $|E_j \cap E_k| = 0$ for all $E_j, E_k \in \mathcal{E}_\ell$ with $E_j \neq E_k$. If Γ is partitioned into Γ_D and Γ_N, we assume that this partition is resolved by $\mathcal{E}_\ell$, i.e., $E_j \in \mathcal{E}_\ell$ satisfies either $E_j \subseteq \overline{\Gamma}_D$ or $E_j \subseteq \overline{\Gamma}_N$. Finally, $\mathcal{K}_\ell = \{z_1, \dots, z_N\}$ denotes the set of all nodes of the mesh $\mathcal{E}_\ell$, and we note that there holds $\#\mathcal{E}_\ell = \#\mathcal{K}_\ell$ for the closed boundary Γ.

`HILBERT` [1] is a MATLAB library for the solution of (1.6)–(1.8) by use of h-adaptive lowest-order Galerkin BEM. In particular, missing Neumann data are approximated by an $\mathcal{E}_\ell$-piecewise constant function $\Phi_\ell \approx \phi$, and missing Dirichlet data are approximated by an $\mathcal{E}_\ell$-piecewise affine and continuous function $G_\ell \approx g$. Given an initial coarse mesh $\mathcal{E}_0$ of Γ, the adaptive loop generates a sequence of improved meshes $\mathcal{E}_\ell$ by iterative local mesh-refinement. Throughout, `HILBERT` uses the canonical bases, i.e., characteristic functions χ_j associated with elements $E_j \in \mathcal{E}_\ell$ to represent discrete fluxes Φ_ℓ and nodal hat functions ζ_k associated with nodes $z_k \in \mathcal{K}_\ell$ to represent discrete traces (of concentrations) G_ℓ.

1.2. Outline of Documentation. Section 2 recalls some analytical preliminaries like the definition of boundary integrals and the arclength derivative. Moreover, some notation is introduced which is used throughout the entire document. In Section 3, we give a concise overview on all functions and functionality provided by `HILBERT`. Section 4 discusses our implementation of the local mesh-refinement and the marking strategy used in the adaptive mesh-refining algorithms. The Dirichlet problem (1.6) and its numerical solution by use of `HILBERT` is discussed in Section 5, whereas Section 6 is concerned with the Neumann problem (1.7). Section 7 treats the use of `HILBERT` for the numerical solution of the mixed boundary value problem (1.8). Finally,

Section 8 is devoted to the implementation of integral equations with non-homegeneous volume forces in HILBERT and Section 9 gives a detailed overview of new error estimators and functions for evaluating the integral operators.

2. Preliminaries

2.1. Functions on the Boundary. With each element $E_j = [a_j, b_j] \in \mathcal{E}_\ell$, we associate the affine mapping

$$\gamma_j : [-1,1] \to E_j, \quad \gamma_j(s) = \frac{1}{2}\big(a_j + b_j + s(b_j - a_j)\big) \tag{2.1}$$

which maps the reference element $[-1,1] \subset \mathbb{R}$ bijectively onto E_j.

Let $\mathcal{P}^p(\mathcal{E}_\ell)$ be the space of all $\mathcal{E}_\ell$-piecewise polynomials of degree $p \in \mathbb{N}_0$ with respect to the arclength. By definition, this means that for all $f_\ell \in \mathcal{P}^p(\mathcal{E}_\ell)$ and all elements $E_j \in \mathcal{E}_\ell$, the function $f_\ell \circ \gamma_j : [-1,1] \to \mathbb{R}$ satisfies

$$f_\ell \circ \gamma_j \in \mathcal{P}^p[-1,1], \tag{2.2}$$

i.e., $f_\ell \circ \gamma_j$ is a usual polynomial of degree (at most) p. Note that functions $f_\ell \in \mathcal{P}^p(\mathcal{E}_\ell)$ are, in general, not continuous, but have jumps at the nodes of $\mathcal{E}_\ell$.

In particular, $\mathcal{P}^0(\mathcal{E}_\ell)$ is the space of all $\mathcal{E}_\ell$-piecewise constant functions. If $\chi_j \in \mathcal{P}^0(\mathcal{E}_\ell)$ denotes the characteristic function of $E_j \in \mathcal{E}_\ell$, the set $\{\chi_1, \dots, \chi_N\}$ is a basis of $\mathcal{P}^0(\mathcal{E}_\ell)$.

One particular example for a function in $\mathcal{P}^0(\mathcal{E}_\ell)$ is the local mesh-width $h_\ell \in \mathcal{P}^0(\mathcal{E}_\ell)$ which is defined $\mathcal{E}_\ell$-elementwise by

$$h_\ell|_E := \operatorname{length}(E) = |b - a| \quad \text{for all } E = [a,b] \in \mathcal{E}_\ell. \tag{2.3}$$

Let $\mathcal{S}^1(\mathcal{E}_\ell) := \mathcal{P}^1(\mathcal{E}_\ell) \cap C(\Gamma)$ denote the set of all continuous and (with respect to the arclength) $\mathcal{E}_\ell$-piecewise affine functions. For each node $z_j \in \mathcal{K}_\ell$ of $\mathcal{E}_\ell$, let $\zeta_j \in \mathcal{S}^1(\mathcal{E}_\ell)$ be the associated hat function, i.e., $\zeta_j(z_k) = \delta_{jk}$. Then, the set $\{\zeta_1, \dots, \zeta_N\}$ is a basis of $\mathcal{S}^1(\mathcal{E}_\ell)$.

In HILBERT, we only consider the lowest-order Galerkin BEM, and the spaces $\mathcal{P}^0(\mathcal{E}_\ell)$ and $\mathcal{S}^1(\mathcal{E}_\ell)$ will be of major interest.

2.2. Boundary Integrals. Let $I \subset \mathbb{R}$ be a compact interval in $\mathbb{R}$. For $E_j \in \mathcal{E}_\ell$, let $\pi_j : I \to E_j$ be a continuously differentiable and bijective mapping. For any function $f : E_j \to \mathbb{R}$, the boundary integral is then defined via

$$\int_{E_j} f \, d\Gamma = \int_{E_j} f(x) \, d\Gamma(x) := \int_I (f \circ \pi_j)(t) \, |\pi_j'(t)| \, dt. \tag{2.4}$$

One can prove that this definition is independent of the parametrization π_j. For the reference parametrization γ_j, there holds

$$\int_{E_j} f \, d\Gamma = \frac{\operatorname{length}(E_j)}{2} \int_{-1}^1 f \circ \gamma_j \, dt,$$

where $\operatorname{length}(E_j) := |b_j - a_j|$ denotes the Euclidean length of $E_j = [a_j, b_j] \subset \mathbb{R}^2$. For the arclength parametrization

$$\beta_j : [0, \operatorname{length}(E_j)] \to E_j, \quad \beta_j(t) := a_j + \frac{t}{\operatorname{length}(E_j)} \, (b_j - a_j)$$

holds

$$\int_{E_j} f \, d\Gamma = \int_0^{\operatorname{length}(E_j)} f \circ \beta_j \, dt.$$

2.3. Arclength Derivative. Let $I \subset \mathbb{R}$ be a compact interval in $\mathbb{R}$. For $E_j \in \mathcal{E}_\ell$, let $\pi_j : I \to E_j$ be a continuously differentiable and bijective mapping with $|\pi_j'(t)| > 0$ for all $t \in I$. For any function $f : E_j \to \mathbb{R}$, the arclength derivative f' is then defined by

$$(f' \circ \pi_j)(t) = \frac{1}{|\pi_j'(t)|} (f \circ \pi_j)'(t) \quad \text{for all } x = \pi_j(t) \in E_j. \tag{2.5}$$

Again, one can show that this definition is independent of the chosen parametrization. For the reference parametrization γ_j, we obtain

$$(f' \circ \gamma_j)(t) = \frac{2}{\operatorname{length}(E_j)} (f \circ \gamma_j)'(t) \quad \text{for all } x = \gamma_j(t) \in E_j,$$

whereas the arclength parametrization β_j leads to

$$f' \circ \beta_j = (f \circ \beta_j)'.$$

3. Overview on Hilbert

3.1. Tree Structure. The BEM library `HILBERT` is contained in a zip-file `hilberttools.zip`. Unzipping this zip-archive, you obtain the following tree structure

```
hilberttools/
   demos/
      laplace/
      poisson/
   paper/
   source/
      make/
```

The root directory `hilberttools/` contains this documentation `documentation.pdf` as well as the m-files. Besides this, it contains the installation file and `make.m` to build the integral operators from the Matlab command line. The C-source codes of the integral operators are contained in `source/`, and `source/make/` contains `HILBERT`'s make system. The folders `demos/laplace/` and `demos/poisson/` contain examples and demo files which demonstrate the use of `HILBERT`.

3.2. Installation. To install `HILBERT`, unpack the zip-archive, change to the root folder, start Matlab, and type `make` at the Matlab command line. If you have started Matlab in graphical mode, some dialogs pop up and ask certain questions depending on your operating system. If you are running Windows, you may choose to install optimized pre-compiled binaries of all Mex functions. You may select between different versions which are optimized for different numbers of CPU cores. Please note that the precompiled binaries depend on the versions 2008 and 2010 of Microsoft's *Visual C++ Redistributable* package. These packages might be installed on your system already, but in case one of them is missing Matlab will fail when calling any of the Mex functions that ship with `HILBERT`. The Visual C++ Redistributable packages are freely available at `http://www.microsoft.com`.

If you decide to compile `HILBERT` yourself, you may customize its configuration. If you select "No", a default configuration file will be created in `source/make/Configure.m`. Otherwise, a configuration dialog pops up, where you may choose to enable multi-threading or change some of `HILBERT`'s internal options. If you enable multi-threading, you have to make sure that you compile `HILBERT` using either `GCC` or Microsoft's Visual C Compiler and that a POSIX threads library is available on your system. In case that you use UNIX or Linux, this is most certainly the case. For Windows, `HILBERT` ships with the required libraries. If multi-threading is enabled, `HILBERT` will use as many CPU cores as you specify to build the integral operator matrices. This may lead to a huge speed-up on multi-core or multi-processor systems.

As soon as the configuration file has been created, all Mex functions will be compiled. The progress is indicated on Matlab's command line.

If you want to remove the installed binaries, you may type `make clean` on MATLAB's command line from within `HILBERT`'s root directory. If you also want to remove `HILBERT`'s configuration, you may type `make realclean`.

If you start MATLAB in non-graphical mode, text-based dialogs will appear on MATLAB's command line, which provide the same features and options as the graphical dialogs.

3.3. Data Structure. The set of nodes $\mathcal{K}_\ell = \{z_1, \dots, z_N\}$ of the mesh $\mathcal{E}_\ell = \{E_1, \dots, E_N\}$ is represented by the $(N \times 2)$-array `coordinates`. The j-th row of `coordinates` stores the coordinates of the j-th node $z_j = (x_j, y_j) \in \mathbb{R}^2$ as

`coordinates(`j`,:)` $= [\ x_j \ \ y_j\ ]$.

If Γ is not split into several parts, the mesh $\mathcal{E}_\ell$ is represented by the $(N \times 2)$-(formally integer) array `elements`. The i-th boundary element $E_i = [z_j, z_k]$ with nodes $z_j, z_k \in \mathcal{K}_\ell$ is stored as

`elements(`i`,:)` $= [\ j \ \ k\ ]$,

where the nodes are given in counterclockwise order, i.e., the parametrization of the boundary element $E_i \subset \Gamma$ is mathematically positive. Put differently, the outer normal vector $\mathbf{n}_i \in \mathbb{R}^2$ of Γ on a boundary element $E_i = [z_j, z_k]$ reads

$$\mathbf{n}_i = \frac{1}{|z_k - z_j|} \begin{pmatrix} y_k - y_j \\ x_j - x_k \end{pmatrix} \quad \text{with} \quad z_j = (x_j, y_j), z_k = (x_k, y_k).$$

If Γ is split into Dirichlet boundary Γ_D and Neumann boundary Γ_N, the mesh $\mathcal{E}_\ell$ is represented by the $(N_D \times 2)$-array `dirichlet` and the $(N_N \times 2)$-array `neumann` which describe the elements $E_j \subseteq \overline{\Gamma}_D$ and $E_k \subseteq \overline{\Gamma}_N$ as before. Then, there formally holds

```
elements = [dirichlet;neumann]
```

with $N = N_D + N_N$. The array `elements`, however, is not explicitly built or stored in this case.

For problems with non-homogeneous volume force, an additional triangulation $\mathcal{T}_\ell = \{T_1, \dots, T_n\}$ of Ω is necessary. The set of vertices $v_j \in \mathbb{R}^2$ of $\mathcal{T}_\ell$ is denoted by $\mathcal{V}_\ell = \{v_1, \dots, v_m\}$ and represented by the $(m \times 2)$-array `vertices`. As for `coordinates`, the j-th row of `vertices` stores the j-th vertex $v_j = (x_j, y_j) \in \mathbb{R}^2$ as

`vertices(`j`,:)` $= [\ x_j \ \ y_j\ ]$.

The triangulation $\mathcal{T}_\ell$ is represented by the $(n \times 3)$- (formally integer) array `triangles`. The p-th triangle $T_p = \mathrm{conv}\{v_i, v_j, v_k\}$ with $v_i, v_j, v_k \in \mathcal{T}_\ell$ is stored as

`triangles(`p`,:)` $= [\ i \ j \ k\ ]$.

The order is chosen in a mathematically positive way, i.e. z_i, z_j, z_k are given in counter-clockwise order. Moreover, the edge $\mathrm{conv}\{v_i, v_j\}$ is assumed to be the reference edge of $T_p \in \mathcal{T}_\ell$ for later mesh-refinement, see 4.4 below.

3.4. Overview on Functions and Functionality. In this section, we list all functions provided by `HILBERT`, describe their input and output parameters, provide a short overview on their functionality, and give links to a detailed description within this documentation.

Throughout, let $\mathcal{E}_\ell = \{E_1, \dots, E_N\}$ be a given mesh of Γ with nodes $\mathcal{K}_\ell = \{z_1, \dots, z_N\}$ described in terms of `coordinates` and `elements`. Recall that χ_j denotes the characteristic function associated with $E_j \in \mathcal{E}_\ell$ and that ζ_k denotes the hat function associated with $z_k \in \mathcal{K}_\ell$. Moreover, as far as non-homogeneous volume forces are involved, $\mathcal{T}_\ell = \{T_1, \dots, T_n\}$ denotes a regular triangulation of Ω into non-degenerate triangles $T_i \in \mathcal{T}_\ell$, and $\mathcal{V}_\ell = \{v_1, \dots, v_m\}$ denotes the corresponding set of vertices.

3.4.1. General Functions. For marking elements in an adaptive mesh-refining strategy, we use the Dörfler marking introduced in [13]. This is realized in a generalized way by the function

```
[marked [,marked2,..]] = markElements(theta [,rho], indicator1 [,indicator2,..]);
```

see Section 4.1 for details.

For the local mesh-refinement of the boundary mesh $\mathcal{E}_\ell$, we realize an Algorithm analyzed in [3] which is proven to be optimal with respect to the number of generated elements. For a mesh $\mathcal{E}_\ell$ described in terms of `coordinates` and `elements` and a vector `marked` containing the indices of elements $E_j \in \mathcal{E}_\ell$ to be refined, the function call reads

```
[coord,elem,father2son] = refineBoundaryMesh(coordinates,elements,marked);
```

where the generated mesh $\mathcal{E}_{\ell+1}$ is described by `coord` and `elem`. Moreover, `father2son` returns a link between the meshes $\mathcal{E}_{\ell+1}$ and $\mathcal{E}_\ell$. Further optional arguments of the function are discussed in Section 4.3. In particular, this includes the mesh-refinement if Γ is split, e.g., into Dirichlet and Neumann boundary parts.

Next, we consider a function for local mesh-refinement of a coupled mesh. Let $\mathcal{T}_\ell$ be a triangulation of the domain Ω and $\mathcal{E}_\ell = \mathcal{T}_\ell|_\Gamma$ be the boundary mesh for $\Gamma := \partial\Omega$. The triangulation is described in terms of `vertices` and `triangles` and the boundary mesh is described in terms of `vertices` and `boundary`. A vector `marked` contains the indices of the elements $T_j \in \mathcal{T}_\ell$ of the elements of the triangulation that are refined and a vector `marked_boundary` contains the indices of the boundary elements $E_j \in \mathcal{E}_\ell$ that are refined. Then the function call reads

```
[vert, tri, bdry, father2volumes, father2boundary] ...
  = refineMesh(vertices, triangles, boundary, marked, marked_boundary)
```

For the local mesh-refinement of the volume triangulation $\mathcal{T}_\ell$, we use newest vertex bisection, where marked elements $T_i \in \mathcal{T}_\ell$ are refined by one bisection, see Section 4.4. For a triangulation $\mathcal{T}_\ell$ described in terms of `vertices` and `triangles` and a vector `marked` containing the indices of elements $T_j \in \mathcal{T}_\ell$ to be refined, the function call reads

```
[vert,tri] = refineMesh(vertices,triangles,marked);
```

where the generated mesh $\mathcal{T}_{\ell+1}$ is described by `vert` and `tri`. Further optional arguments of the function allow to treat meshes, where the boundary mesh $\mathcal{E}_\ell = \mathcal{T}_\ell|_\Gamma$ is the partition of Γ induced by $\mathcal{T}_\ell$. We refer to Section 4.4 for a detailed discussion of the optional parameters.

When dealing with mixed boundary partitions and/or non-homogeneous volume forces, it is sometimes necessary to change the numbering of the boundary nodes $\mathcal{K}_\ell = \{z_1, \dots, z_N\}$ (or vertices $\mathcal{V}_\ell$) such that, e.g., the Neumann nodes are first, i.e. $\mathcal{K}_\ell \cap \overline{\Gamma}_N = \{z_1, \dots, z_{N_N}\}$. This reordering is done by use of

```
[coordinates,neumann,dirichlet] = buildSortedMesh(coordinates,neumann,dirichlet);
```

Moreover, this function can be used to generate the boundary partition $\mathcal{E}_\ell := \mathcal{T}_\ell|_\Gamma$ induced by a volume mesh $\mathcal{T}_\ell$

```
[vertices,triangles,coordinates,elements] = buildSortedMesh(vertices,triangles);
```

For further optional parameters, we refer to Section 4.2.

3.4.2. Discrete Integral Operators. The discrete simple-layer potential matrix

$$\mathbf{V} \in \mathbb{R}^{N\times N}_{sym}, \quad \mathbf{V}_{jk} = \langle V\chi_k \, , \, \chi_j \rangle_{L^2(\Gamma)} \tag{3.1}$$

is returned by call of

```
V = buildV(coordinates,elements [,eta]);
```

Note that $\mathbf{V}$ is a dense matrix. Since MATLAB does not easily allow matrix compression techniques like hierarchical matrices [20] or the fast multipole method, the assembly of $\mathbf{V}$ (and the other discrete integral operators below) as well as the storage is of quadratic complexity. To lower the computational time in MATLAB, the computation is done in C via the MATLAB-MEX interface. We stress that all matrix entries can be computed analytically by use of analytic anti-derivatives derived, e.g., in [22]. If numerical stability is concerned, it is, however, better to compute certain entries by use of numerical quadrature instead, see [24]. The optional

admissibility parameter `eta` determines whether certain entries $\mathbf{V}_{jk}$ are computed by numerical quadrature instead of analytic integration. Details are found in Section 5.2.

The discrete double-layer potential matrix

$$\mathbf{K} \in \mathbb{R}^{N\times N}, \quad \mathbf{K}_{jk} = \langle K\zeta_k \, , \, \chi_j \rangle_{L^2(\Gamma)} \tag{3.2}$$

is returned by call of

```
K = buildK(coordinates,elements [,eta]);
```

see Section 5.2. The discrete hypersingular integral operator matrix

$$\mathbf{W} \in \mathbb{R}^{N\times N}_{sym}, \quad \mathbf{W}_{jk} = \langle W\zeta_k \, , \, \zeta_j \rangle_{L^2(\Gamma)} \tag{3.3}$$

is returned by call of

```
W = buildW(coordinates,elements [,eta]);
```

see Section 6.2. As above, the optional parameter `eta` in both functions determines whether all matrix entries are computed analytically via anti-derivatives from [22] or whether certain entries are computed by numerical quadrature [24].

Examples with non-homogeneous volume force $f \neq 0$ additionally involve the discrete Newton potential matrix

$$\mathbf{N} \in \mathbb{R}^{N\times n}, \quad \mathbf{N}_{jk} = \langle N\chi_{T_k} \, , \, \chi_j \rangle_{L^2(\Gamma)}, \tag{3.4}$$

where χ_j is the characteristic function of $E_j \in \mathcal{E}_\ell = \{E_1, \ldots, E_N\}$, whereas χ_{T_k} is the characteristic function of the triangle $T_k \in \mathcal{T}_\ell = \{T_1, \ldots, T_n\}$. This matrix is returned by call of

```
N = buildN(coordinates,elements,vertices,triangles [,eta]);
```

see Section 8.2.

Finally, the jump conditions of the double-layer potential K and its adjoint K' give rise to the mass-type matrix

$$\mathbf{M} \in \mathbb{R}^{N\times N}, \quad \mathbf{M}_{jk} = \langle \zeta_k \, , \, \chi_j \rangle_{L^2(\Gamma)} \tag{3.5}$$

This sparse matrix is provided by

```
M = buildM(coordinates,elements);
```

see Section 5.3.

3.4.3. Numerical Solution of Dirichlet Problem. The Laplace problem with Dirichlet boundary condition (1.6) is equivalently recast in Symm's integral equation

$$V\phi = (K + 1/2)g$$

with g being the known Dirichlet data and ϕ being the unknown Neumann data. We refer to Section 5 for details. In the Galerkin formulation, we replace the Dirichlet data g by its nodal interpolant or its L^2-projection G_ℓ in $\mathcal{S}^1(\mathcal{E}_\ell)$

By approximation of g, we introduce an additional error which is controlled by the so-called Dirichlet data oscillations

$$\mathrm{osc}^2_{D,\ell} = \sum_{E\in\mathcal{E}_\ell} \mathrm{osc}_{D,\ell}(E)^2 \quad \text{with} \quad \mathrm{osc}_\ell(E)^2 = \mathrm{length}(E)\|(g - G_\ell)'\|^2_{L^2(E)}, \tag{3.6}$$

cf. [6] and [21]. The function call

```
[oscD,gh] = computeOscDirichlet(coordinates,elements,g);
```

returns a column vector with `oscD(j)` $= \mathrm{osc}_{D,\ell}(E_j)^2$ as well as the nodal vector `gh` of g, see Section 5.1.1. To discretize g by the L^2-projection, one uses the function call

```
[oscD,gh] = computeOscDirichletL2(coordinates,elements,g);
```

where `gh` contains the nodal values of G_ℓ, see Section 5.1.2.

The right-hand side vector of the Galerkin scheme then takes the form

$$\mathbf{b} \in \mathbb{R}^N \quad \text{with} \quad \mathbf{b}_j = \langle (K+1/2)G_\ell \,,\, \chi_j \rangle_{L^2(\Gamma)} \tag{3.7}$$

and is computed by

```
b = buildSymmRHS(coordinates,elements,gh);
```

where `gh` is the nodal vector of g (or an arbitrary nodal vector of another approximation G_ℓ of g), see Section 5.4.

In academic experiments, the exact solution $\phi \in L^2(\Gamma)$ is known, and

$$\mathrm{err}_{N,\ell}^2 + \mathrm{osc}_{D,\ell}^2 = \sum_{E \in \mathcal{E}_\ell} \big(\mathrm{err}_{N,\ell}(E)^2 + \mathrm{osc}_{D,\ell}(E)^2 \big) \tag{3.8}$$

with $\mathrm{err}_{N,\ell}(E)^2 = \mathrm{length}(E) \|\phi - \Phi_\ell\|_{L^2(E)}^2$ is an upper bound for the Galerkin error $|||\phi - \Phi_\ell|||_V^2$ with respect to the energy norm $|||\cdot|||_V$. The function call

```
err = computeErrNeumann(coordinates,elements,x,phi);
```

returns a column vector with `err(j)` $= \mathrm{err}_\ell(E_j)^2$. Here, `phi` is a function handle for the known exact solution and `x` is the coefficient vector of the Galerkin solution $\Phi_\ell = \sum_{j=1}^N x_j \chi_j$. We refer to Section 5.5 for details.

For a posteriori error estimation and to steer an adaptive mesh-refinement, `HILBERT` includes four $(h - h/2)$-based error estimators from [18, 15], discussed in Section 5.6 in detail: With $\widehat{\Phi}_\ell$ a more accurate Galerkin solution with respect to a uniformly refined mesh $\widehat{\mathcal{E}}_\ell$, the a posteriori error estimators

$$\eta_\ell = |||\widehat{\Phi}_\ell - \Phi_\ell|||_V \;\text{ and }\; \mu_\ell^2 = \sum_{E \in \mathcal{E}_\ell} \mu_\ell(E)^2 \;\text{ with }\; \mu_\ell(E)^2 = \mathrm{length}(E)\, \|\widehat{\Phi}_\ell - \Phi_\ell\|_{L^2(E)}^2 \tag{3.9}$$

can be computed by

```
eta = computeEstSlpEta(father2son,V_fine,x_fine,x_coarse);
```

and

```
mu = computeEstSlpMu(coordinates,elements,father2son,x_fine,x_coarse);
```

respectively. Here, `V_fine` is the Galerkin matrix for the uniformly refined mesh $\widehat{\mathcal{E}}_\ell$, and `x` and `x_fine` are the coefficient vectors for Φ_ℓ and $\widehat{\Phi}_\ell$, respectively. Then, `eta` $= \eta_\ell^2$ is a scalar and `mu` is a column vector with `mu(j)` $= \mu_\ell(E_j)^2$. The additional field `father2son` describes how to obtain $\widehat{\mathcal{E}}_\ell$ from the given mesh $\mathcal{E}_\ell$, cf. Section 4.3. An adaptive algorithm based on μ_ℓ is realized in the MATLAB script **`adaptiveSymmMu`** found in the subdirectory `demos/laplace/`.

The computation of Φ_ℓ can be avoided by taking the L^2-projection onto $\mathcal{P}^0(\mathcal{E}_\ell)$. This leads to a posteriori error estimators

$$\widetilde{\eta}_\ell = |||(1 - \Pi_\ell)\widehat{\Phi}_\ell|||_V \;\text{ and }\; \widetilde{\mu}_\ell^2 = \sum_{E \in \mathcal{E}_\ell} \widetilde{\mu}_\ell(E)^2 \;\text{ with }\; \widetilde{\mu}_\ell(E)^2 = \mathrm{length}(E)\, \|(1 - \Pi_\ell)\widehat{\Phi}_\ell\|_{L^2(E)}^2 \tag{3.10}$$

which are computed by

```
eta_tilde = computeEstSlpEtaTilde(father2son,V_fine,x_fine);
```

and

```
mu_tilde = computeEstSlpMuTilde(coordinates,elements,father2son,x_fine);
```

Then, `eta_tilde` $= \widetilde{\eta}_\ell^2$ and `mu_tilde` is a column vector with `mu_tilde(j)` $= \widetilde{\mu}_\ell(E_j)^2$.

An adaptive algorithm based on $\widetilde{\mu}_\ell$ is realized in the MATLAB script **`adaptiveSymmMuTilde`** found in the subdirectory `demos/laplace/`. Details are given in Section 5.7.1.

Another estimator which is based on the $h - h/2$ structure ist the two-level error estimator τ_ℓ from [27] which is called by

```
tau = computeEstSlpTau(father2son, V_fine, b_fine, x);
```

Here, `V_fine` is the Galerkin matrix for the uniformly refined mesh $\widehat{\mathcal{E}}_\ell$, and the vector `b_fine` is the corresponding right-hand side vector returned by the function **`buildSymmRHS`**. For the detailed definition of τ_ℓ, we refer to Section 9.4. The example file **`adaptiveSymmHierarchical`** implements an adaptive algorithm steered by τ_ℓ.

A completely different error estimation approach is implemented by the weighted-residual error estimator ρ_ℓ from [8], which is called by

```
ind = computeEstSlpResidual(coordinates,elements,x,gh);
```

where `x` is the coefficient vector of Φ_ℓ and `gh` is the discretized data returned by **`computeOscDirichlet`** or **`computeOscDirichletL2`**. The function can also be used to compute the estimator for the indirect formulation of Symm's integral equation and it can handle extra volume forces. For details, we refer to Section 9.7. The example files **`adaptiveSymmResidual`** and **`adaptiveSymmResidualIndirect`** implement adaptive algorithms steered by ρ_ℓ.

3.4.4. Numerical Solution of Neumann Problem. The Laplace problem with Neumann boundary condition (1.7) is equivalently recast in the hypersingular integral equation

$$Wg = (1/2 - K')\phi$$

with g being the unknown Dirichlet data and ϕ being the known Neumann data. We refer to Section 6 for details.

In the Galerkin formulation, we replace the Neumann data ϕ by its L^2-projection Φ_ℓ. Similar to above, the additional approximation error is controlled by the so-called Neumann data oscillations

$$\operatorname{osc}_{N,\ell}^2 = \sum_{E\in\mathcal{E}_\ell} \operatorname{osc}_{N,\ell}(E)^2 \quad \text{with} \quad \operatorname{osc}_{N,\ell}(E)^2 = \operatorname{length}(E)\|\phi - \Phi_\ell\|_{L^2(E)}^2 \tag{3.11}$$

introduced in [6]. The function call

```
[oscN,phih] = computeOscNeumann(coordinates,elements,phi);
```

returns a column vector with `oscN(j)` $= \operatorname{osc}_{N,\ell}(E_j)^2$ as well as the elementwise values `phih` of Φ_ℓ, cf. Section 6.1.

The right-hand side vector of the Galerkin scheme then takes the form

$$\mathbf{b} \in \mathbb{R}^N \quad \text{with} \quad \mathbf{b}_j = \langle (1/2 - K')\Phi_\ell \,,\, \zeta_j \rangle_{L^2(\Gamma)} \tag{3.12}$$

and is computed by

```
b = buildHypsingRHS(coordinates,elements,phih);
```

where the column vector `phih` provides the elementwise values of the L^2-projection Φ_ℓ (or another piecewise constant approximation of ϕ), see Section 6.4.

Since the hypersingular integral operator W is only semi-elliptic with kernel being the constant functions, we use the Galerkin matrix

$$\mathbf{W} + \mathbf{S} \in \mathbb{R}_{sym}^{N\times N} \quad \text{with} \quad \mathbf{S}_{jk} = \Big(\int_\Gamma \zeta_j \, d\Gamma\Big)\Big(\int_\Gamma \zeta_k \, d\Gamma\Big).$$

The stabilization matrix $\mathbf{S}$ is provided by

```
S = buildHypsingStabilization(coordinates,elements);
```

cf. Section 6.3.

In academic experiments, the exact solution $g \in H^1(\Gamma)$ is known, and

$$\operatorname{err}_{D,\ell}^2 + \operatorname{osc}_{N,\ell}^2 = \sum_{E\in\mathcal{E}_\ell} \operatorname{err}_{D,\ell}(E)^2 + \operatorname{osc}_{N,\ell}(E)^2 \tag{3.13}$$

with $\mathrm{err}_{D,\ell}(E)^2 = \mathrm{length}(E)\|(g - G_\ell)'\|^2_{L^2(E)}$ is an upper bound for the Galerkin error $\|\!|g - G_\ell|\!\|^2_{W+S}$ with respect to the energy norm $\|\!|\cdot|\!\|_{W+S}$. The function call

```
errD = computeErrDirichlet(coordinates,elements,x,g);
```

returns a column vector with `errD(j)` $= \mathrm{err}_{D,\ell}(E_j)^2$. Here, `g` is a function handle for the known exact solution and `x` is the nodal vector of the Galerkin solution $G_\ell = \sum_{j=1}^N x_j \zeta_j$. We refer to Section 6.5 for details.

For a posteriori error estimation and to steer an adaptive mesh-refinement, `HILBERT` includes four $(h - h/2)$-based error estimators from [16], discussed in Section 6.6 in detail: These read

$$\eta_\ell = \|\!|\widehat{G}_\ell - G_\ell|\!\|_{W+S} \quad \text{and} \quad \mu_\ell^2 = \sum_{E\in\mathcal{E}_\ell} \mu_\ell(E)^2 \text{ with } \mu_\ell(E)^2 = \mathrm{length}(E)\,\|(\widehat{G}_\ell - G_\ell)'\|^2_{L^2(E)}$$

$$\widetilde{\eta}_\ell = \|\!|(1 - I_\ell)\widehat{G}_\ell|\!\|_{W+S} \quad \text{and} \quad \widetilde{\mu}_\ell^2 = \sum_{E\in\mathcal{E}_\ell} \widetilde{\mu}_\ell(E)^2 \text{ with } \widetilde{\mu}_\ell(E)^2 = \mathrm{length}(E)\,\|((1 - I_\ell)\widehat{G}_\ell)'\|^2_{L^2(E)}$$

with I_ℓ the nodal interpolation onto $\mathcal{S}^1(\mathcal{T}_\ell)$. These are computed by

```
eta = computeEstHypEta(elem_fine,elements,father2son,WS_fine,x_fine,x);
eta_tilde = computeEstHypEtaTilde(elem_fine,elements,father2son,WS_fine,x_fine);
```

and

```
mu = computeEstHypMu(elem_fine,elements,father2son,x_fine,x);
mu_tilde = computeEstHypMuTilde(elem_fine,elements,father2son,x_fine);
```

Then, `eta` $= \eta_\ell^2$ as well as `eta_tilde` $= \widetilde{\eta}_\ell^2$ are scalars, whereas and `mu` as well as `mu_tilde` are column vectors with `mu(j)` $= \mu_\ell(E_j)^2$ and `mu_tilde(j)` $= \widetilde{\mu}_\ell(E_j)^2$, respectively. As input, these functions take `elements` and `elem_fine` which describe $\mathcal{E}_\ell$ and $\widehat{\mathcal{E}}_\ell$, respectively, as well as the link between both meshes given by `father2son`. `WS_fine` is the Galerkin matrix for $\widehat{\mathcal{E}}_\ell$. The vectors `x` and `x_fine` are the nodal vectors of the Galerkin solutions G_ℓ and $\widehat{G}_\ell$, respectively.

Adaptive algorithms based on μ_ℓ and $\widetilde{\mu}_\ell$ are implemented in the MATLAB scripts **`adaptiveHypMu`** and **`adaptiveHypMuTilde`** in the subdirectory `demos/laplace/`, cf. Section 6.7.1.

Similar to Symm's integral equation, we implement the two-level estimator τ_ℓ from [26, 23] in

```
tau = computeEstHypTau(elements_fine,elements,father2son,W_fine,b_fine,x);
```

For the detailed definition of τ_ℓ, we refer to Section 9.5. The vector `b_fine` is the right-hand side vector returned by the function **`buildHypsingRHS`**. The example file **`adaptiveHypHierarchical`** implements an adaptive algorithm steered by τ_ℓ. The weighted-residual error estimator ρ_ℓ from [8] for the hypersingular integral equation is called by

```
ind = computeEstHypResidual(coordinates,elements,gh,phih)
```

where `gh` describes the solution G_ℓ of (6.7) and `phih` is the discretized data returned by **`computeOscNeumann`**. The function can also be used to compute the estimator for the indirect formulation of the hypersingular integral equation and it can handle extra volume forces. For details, we refer to Section 9.8. The example files **`adaptiveHypResidual`** and **`adaptiveHypResidualIndirect`** implement an adaptive algorithms steered by ρ_ℓ.

3.4.5. Numerical Solution of Mixed Boundary Value Problem. The Laplace problem with mixed boundary condition (1.8) is equivalently recast in an integral equation which involves the Calderón projector, see Section 7. Our implementation is based on the functions provided for the Dirichlet and Neumann problem. Note that $\mathcal{E}_\ell$ is now given in terms of `coordinates`, `dirichlet`, and `neumann`.

In the problem formulation, the Dirichlet data g are only known on Γ_D. For the Galerkin formulation, one has to fix some extension $\overline{g}$ to Γ and to replace $\overline{g}$ by its nodal interpolation

G_ℓ. Since $\overline{g}$ is only implicitly built on the initial mesh $\mathcal{E}_0$, we need, however, to guarantee that $G_\ell|_{\Gamma_N} = \overline{g}|_{\Gamma_N}$. Moreover, the Neumann data ϕ are only known on Γ_N, and we extend ϕ implicitly by zero to a function $\overline{\phi}$ on the entire boundary Γ. For the Galerkin scheme, ϕ is then replaced by its L^2-projection Φ_ℓ onto the piecewise constants.

By replacing $(\overline{g}, \overline{\phi})$ by some approximation (G_ℓ, Φ_ℓ), we introduce an additional error which is controlled by the Dirichlet and Neumann data oscillations

$$\text{osc}_{D,\ell}^2 = \sum_{E\in\mathcal{E}_\ell} \text{osc}_{D,\ell}(E)^2 \quad \text{with} \quad \text{osc}_{D,\ell}^2 = \text{length}(E)\,\|(\overline{g} - G_\ell)'\|_{L^2(E)}^2, \tag{3.14}$$

$$\text{osc}_{N,\ell}^2 = \sum_{E\in\mathcal{E}_\ell} \text{osc}_{N,\ell}(E)^2 \quad \text{with} \quad \text{osc}_{N,\ell}^2 = \text{length}(E)\,\|\overline{\phi} - \Phi_\ell\|_{L^2(E)}^2, \tag{3.15}$$

see [6]. Our particular choice of G_ℓ and Φ_ℓ leads to $\text{osc}_{D,\ell}(E) = 0$ for $E \subseteq \overline{\Gamma}_N$ and $\text{osc}_{N,\ell}(E) = 0$ for $E \subseteq \overline{\Gamma}_D$, respectively. The function call

```
[oscD,oscN,gh,phih] = computeOscMixed(coordinates,neumann,dirichlet,g,phi);
```

returns the column vectors with the (squared) data oscillations on Γ_D and Γ_N as well as vectors `gh` and `phih` which provide the nodal vector of G_ℓ and the elementwise values of Φ_ℓ, respectively.

As has been pointed out before, this function call must not be used if the mesh $\mathcal{E}_\ell$ is obtained by refinement of $\mathcal{E}_{\ell-1}$, since otherwise the chosen prolongation $\overline{g}$ is changed. Instead, one may use the function call

```
[oscD,oscN,gh,phih] = computeOscMixed(coordinates,neumann,dirichlet, ...
                                      father2neumann,neumann_old,gh_old,g,phi);
```

where the relation between $\mathcal{E}_{\ell-1}$ and $\mathcal{E}_\ell$ is given by `neumann_old` and `father2neumann` and where `gh_old` provides the nodal vector of $\overline{g}$ with respect to $\mathcal{E}_{\ell-1}$. Details are given in Section 7.1.

To re-use the functions implemented for the hypersingular integral equation from the previous section, we have to guarantee that the first N_N nodes of $\mathcal{K}_\ell$ belong to $\overline{\Gamma}_N$. This possibly needs some reordering of `coordinates` as well as some corresponding update of `dirichlet` and `neumann`. This is done by

```
[coordinates,neumann,dirichlet] = buildSortedMesh(coordinates,neumann,dirichlet);
```

We stress that the ordering of `dirichlet` and `neumann`, i.e., the numbering of the elements $\mathcal{E}_\ell = \{E_1, \dots, E_N\}$ is not affected. Details are found in Section 4.2.

The right-hand side vector $\mathbf{b}$ for the Galerkin formulation, split into contributions on Γ_N and contributions on Γ_D, is computed by

```
[bN,bD] = buildMixedRHS(coordinates,dirichlet,neumann,V,K,W,gh,phih);
```

Here, `gh` is the nodal vector of the extended Dirichlet data, and `phih` provides the nodal values of the known Neumann data Φ_ℓ. The matrices `V`, `K`, and `W` are the discrete integral operators associated with $\mathcal{E}_\ell$.

Adaptive algorithms from [6] based on the μ_ℓ and $\widetilde{\mu}_\ell$-estimators from the previous sections are provided by **`adaptiveMixedMu`** and **`adaptiveMixedMuTilde`** in the folder `demos/laplace/`.

Again, we provide the two-level estimator τ_ℓ in

```
ind = computeEstMixTau(father2dirichlet, father2neumann,neumann,...
      neumann_fine, V_fine, K_fine, W_fine, bD_fine, bN_fine, x,...
      free_neumann,free_neumann_fine);
```

For the detailed definition of τ_ℓ, we refer to Section 9.6. The vectors `bD_fine` and `bN_fine` are the right-hand side vectors returned by the function **`buildMixedRHS`** corresponding to the fine mesh $\widehat{\mathcal{E}}_\ell$. The vectors `free_neumann` and `free_neumann_fine` indicate the degrees of freedom on the Neumann boundary. For details on this vectors, consider Section 9.6. The example file **`adaptiveMixedHierarchical`** implements an adaptive algorithm steered by τ_ℓ.

The weighted-residual error estimator ρ_ℓ for the mixed problem is

```
[indSLP,indHYP] = computeEstMixResidual(coordinates,dirichlet,neumann,...
                  uNh,phiDh,uDh,phiNh);
```

where the discretized data `uDh`, `phiNh` is returned by **`computeOscMixed`** and `uNh`, `phiDh` represents the solution vectors (see Section 9.9 for details). The function can also be used in case of non-homogeneous volume forces. For details, we refer to Section 9.8. The example file **`adaptiveMixedResidual`** implements an adaptive algorithm steered by ρ_ℓ.

3.5. Non-Homogeneous Volume Force. Compared with the the homogeneous case $f = 0$, the non-homogeneous case

$$-\Delta u = f \quad \text{in } \Omega \tag{3.16}$$

only leads to additional contributions on the right-hand side of the integral equations. Throughout, we replace f by its L^2-projection $F_\ell \in \mathcal{P}^0(\mathcal{T}_\ell)$, where $\mathcal{T}_\ell$ is a triangulation of Ω. The additional approximation error is controlled by the volume data oscillations

$$\operatorname{osc}_{\Omega,\ell}^2 = \sum_{T\in\mathcal{T}_\ell} \operatorname{osc}_{\Omega,\ell}(T)^2 \quad \text{with} \quad \operatorname{osc}_{\Omega,\ell}^2 = \operatorname{area}(T)\,\|f - F_\ell\|_{L^2(T)}^2, \tag{3.17}$$

cf. [6]. The function call

```
[oscV,fh] = computeOscVolume(vertices,triangles,f)
```

returns a column vector `oscV(j)` $= \operatorname{osc}_{\Omega,\ell}(T_j)^2$ as well as the vector `fh` of elementwise values of F_ℓ. Details are found in Section 8.3.

For the Dirichlet problem, the right-hand side for the Galerkin scheme is then returned by

```
b = buildSymmVolRHS(coordinates,elements,gh,vertices,triangles,fh);
```

see Section 8.1 and Section 8.4.

For the Neumann problem, the right-hand side formally includes the computation of the normal derivative $N_1 f$ of the Newton potential $\widetilde{N} f$. This can, however, be avoided by using an additional weakly-singular integral equation, see Section 8.7. The right-hand side vector for the Galerkin scheme is returned by

```
[b,lambda] = buildHypsingVolRHS(coordinates,elements,gh,vertices,triangles,fh);
```

where the additional return value `lambda` provides an approximation of some λ with $-N_1 f = (1/2 - K')\lambda$, cf. Section 8.8.

For the mixed boundary value problem with Dirichlet-Neumann boundary conditions, the right-hand side is built by

```
[bN,bD,lambda] = buildMixedVolRHS(coordinates,dirichlet,neumann,V,K,W,gh,phih, ...
                 vertices,triangles,fh);
```

where the additional return value `lambda` arises for the same reason as for the hypersingular integral equation with non-homogeneous volume force. Details are found in Section 8.11 and Section 8.12.

The two-level error estimators τ_ℓ as well as the weighted residual error estimators ρ_ℓ are also available for Symm's integral equation, the hypersingular integral equation, as well as the mixed problem with non-homogeneous volume forces. For the correct function calls, please consider Sections 9.4–9.6 for the two-level estimators and 9.7–9.9 for the weighted-residual error estimators.

The folder `demos/poisson/` contains various demo files based on different a posteriori error estimators for a model problem with non-homogeneous volume force, see also Section 3.7.

3.6. Visualization of Discrete Solutions. Provided that Γ is connected, the function

```
plotArclengthP0(coordinates,elements,phih [,phi] [,figure])
```

plots a discrete solution $\Phi_\ell \in \mathcal{P}^0(\mathcal{E}_\ell)$ over the arclength. The elementwise values of Φ_ℓ are provided by the column vector `phih`. With the optional function handle `phi` the exact solution ϕ can be plotted into the same plot for comparison. With the function

```
plotArclengthS1(coordinates,elements,gh [,g] [,figure])
```

one can plot a discrete solution $G_\ell \in \mathcal{S}^1(\mathcal{E}_\ell)$ over the arclength. The nodal values of G_ℓ are provided by the column vector `gh`. With the optional function handle `g` the exact solution g can be plotted into the same plot for comparison.

For both functions, the optional parameter `figure` prescribes the figure number for the plot.

3.7. Numerical Examples. `HILBERT` contains two different examples and different demo files. For the first example, contained in `demos/laplace/`, the exact solution u of

$$-\Delta u = 0 \quad \text{in } \Omega \tag{3.18}$$

is prescribed in polar coordinates as

$$u(r,\varphi) = r^{2/3}\cos(2\varphi/3), \tag{3.19}$$

and Ω is a rotated L-shaped domain with $\operatorname{diam}(\Omega) < 1$ and reentrant corner with angle $3\pi/2$. The rotation is done in a way that the Dirichlet data $g = u|_\Gamma$ are smooth, whereas the Neumann data $\phi = \partial_n u$ have a generic singularity at the reentrant corner. Based on this setting, the folder `demos/laplace/` contains different demo files to solve the associated integral equations by adaptive algorithms, which basically differ in the a-posteriori error estimator which is used. For Symm's integral equation (cf. Section 3.4.3), there are 5 different demo files:

- **`adaptiveSymmHierarchical`** uses the 2-level error estimator τ from Section 9.4,
- **`adaptiveSymmMu`** uses the $h - h/2$ error estimator μ from Section 5.6.3,
- **`adaptiveSymmMuTilde`** uses the $h - h/2$ error estimator $\widetilde{\mu}$ from Section 5.6.4,
- **`adaptiveSymmResidual`** uses the weighted-residual error estimator ρ from Section 9.7.

Additionally, the file

- **`adaptiveSymmResidualIndirect`**

uses the weighted-residual error estimator to solve Symm's integral equation $V\phi = g$ on the screen $[-1,1]$ with $g(x) = -x$, where the exact exact solution is known to be $\phi(x) = -2x/\sqrt{(1-x^2}$.

For the hypersingular integral equation (cf. Section 3.4.4), there are also 5 different demo files:

- **`adaptiveHypHierarchical`** uses the 2-level error estimator τ from Section 9.5,
- **`adaptiveHypMu`** uses the $h - h/2$ error estimator μ from Section 6.6.3,
- **`adaptiveHypMuTilde`** uses the $h - h/2$ error estimator $\widetilde{\mu}$ from Section 6.6.4,
- **`adaptiveHypResidual`** uses the weighted-residual error estimator ρ from Section 9.8.

Additionally, the file

- **`adaptiveHypResidualIndirect`**

uses the weighted-residual error estimator to solve the hypersingular integral equation $Wg = \phi$ on the screen $[-1,1]$ with $\phi(x) = 1$, where the exact exact solution is known to be $g(x) = 2\sqrt{(1-x^2}$.

For the mixed boundary value problem (cf. Section 3.4.5), there are 4 different demo files:

- **`adaptiveMixedHierarchical`** uses the 2-level error estimator τ from Section 9.6,
- **`adaptiveMixedMu`** uses the $h - h/2$ error estimator μ from Section 7.6,
- **`adaptiveMixedMuTilde`** uses the $h - h/2$ error estimator $\widetilde{\mu}$ from Section 7.6,
- **`adaptiveMixedResidual`** uses the weighted-residual error estimator ρ from Section 9.9.

The folder `demos/poisson/` features the model problem

$$-\Delta u = f \quad \text{in } \Omega,$$

where we prescribe the exact solution in polar coordinates by

$$u(r,\varphi) = |(r,\varphi) - z|^{1.8} + r^{2/3}\cos(2\varphi/3),$$

with $z = (0.14, 0.14)$. Again, Ω is a rotated and scaled L-shape. Adaptive algorithms based on different a posteriori error estimators demonstrate the use of `HILBERT` in this case. The provided files are

- **`adaptiveHypVolMuTilde`**
- **`adaptiveHypVolResidual`**
- **`adaptiveMixedVolMuTilde`**
- **`adaptiveMixedVolResidual`**
- **`adaptiveSymmVolHierarchical`**
- **`adaptiveSymmVolMu`**
- **`adaptiveSymmVolMuTilde`**
- **`adaptiveSymmVolResidual`**

The names again indicate which a posteriori error estimators are used for the adaptive algorithms.

4. Mesh-Refinement

Listing 1. Dörfler Marking for Local Mesh-Refinement

```
function varargout = markElements(theta,varargin)
%*** check whether optional parameter rho is given or not
if nargin == nargout +1
    rho = 0;
else
    rho = varargin{1};
    varargin = varargin(2:end);
end

%*** enforce input parameters to be column vectors and count their length
nE = zeros(1,nargout+1);
for j = 1:nargout
    nE(j+1) = length(varargin{j});
    varargin{j} = reshape(varargin{j},nE(j+1),1);
end

%*** generate set of all indicators
indicators = cat(1,varargin{:});
nE = cumsum(nE);

%*** realization of Doerfler marking
[indicators,idx] = sort(indicators,'descend');
sum_indicators = cumsum(indicators);
ell = max(ceil(rho*nE(end)),find(sum_indicators>=sum_indicators(end)*theta,1));
marked = idx(1:ell);

%*** split subset marked into subsets with respect to input vectors
for j = 1:nargout
    varargout{j} = marked( marked>nE(j) & marked<=nE(j+1) ) - nE(j);
end
```

4.1. Dörfler Marking (5.34) for Local Mesh-Refinement (Listing 1). We realize the marking criterion proposed by DÖRFLER [13] in a generalized form which is suitable even for mixed boundary value problems or the FEM-BEM coupling. Suppose that $\mathcal{E}_\ell^{(1)}, \dots, \mathcal{E}_\ell^{(n)}$ are pairwise disjoint meshes which provide indicators $\varrho_\ell^{(k)}(E)$ for all $E \in \mathcal{E}_\ell^{(k)}$. We formally define $\mathcal{E}_\ell := \bigcup_{j=1}^n \mathcal{E}_\ell^{(j)}$ and $\varrho_\ell(E) := \varrho_\ell^{(k)}(E)$ for all $E \in \mathcal{E}_\ell^{(k)}$ and $k = 1, \dots, n$. For given $0 < \theta < 1$, we then aim to find the minimal set $\mathcal{M}_\ell \subseteq \mathcal{E}_\ell$ such that

$$\theta \sum_{E \in \mathcal{E}_\ell} \varrho_\ell(E) \le \sum_{E \in \mathcal{M}_\ell} \varrho_\ell(E). \tag{4.1}$$

Finally, we define and return $\mathcal{M}_\ell^{(k)} := \mathcal{M}_\ell \cap \mathcal{E}_\ell^{(k)}$ for all $k = 1, \dots, n$.

A second generalization is concerned with the minimal cardinality of $\mathcal{M}_\ell$. For analytical convergence results, the minimal set $\mathcal{M}_\ell$ with (4.1) is sufficient. However, small sets $\mathcal{M}_\ell$ lead to many iterations in the adaptive loop and may thus lead to a large runtime. With an additional parameter $0 < \rho < 1$, one remedy for this drawback can be to determine the minimal superset $\mathcal{M}_\ell \subseteq \overline{\mathcal{M}}_\ell \subseteq \mathcal{E}_\ell$ with

$$\frac{\#\overline{\mathcal{M}}_\ell}{\#\mathcal{E}_\ell} \ge \rho \quad \text{and} \quad \varrho_\ell(E) \ge \varrho_\ell(E') \quad \text{for all } E \in \overline{\mathcal{M}}_\ell \text{ and } E' \in \mathcal{E}_\ell \backslash \overline{\mathcal{M}}_\ell. \tag{4.2}$$

From an analytical point of view, any superset $\overline{\mathcal{M}}_\ell$ of $\mathcal{M}_\ell$ also leads to a convergent adaptive algorithm. Our definition of $\overline{\mathcal{M}}_\ell$ guarantees that at least a fixed percentage of elements is refined and that these elements have the largest associated refinement indicators. Note that the parameter ρ gives a lower bound for the percentage of elements which are refined.

Our implementation of the marking criterion includes (optionally) the generalizations (4.1)–(4.2) of the original strategy from [13]:

- In the simplest case, the function **`markElements`** is called by

  ```
  marked = markElements(theta,indicator)
  ```

  where `indicator` is a column vector, where `indicator(j)` corresponds to some element $E_j \in \mathcal{E}_\ell$. The function **`markElements`** then returns the indices corresponding to the minimal set $\mathcal{M}_\ell$.
- Alternatively, the function can be called by

  ```
  marked = markElements(theta,rho,indicator)
  ```

  and returns the indices corresponding to the minimal set $\overline{\mathcal{M}}_\ell \supseteq \mathcal{M}_\ell$ with (4.2).
- For the general formulation described above, the function is called by

  ```
  [marked1,marked2,...] = markElements(theta [,rho], ind1,ind2,...)
  ```

  where, e.g., `ind1` is the vector of indicators $\varrho_\ell(E_j^{(1)})$ for all $E_j^{(1)} \in \mathcal{E}_\ell^{(1)} = \{E_1^{(1)}, \dots, E_{N^{(1)}}^{(1)}\}$. The function returns the indices corresponding to the sets $\mathcal{M}_\ell^{(k)} \subseteq \mathcal{E}_\ell^{(k)}$ (or $\overline{\mathcal{M}}_\ell^{(k)}$ if ρ is given).
- First, we check whether the parameter ρ is given. If not, it is set to 0 (Line 3–8).
- The given indicator vectors are reshaped into column vectors, and their length is stored in the vector `nE` (Line 11–15).
- We build the vector of all indicators $\varrho_\ell(E_j)$ (Line 18) which corresponds to the ordered set $\mathcal{E}_\ell = \{E_1, \dots, E_N\}$. Moreover, `nE` becomes a vector of pointers so that `nE(`j`)`$+1$ and `nE(`$j+1$`)` give the start and the end of $\mathcal{E}_\ell^{(j)}$ with respect to `indicators` (Line 19).
- To determine the minimal set $\mathcal{M}_\ell$ we sort the vector `indicators` (Line 22). Mathematically, this corresponds to finding a permutation π such that $\varrho_\ell(E_{\pi(j)}) \ge \varrho_\ell(E_{\pi(j+1)})$. We then compute the vector `sum_indicators` of sums $\sum_{j=1}^k \varrho_\ell(E_{\pi(j)})$ (Line 23). Note that `sum_indicators(end)` contains $\sum_{j=1}^N \varrho_\ell(E_{\pi(j)}) = \sum_{j=1}^N \varrho_\ell(E_j)$. Finding the minimal set $\mathcal{M}_\ell$ is thus equivalent to finding the minimal index k with $\theta \sum_{j=1}^N \varrho_\ell(E_{\pi(j)}) \le$

$\sum_{j=1}^{k} \varrho_\ell(E_{\pi(j)})$, and there holds $\mathcal{M}_\ell = \{E_{\pi(1)}, \dots, E_{\pi(k)}\}$. If ρ is specified, we choose the minimal index $\overline{k} \geq k$ with $\overline{k} \geq \rho N$. Altogether, Line 24–25 thus determines the indices of elements in $\mathcal{M}_\ell$ and $\overline{\mathcal{M}}_\ell$, respectively.

- Finally, we use the pointer vector `nE` to determine the indices of $\mathcal{M}_\ell^{(k)}$ with respect to $\mathcal{E}_\ell^{(k)} = \{E_1^{(k)}, ..., E_{N^{(k)}}^{(k)}\}$ (Line 28–30).

LISTING 2. Sorting Routine for Various Meshes

```
1  function [varargout] = buildSortedMesh(varargin)
2  %*** CASE 2 is covered by recursion to CASE 3, i.e., bdry mesh is generated
3  if nargin == 2
4      %*** only volume mesh is given
5      vertices = varargin{1};
6      triangles = varargin{2};
7
8      %*** create list of edges and sort node numbers per edge
9      edges = [triangles(:,[1 2]); triangles(:,[2 3]); triangles(:,[3 1])];
10     edges_sorted = sort(edges,2);
11
12     %*** an edge is a boundary edge iff it appears once in edges_sorted
13     [foo,sort2unique,unique2sort] = unique(edges_sorted,'rows');
14     idx = sort2unique(accumarray(unique2sort,1) == 1);
15     elements = edges(idx,:);
16
17     %*** recursive call of buildSortedMesh, elements is now bdry part
18     if nargout == 3
19         %*** return value = [vertices,triangles,elements]
20         [varargout{1},varargout{2},varargout{3}] ...
21             = buildSortedMesh(vertices,triangles,elements);
22     elseif nargout == 4
23         %*** return value = [vertices,triangles,coordinates,elements]
24         [varargout{1},varargout{2},varargout{3},varargout{4}] ...
25             = buildSortedMesh(vertices,triangles,elements);
26     end
27
28     return;
29 end
30
31 %
32 %*** from now on, only CASE 1 and CASE 3 have to be considered
33 %
34
35 %*** second input parameter is volume mesh
36 if size(varargin{2},2) == 3
37     input_offset = 3;
38 else
39     input_offset = 2;
40 end
41
42 %*** user wants coordinates of bdry parts
43 if nargout == nargin + 1
44     output_offset = 1;
45 else
46     output_offset = 0;
47 end
48
```

```
%*** all nodes which belong to the volume and boundary mesh
nodes = varargin{1};
nN = size(nodes,1);

%*** build list of all boundary nodes after appearance
list_nodes = [];
for i = input_offset:nargin
    current_nodes = unique(varargin{i});
    list_nodes = [ list_nodes ; setdiff(current_nodes,list_nodes) ];
end

%*** number of boundary nodes
nC = size(list_nodes,1);

%*** add to list the remaining nodes which are not on the boundary
list_nodes = [list_nodes;setdiff(1:nN,list_nodes)'];

%*** build permutation such that vertices are sorted according to boundary parts
[foo,permutation] = sort(list_nodes);

%*** permute indices of nodes such that boundary nodes are first
nodes(permutation',:) = nodes;
varargout{1} = nodes;

%*** if coordinates of bdry nodes are wanted, return varargout{3} = coordinates
if output_offset == 1
    varargout{3} = nodes(1:nC,:);
end

%*** if a volume mesh was given, return varargout{2} = triangles
if input_offset == 3
    varargout{2} = permutation(varargin{2});
end

%*** update and return boundary parts
for i = input_offset:nargin
    varargout{i + output_offset} = permutation(varargin{i});
end
```

4.2. Sorting Routine for Various Meshes (Listing 2). In some cases, HILBERT implicitly assumes that the mesh is ordered in a special way. These cases are:

- **Solving a mixed problem.** In this case, we want to re-use functions which were developed for the hypersingular integral equation. Therefore, the mesh needs to be ordered such that the nodes on the Neumann boundary appear first in the vector `coordinates`.
- **Solving a problem with a non-homogeneous volume force.** There, we only want to provide the coarsest mesh in terms of `vertices` and `triangles`. It is necessary to extract the boundary mesh and enforce an ordering of the vertices such that the nodes on the boundary appear first.

The routine **`buildSortedMesh`** of listing 2 provides such functionality. Let us give some particular examples for various input/output parameters.

1. Suppose that a volume mesh is given by `vertices` and `volumes`. Then,

   ```
   [vertices,triangles,elements] = buildSortedMesh(vertices,triangles)
   ```

   sorts the array `vertices` such that the nodes on the boundary appear first, and `triangles` are updated as well. The array `elements` contains all edges on the boundary. The call of

```
[vertices,triangles,coordinates,elements] = buildSortedMesh(vertices,triangles)
```

additionally yields the array `coordinates` which contains all nodes on the boundary. Note that `coordinates(1:nG,:) = vertices(1:nG,:)`, when `nG` is the number of nodes on the boundary.

2. Suppose that a volume mesh is given by `vertices` and `volumes`. Additionally, we have two boundary parts given by the arrays `dirichlet` and `neumann`. Then, the call of

```
[vertices,triangles,neumann,dirichlet] ...
    = buildSortedMesh(vertices,triangles,neumann,dirichlet)
```

sorts the array `vertices` such that the nodes on `neumann` appear first, the nodes of `dirichlet` appear second, and the inner nodes appear last. The arrays `triangles`, `neumann` and `dirichlet` are updated as well. Additionally, one could ask for `coordinates` as output parameter.

3. Suppose that a boundary mesh is given by `coordinates` and the arrays `neumann` and `dirichlet`. Then, the call of

```
[coordinates,neumann,dirichlet] = buildSortedMesh(coordinates,neumann,dirichlet)
```

sorts the array `coordinates` such that the nodes in `neumann` appear first. The arrays `neumann` and `dirichlet` are updated as well.

- We point out that in the cases 2. and 3. two boundary parts `neumann` and `dirichlet` were given for ease of presentation. The function **`buildSortedMesh`** can deal with any finite number of boundary parts `bdry_1`, ..., `bdry_n`.

Let us discuss the implementation of Listing 2:

- If two input parameters are given, then we have to deal with Case 1. First, input parameters are loaded (Line 5–6). Then, we determine the array `elements`, which contains the array of boundary edges (Line 9–15). Finally, we choose the output parameters (Line 18 and 22), and make a recursive call of **`buildSortedMesh`**, where we now take `elements` as boundary part (Line 21 and 25).
- After Line 36, we deal with the Case 2 and Case 3. We check whether a volume mesh is given (Line 36–40) and whether we have to return `coordinates` (Line 43–47).
- All nodes are collected (Line 50).
- We build an array `list_nodes`, which contains all nodes of the given boundary parts `bdry_1`, ..., `bdry_n`, such that the nodes appear in the order which is implied by the ordering of the boundary parts, i.e, the nodes on `bdry_1` appear first, the nodes of `bdry_2` appear second and so on. Note that we use the `setdiff` command because the boundary parts `bdry_i` and `bdry_i+1` may share (at most) 2 nodes (Line 54–58).
- We add to the array `list_nodes` all inner nodes (Line 64).
- In Line 67, we sort `list_nodes` and obtain the important vector `permutation` such that

```
list_nodes(permutation(i)) = i
```

and permute the array `nodes`. Note that, at this point, `nodes` could be `coordinates` or `vertices`. The array `nodes` is permuted such that the ith entry in node, `node(i,:)`, then is shifted to the `permutation(i)`th position (Line 70–71).

- If the user asks for boundary coordinates, return the first `nC` entries in `nodes` (Line 74–76). If the function was given a volume triangulation, then we have to permute the volume triangulation according to `permutation` (Line 79–81). Finally, return the updated boundary parts (Line 84–86).

LISTING 3. Local Refinement of Boundary Element Mesh

```
1 function [coordinates,varargout] = refineBoundaryMesh(coordinates,varargin)
2 %*** fix the blow-up factor for the K-mesh constant,
3 %*** where we assume C(Mesh_0) = 1, i.e., the initial mesh is uniform
```

```
4  kappa = 2;
5
6  %*** count number of boundary parts from input
7  %*** nB will hold this number
8  nB = 0;
9
10 for iter = 1 : (nargin - 1)
11
12     if size(varargin{iter},2) == 2
13         nB = nB + 1;
14         nE_boundary(iter) = size(varargin{iter},1);
15     else
16         break;
17     end
18
19 end
20
21 %*** check the correct number of input parameters
22 if ~( (nargin == (nB+1)) || (nargin == (2*nB+1)) )
23     error('refineBoundaryMesh: Wrong number of input arguments!');
24 end
25
26 %*** check the correct number of output parameters
27 if ~( (nargout == (nB+1)) || (nargout == (2*nB+1)) )
28     error('refineBoundaryMesh: Wrong number of output arguments!');
29 end
30
31 %*** check, if user asks for father2son fields in output
32 if nargout == (2*nB+1)
33     output_father2son = true;
34 else
35     output_father2son = false;
36 end
37
38 %*** obtain set of all elements of the boundary partition
39 elements = cat(1,varargin{1 : nB});
40
41 %*** indices of a boundary part w.r.t. entire field elements
42 ptr_boundary = cumsum([0,nE_boundary]);
43
44 %*** 1. determine whether uniform or adaptive mesh-refinement
45 %*** 2. in case of adaptive mesh-refinement compute vector marked
46 %***    of marked elements w.r.t. entire field elements
47 if (nB+1) == nargin
48     refinement = 'uniform';
49 else
50     refinement = 'adaptive';
51     marked = zeros(0,1); % marked elements w.r.t. entire field elements
52
53     for iter = 1 : nB
54         marked = [marked; varargin{iter + nB} + ptr_boundary(iter)];
55     end
56
57 end
58
59 nC = size(coordinates,1); % number of coordinates
60 nE = size(elements,1);    % number of elements
```

```
61
62  if strcmp(refinement,'adaptive')
63
64      %*** if element Ej is marked and if its neighbour Ek satisfies
65      %*** hk >= kappa*hj, we (recursively) mark Ek for refinement as well
66
67      %*** marked elements Ej will be refined, i.e., flag(j) = 1
68      flag = zeros(nE,1);
69      flag(marked) = 1;
70
71      %*** determine neighbouring elements
72      node2element = zeros(nC,2);
73      node2element(elements(:,1),2) = (1:nE)';
74      node2element(elements(:,2),1) = (1:nE)';
75      element2neighbour = [ node2element(elements(:,1),1), ...
76                            node2element(elements(:,2),2) ];
77
78      %*** compute (squared) local mesh-size
79      h = sum((coordinates(elements(:,1),:)-coordinates(elements(:,2),:)).^2,2)';
80
81      %*** the formal recursion is avoided by sorting elements by mesh-size
82      [tmp,sorted_elements] = sort(h);
83      for j = sorted_elements
84          if flag(j)
85              neighbours = element2neighbour(j,:);
86              neighbours = neighbours( find(neighbours) );
87              flag( neighbours(h(neighbours) >= kappa*h(j)) ) = 1;
88          end
89      end
90
91      %*** obtain vector of marked elements
92      marked = find(flag);
93      nM = length(marked);
94
95      %*** compute and add new nodes
96      coordinates = [coordinates;zeros(nM,2)];
97      coordinates((1:nM)+nC,:) = ( coordinates(elements(marked,1),:) ...
98                                   + coordinates(elements(marked,2),:) )*0.5;
99
100     %*** refinement of mesh iterates over each boundary part
101     for iter = 1:nB
102
103         %*** determine which marked elements belong to boundary part
104         idx = find( (ptr_boundary(iter) < marked) ...
105                     & (marked <= ptr_boundary(iter+1)) );
106         nM_boundary = length(idx);
107
108         %*** allocate new elements
109         new_elements = [varargin{iter};zeros(nM_boundary,2)];
110
111         %*** generate new elements
112         new_elements((1:nM_boundary)+nE_boundary(iter),:) ...
113             = [ nC + idx, elements(marked(idx),2) ];
114         new_elements( marked(idx) - ptr_boundary(iter),2 ) = nC + idx;
115
116         %*** add new_elements and father2son to output
117         varargout{iter} = new_elements;
```

```
118
119         %*** compute father2son only if desired
120         if output_father2son == true
121
122             %*** generate father2son
123             father2son = repmat((1:nE_boundary(iter))',1,2);
124             father2son( marked(idx) - ptr_boundary(iter),2 ) ...
125                 = (1:nM_boundary)' + nE_boundary(iter);
126
127             %*** add new_elements and father2son to output
128             varargout{nB+iter} = father2son;
129         end
130
131     end
132
133 elseif strcmp(refinement,'uniform')
134
135     %*** compute and add new nodes
136     coordinates = [coordinates;zeros(nE,2)];
137     coordinates((nC+1):end,:) = ( coordinates(elements(:,1),:) ...
138                                 + coordinates(elements(:,2),:) )*0.5;
139
140     %*** uniform refinement of mesh iterates over each boundary part
141     for iter = 1:nB
142
143         %*** generate new elements
144         idx = (ptr_boundary(iter)+1):ptr_boundary(iter+1);
145         varargout{iter} = [ varargin{iter}(:,1),nC + idx' ; ...
146                             nC + idx', varargin{iter}(:,2) ];
147
148         %*** compute father2son only if desired
149         if output_father2son == true
150
151             %*** build father2son
152             varargout{nB+iter} ...
153                 = [(1:nE_boundary(iter))', ...
154                    (1:nE_boundary(iter))' + nE_boundary(iter)];
155         end
156
157     end
158 end
```

4.3. Local Refinement of Boundary Element Mesh (Listing 3). In many cases, one is not interested in computing only one approximation U with respect to a fixed given boundary element mesh $\mathcal{E}$, but in computing a sequence of more and more accurate approximations U_ℓ corresponding to a sequence $\mathcal{E}_\ell$ of boundary element meshes with decreasing mesh-sizes. To that end, our software package `HILBERT` provides an efficient mesh-refinement **`refineBoundaryMesh`** for boundary element meshes, which covers the following tasks:

- uniform refinement of a given mesh
- refinement of certain marked elements, specified by the user
- linkage between elements of the input mesh with elements of the refined mesh
- handling of meshes that are split into finitely many distinct parts, e.g., $\Gamma = \overline{\Gamma}_D \cup \overline{\Gamma}_N$
- guaranteed boundedness of the K-mesh constant

Throughout, refinement of an element means that E_i is bisected into two elements e_j, e_k of half length. We now discuss certain aspects of our implementation from Listing 3, where the data

structure of `coordinates`, `elements`, `dirichlet`, and `neumann` is described in Section 3.3 above. The main focus is, however, on the practical use of the function.

- **Input/Output Parameters:** To allow a partition of Γ into finitely many parts (e.g., a Dirichlet and a Neumann boundary), the formal signature reads

```
[coordinates,varargout] = refineBoundaryMesh(coordinates,varargin)
```

To explain the variable input/output parameters, we consider certain examples.

- Suppose that $\mathcal{E}_\ell = \{E_1, \dots, E_N\}$ is described by `coordinates` and `elements`. Then,

```
[coordinates_fine,elements_fine,father2son] ...
    = refineBoundaryMesh(coordinates,elements)
```

provides the uniformly refined mesh $\widehat{\mathcal{E}}_\ell = \{e_1, \dots, e_{2N}\}$, where each element $E_i \in \mathcal{E}_\ell$ is bisected in certain sons $e_j, e_k \in \widehat{\mathcal{E}}_\ell$. The $(N \times 2)$-matrix `father2son` provides a link between the element indices in the sense that

`father2son`$(i,:) =$ `[` j, k `]` for $E_i = e_j \cup e_k$.

The output parameter `father2son` is optional and can be omitted.

- Suppose that $\mathcal{M}_\ell \subseteq \mathcal{E}_\ell$ is a set of elements which are marked for refinement. Let `marked` be an $(M \times 1)$-column vector containing the indices of the elements in $\mathcal{M}_\ell$. Then,

```
[coordinates_fine,elements_fine,father2son] ...
    = refineBoundaryMesh(coordinates,elements,marked)
```

provides a mesh $\mathcal{E}_{\ell+1}$ which is only refined locally in the sense that all elements of $\mathcal{M}_\ell$ are refined. If an element $E_i \in \mathcal{E}_\ell$ is not refined, there holds $E_i = e_j \in \mathcal{E}_{\ell+1}$, where the link between these indices is given by

`father2son`$(i,:) =$ `[` j, j `]` for $E_i = e_j$.

Again, the output parameter `father2son` is optional and can be omitted.

- Suppose that Γ is split into a Dirichlet boundary Γ_D and a Neumann boundary Γ_N. In this case, the mesh $\mathcal{E}_\ell = \{E_1, \dots, E_N\}$ is described in terms of `coordinates`, `dirichlet`, and `neumann`, cf. Section 3.3. Then,

```
[coordinates_fine,dirichlet_fine,neumann_fine,dir2son,neu2son] ...
    = refineBoundaryMesh(coordinates,dirichlet,neumann)
```

provides the uniformly refined mesh $\widehat{\mathcal{E}}_\ell$. As `father2son` in the previous cases with a single boundary part, now the arrays `dir2son` and `neu2son` provide the link between the coarse mesh parts and the refined ones, e.g., `dirichlet` and `dirichlet_fine`. For instance, suppose that $\mathcal{E}_\ell|_{\Gamma_D} = \{E_1^D, \dots, E_{N_D}^D\}$ and $\widehat{\mathcal{E}}_\ell|_{\Gamma_D} = \{e_1^D, \dots, e_{2N_D}^D\}$. Then, there holds

`dir2son`$(i,:) =$ `[` j, k `]` for $E_i^D = e_j^D \cup e_k^D$.

Finally, the fields `dir2son` and `neu2son` are optional in the sense that they can either both be asked for or both be omitted.

- Suppose that $\mathcal{M}_\ell^D \subseteq \mathcal{E}_\ell|_{\Gamma_D}$ and $\mathcal{M}_\ell^N \subseteq \mathcal{E}_\ell|_{\Gamma_N}$ are sets of marked elements. Let `marked_dirichlet` and `marked_neumann` be $(M_D \times 1)$- and $(M_N \times 1)$-column vectors containing the indices of the elements in $\mathcal{M}_\ell^D$ and $\mathcal{M}_\ell^N$, respectively. Then,

```
[coordinates_fine,dirichlet_fine,neumann_fine,dir2son,neu2son] ...
    = refineBoundaryMesh(coordinates,dirichlet,neumann, ...
                         marked_dirichlet,marked_neumann)
```

provides a mesh $\mathcal{E}_{\ell+1}$ which is only refined locally in the sense that all elements of $\mathcal{M}_\ell^D \cup \mathcal{M}_\ell^N$ are refined. We stress that the optional input `marked_dirichlet` and `marked_neumann` can either both be given or both be omitted. The optional output has already been described before.

- If Γ is split into more than two boundary parts, described by, e.g., `dirichlet`, `neumann`, and `robin`, the function **`refineBoundaryMesh`** can be used accordingly.

- **Refinement of an Element:** Suppose that element $E_i = [a,b] \in \mathcal{E}_\ell$ is bisected into two sons $e_j, e_k \in \mathcal{E}_{\ell+1}$ (or $\widehat{\mathcal{E}}_\ell$). Then, there holds $e_j = [a,m]$ and $e_k = [m,b]$, where $m = (a+b)/2$ denotes the midpoint of E_i. Note that `elements(i,:)` returns the indices of the nodes $a, b \in \mathcal{K}_\ell$. Clearly, $\mathcal{K}_\ell \subseteq \mathcal{K}_{\ell+1}$ and, e.g., the index of $a = z_p \in \mathcal{K}_{\ell+1}$ is obtained by $p =$ `elements_fine(father2son(`i`,1),1)`.
- **Boundedness of K-Mesh Constant:** Many estimates in numerical analysis depend on local quantities of the mesh, e.g., on an upper bound of the K-mesh constant

$$\kappa(\mathcal{E}_\ell) := \sup\big\{\text{length}(E_j)/\text{length}(E_k) \,:\, E_j, E_k \in \mathcal{E}_\ell \text{ with } E_j \cap E_k \neq \emptyset\big\} \geq 1 \tag{4.3}$$

which is the maximal ratio of the element widths of neighbouring elements. Let $\mathcal{E}_0$ be a given initial mesh. Let $\mathcal{E}_\ell$ be inductively obtained by refinement of arbitrary sets of marked elements $\mathcal{M}_j \subseteq \mathcal{E}_j$ with $0 \leq j \leq \ell - 1$. To avoid the blow-up of the K-mesh constant, one thus aims for a refinement rule which guarantees $\sup_{\ell\in\mathbb{N}} \kappa(\mathcal{E}_\ell) < \infty$. Our refinement rule, proposed and analyzed in [5, Section 2.2], guarantees

$$\sup_{\ell\in\mathbb{N}} \kappa(\mathcal{E}_\ell) \leq 2\,\kappa(\mathcal{E}_0) \tag{4.4}$$

by refinement of all elements in a certain superset $\overline{\mathcal{M}}_\ell \supseteq \mathcal{M}_\ell$. Moreover, one can prove that our refinement rule guarantees

$$\#\mathcal{E}_\ell - \#\mathcal{E}_0 \lesssim \sum_{j=0}^{\ell-1} \#\mathcal{M}_j, \tag{4.5}$$

i.e. the set $\overline{\mathcal{M}}_j$ is generically of the same size as $\mathcal{M}_j$, cf. [5, Theorem 2.5]. The constant hidden in the symbol $\lesssim$ only depends on the initial mesh $\mathcal{E}_0$.

Finally, we give a rough overview on the code.

- We assume that the initial mesh is uniform, i.e. $\kappa(\mathcal{T}_0) = 1$ (Line 4).
- Variable input-/output parameters are treated in Line 8–60.
- The case of adaptive refinement is treated in Line 62–131.
- In order to ensure the boundedness of the K-mesh constant, the refinement algorithm checks the mesh-size ratio of neighbouring elements: If $E_i \in \mathcal{E}_\ell$ is marked for refinement, any neighbour E_j with

$$h_\ell|_{E_j} / h_\ell|_{E_i} \geq 2$$

is recursively marked for refinement as well (Line 68–92). This guarantees $\kappa(\mathcal{E}_\ell) \leq 2\kappa(\mathcal{E}_0)$ for all generated meshes $\mathcal{E}_\ell$.
- For all refined elements the coordinates of the midpoints of these elements are computed as new nodes for the refined mesh (Line 96–98).
- We loop over each boundary part (Line 101–131), generate new elements as result of bisecting the respective coarse mesh elements (Line 109–114) and build the linkage arrays in Line 123–125.
- The case of uniform refinement (Line 133–158) is a straight forward implementation.

LISTING 4. Local Refinement of Volume Triangulation

```
1  function [vertices,new_volumes,varargout] = ...
2              refineMesh(vertices,volumes,varargin)
3
4  %*** count number of boundary parts from input
5  %*** nB will hold this number
6  %*** nB_elts will hold the number of elements of each boundary part
7  nB = 0;
8
9  for iter = 1 : (nargin - 2)
10     if size(varargin{iter},2) == 2
```

```
        nB = nB + 1;
        nB_elts(iter) = size(varargin{iter},1);
    else
        break;
    end
end

%*** check if there is at least one boundary part
if nB == 0
   error('refineMesh: There has to be at least one boundary part!');
end

%*** check the correct number of input parameters
if ~( (nargin == (nB+2)) || (nargin == (2*nB+3)) )
    error('refineMesh: Wrong number of input arguments!');
end

%*** check the correct number of output parameters
if ~( (nargout == (nB+2)) || (nargout == (2*nB+3)) )
    error('refineMesh: Wrong number of output arguments!');
end

%*** check, if user asks for father2son fields in output
if nargout == (2*nB+3)
    output_father2son = true;
else
    output_father2son = false;
end

%*** check, if user asks for adaptive refinement
if nargin == (2*nB+3)
   adaptive = true;
else
   adaptive = false;
end

%*** number of volumes
nVols = size(volumes,1);

%*** Obtain geometric information on edges
%*** Node vectors of all edges (interior edges appear twice)
I = volumes(:);
J = reshape(volumes(:,[2,3,1]),3*nVols,1);

%*** obtain set of all boundary elements of the boundary partition
bdry_elts = cat(1,varargin{1 : nB});

%*** indices of boundary parts w.r.t. entire boundary elements
%*** and w.r.t. entire edges
ptr_bdry_elts = cumsum([0,nB_elts]);
ptr_bdry_edges = ptr_bdry_elts + 3*nVols;

%*** Node vectors of all edges (all edges appear twice)
I = [I;bdry_elts(:,2)];
J = [J;bdry_elts(:,1)];

%*** Create numbering of edges
```

```
idx_IJ = find(I < J);
edge_number = zeros(length(I),1);
edge_number(idx_IJ) = 1:length(idx_IJ);
idx_JI = find(I > J);
nodes2edges = sparse(I(idx_IJ),J(idx_IJ),1:length(idx_IJ));
[foo{1:2},numbering_IJ] = find(nodes2edges);
[foo{1:2},idx_JI2IJ] = find(sparse(J(idx_JI),I(idx_JI),idx_JI));
edge_number(idx_JI2IJ) = numbering_IJ;

%*** Provide bdry_edges
for j = 1:nB
    bdry_edges{j} = edge_number(ptr_bdry_edges(j)+1:ptr_bdry_edges(j+1));
end

%*** Provide volumes2edges and edge2nodes
volumes2edges = reshape(edge_number(1:3*nVols),nVols,3);
edge2nodes = [I(idx_IJ),J(idx_IJ)];

%*** 1. determine whether uniform or adaptive mesh-refinement
%*** 2. in case of adaptive mesh-refinement compute vector
%***    of marked volumes and vector of marked boundary edges.
%***    The latter will be computed w.r.t. entire boundary edges
marked_bdry_elts = zeros(0,1);
if adaptive
   for iter = 1 : nB
       marked_bdry_elts = [marked_bdry_elts; varargin{iter + nB + 1} + ...
                           ptr_bdry_elts(iter)];
   end
   %*** one bisection per marked triangle
   marked_vols = varargin{nB + 1};
   refine_vols = 1;
else
   %*** each triangle is refined by three bisections
   marked_vols = [1:nVols]';
   refine_vols = [1,2,3];
end

%*** compute index vectors for marking of boundary edges
bdry_nodes_min = min(bdry_elts,[],2);
bdry_nodes_max = max(bdry_elts,[],2);

%*** Mark edges for refinement
edge2new_node = zeros(length(edge2nodes),1);
edge2new_node( nodes2edges( sub2ind( size(nodes2edges),...
   bdry_nodes_min(marked_bdry_elts),bdry_nodes_max(marked_bdry_elts)))) = 1;
edge2new_node(volumes2edges(marked_vols,refine_vols)) = 1;
swap = 1;
while ~isempty(swap)
    marked_edges = edge2new_node(volumes2edges);
    swap = find( ~marked_edges(:,1) & (marked_edges(:,2) | marked_edges(:,3)) );
    edge2new_node(volumes2edges(swap,1)) = 1;
end

%*** Generate new nodes
idx = find(edge2new_node);
edge2new_node(idx) = size(vertices,1) + (1:nnz(edge2new_node));
vertices(edge2new_node(idx),:) = ...
```

```
125     0.5*(vertices(edge2nodes(idx,1),:)+vertices(edge2nodes(idx,2),:));
126
127 %*** Refine boundary edges and build father2boundaries if asked for
128 for j = 1:nB
129     bdry = bdry_elts(ptr_bdry_elts(j)+1:ptr_bdry_elts(j+1),:);
130     new_nodes = edge2new_node(bdry_edges{j});
131     marked_edges = find(new_nodes);
132     nMkd_edgs = length(marked_edges);
133     non_marked_edges = find(~new_nodes);
134     nNon_mkd_edgs = length(non_marked_edges);
135     if ~isempty(marked_edges)
136         bdry = [bdry(non_marked_edges,:); ...
137                 bdry(marked_edges,1),new_nodes(marked_edges); ...
138                 new_nodes(marked_edges),bdry(marked_edges,2)];
139     end
140     varargout{j} = bdry;
141     if output_father2son
142        father2son = zeros(nB_elts(j),2);
143        father2son([non_marked_edges;marked_edges],:) = ...
144           [ repmat([1:nNon_mkd_edgs]',1,2);...
145             [(nNon_mkd_edgs+1):(nNon_mkd_edgs+nMkd_edgs)]',...
146             [(nNon_mkd_edgs+nMkd_edgs+1):(nNon_mkd_edgs+2*nMkd_edgs)]' ];
147        varargout{nB+j+1} = father2son;
148     end
149 end
150
151 %*** Provide new nodes for refinement of volumes
152 new_nodes = edge2new_node(volumes2edges);
153
154 %*** Determine type of refinement for each volume
155 marked_edges = (new_nodes ~= 0);
156 none = ~marked_edges(:,1);
157 bisec1   = ( marked_edges(:,1) & ~marked_edges(:,2) & ~marked_edges(:,3) );
158 bisec12  = ( marked_edges(:,1) &  marked_edges(:,2) & ~marked_edges(:,3) );
159 bisec13  = ( marked_edges(:,1) & ~marked_edges(:,2) &  marked_edges(:,3) );
160 bisec123 = ( marked_edges(:,1) &  marked_edges(:,2) &  marked_edges(:,3) );
161
162 %*** Generate volume numbering for refined mesh
163 idx = ones(nVols,1);
164 idx(bisec1)   = 2; %*** bisec(1): newest vertex bisection of 1st edge
165 idx(bisec12)  = 3; %*** bisec(2): newest vertex bisection of 1st and 2nd edge
166 idx(bisec13)  = 3; %*** bisec(2): newest vertex bisection of 1st and 3rd edge
167 idx(bisec123) = 4; %*** bisec(3): newest vertex bisection of all edges
168 idx = [1;1+cumsum(idx)];
169
170 %*** Generate new elements
171 new_volumes = zeros(idx(end)-1,3);
172 new_volumes(idx(none),:) = volumes(none,:);
173 new_volumes([idx(bisec1),1+idx(bisec1)],:) = ...
174    [volumes(bisec1,3),volumes(bisec1,1),new_nodes(bisec1,1); ...
175     volumes(bisec1,2),volumes(bisec1,3),new_nodes(bisec1,1)];
176 new_volumes([idx(bisec12),1+idx(bisec12),2+idx(bisec12)],:) = ...
177    [volumes(bisec12,3),volumes(bisec12,1),new_nodes(bisec12,1); ...
178     new_nodes(bisec12,1),volumes(bisec12,2),new_nodes(bisec12,2); ...
179     volumes(bisec12,3),new_nodes(bisec12,1),new_nodes(bisec12,2)];
180 new_volumes([idx(bisec13),1+idx(bisec13),2+idx(bisec13)],:) = ...
181    [new_nodes(bisec13,1),volumes(bisec13,3),new_nodes(bisec13,3); ...
```

```
    volumes(bisec13,1),new_nodes(bisec13,1),new_nodes(bisec13,3); ...
    volumes(bisec13,2),volumes(bisec13,3),new_nodes(bisec13,1)];
new_volumes([idx(bisec123),1+idx(bisec123),2+idx(bisec123),...
             3+idx(bisec123)],:) = ...
   [new_nodes(bisec123,1),volumes(bisec123,3),new_nodes(bisec123,3); ...
    volumes(bisec123,1),new_nodes(bisec123,1),new_nodes(bisec123,3); ...
    new_nodes(bisec123,1),volumes(bisec123,2),new_nodes(bisec123,2); ...
    volumes(bisec123,3),new_nodes(bisec123,1),new_nodes(bisec123,2)];

%*** build father2volumes
if output_father2son
   father2son = zeros(nVols,4);
   father2son(none,:) = repmat(idx(none),1,4);
   father2son(bisec1,:) = [repmat(idx(bisec1),1,2),repmat(idx(bisec1)+1,1,2)];
   father2son(bisec12,:) = ...
      [repmat(idx(bisec12),1,2),idx(bisec12)+1,idx(bisec12)+2];
   father2son(bisec13,:) = ...
      [idx(bisec13),idx(bisec13)+1,repmat(idx(bisec13)+2,1,2)];
   father2son(bisec123,:) = ...
      [idx(bisec123),idx(bisec123)+1,idx(bisec123)+2,idx(bisec123)+3];
   varargout{nB+1} = father2son;
end

%*** sorting vertices such that boundary nodes appear first
bdry_nodes = zeros(0,1);
for j = 1:nB
    bdry_nodes = [bdry_nodes;setdiff(unique(varargout{j}),bdry_nodes)];
end

%*** number of boundary nodes and number of vertices
nBdry_nodes = length(bdry_nodes);
nVerts = size(vertices,1);

%*** renumbering of vertices
idx = [bdry_nodes;setdiff([1:nVerts]',bdry_nodes)]; % sort nodes first
verts2new_verts(idx) = ...
   [ [1:nBdry_nodes],[(nBdry_nodes+1):nVerts] ]; % sort first

%*** reorder vertices
vertices = vertices(idx,:);

%*** assign new vertex indices to new_volumes and to all boundary parts
new_volumes=verts2new_verts(new_volumes);
for j = 1:nB
    varargout{j} = verts2new_verts(varargout{j});
end
```

4.4. Local Mesh Refinement of Volume Triangulation (Listing 4). In case of non-homogeneous volume data $f \neq 0$, `HILBERT` involves a regular triangulation $\mathcal{T}_\ell$ of Ω into non-degenerate triangles, i.e.

- $\mathcal{T}_\ell = \{T_1, \dots, T_n\}$ is a finite overlapping $\overline{\Omega} = \bigcup_{j=1}^n T_j$,
- each element $T_j \in \mathcal{T}_\ell$ is a compact triangle with positive area $|T_j| > 0$,
- the intersection $T_j \cap T_k$ for $T_j, T_k \in \mathcal{T}_\ell$ with $j \neq k$ is either empty or a vertex of both T_j and T_k or an edge of both T_j and T_k (so-called *regularity*)

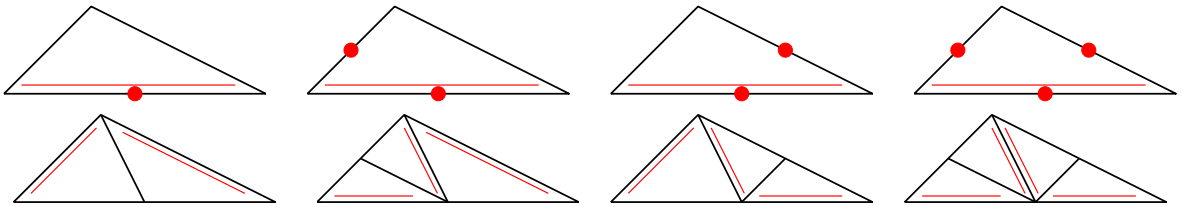

FIGURE 1. For each triangle $T \in \mathcal{T}$, there is one fixed *reference edge*, indicated by the double line (left, top). Refinement of T is done by bisecting the reference edge, where its midpoint becomes a new node. The reference edges of the son triangles are opposite to this newest vertex (left, bottom). To avoid hanging nodes, one proceeds as follows: We assume that certain edges of T, but at least the reference edge, are marked for refinement (top). Using iterated newest vertex bisection, the element is then split into 2, 3, or 4 son triangles (bottom).

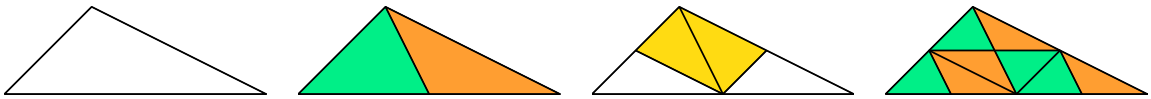

FIGURE 2. Refinement by newest vertex bisection only leads to finitely many interior angles for the family of all possible triangulations. To see this, we start from a macro element (left), where the bottom edge is the reference edge. Using iterated newest vertex bisection, one observes that only four similarity classes of triangles occur, which are indicated by the coloring. After three levels of bisection (right), no additional similarity class appears.

Moreover, for stability reasons (see Section 8 below), we only consider the case that the boundary partition $\mathcal{E}_\ell := \mathcal{T}_\ell|_\Gamma$ is the restriction of $\mathcal{T}_\ell$ to the boundary $\Gamma := \partial\Omega$. For the refinement of marked triangles (or marked boundary edges), we use newest vertex bisection, cf. Figure 1 and Figure 2. In particular, refinement of a boundary element $E_i \in \mathcal{E}_\ell$ means, that E_i is bisected into two elements e_j, e_k of half length.

Note that refinement of a boundary edge $E_i \in \mathcal{E}_\ell$ necessarily leads to a refinement of at least one triangle. Moreover, refinement of a triangle $T_j \in \mathcal{T}_\ell$ may lead to refinement of further triangles to ensure the regularity of the refined triangulation $\mathcal{T}_{\ell+1}$.

For storing the *reference edge* resp. the *newest vertex* of a triangle $T_p \in \mathcal{T}_\ell$, we make the following assumption to avoid the storage of further data: For the triangle T_p stored as

```
triangles(p,:) = [ i, j, k]
```

the vertex z_k is the newest vertex, and $\operatorname{conv}\{z_i, z_j\}$ is the reference edge.

Our implementation in Listing 4 covers the following tasks:

- uniform refinement of a given triangulation $\mathcal{T}_\ell$
- refinement of certain marked triangles and/or boundary elements, specified by the user
- linkage between elements of the input mesh with elements of the refined mesh
- handling of boundaries which are split into finitely many distinct parts, e.g. $\Gamma = \Gamma_D \cup \Gamma_N$.

Finally, we stress that newest vertex bisection only leads to finitely many similarity classes of triangles, i.e. there occur only finitely many angles in all possible triangulations, cf. Figure 2. Consequently, the resulting meshes are uniformly shape-regular

$$\sup_{\ell\in\mathbb{N}} \sigma(\mathcal{T}_\ell) < \infty \quad \text{with} \quad \sigma(\mathcal{T}_\ell) := \max_{T_j\in\mathcal{T}_\ell} \frac{\operatorname{diam}(T_j)^2}{|T_j|}. \tag{4.6}$$

By elementary geometry, this implies that all edges of a triangle $T_j \in \mathcal{T}_\ell$ have, up to the uniformly bounded constant $\sigma(\mathcal{T}_\ell)$, the same length. In particular, this implies that the K-mesh

constant of $\mathcal{E}_\ell = \mathcal{T}_\ell|_\Gamma$ is uniformly bounded

$$\sup_{\ell\in\mathbb{N}} \kappa(\mathcal{E}_\ell) < \infty \tag{4.7}$$

as well. We now discuss certain aspects of our implementation from Listing 4, where the data structure of `vertices`, `triangles`, and `elements` (as well as `dirichlet` and `neumann`) is discussed in Section 3.3 above. Note that the nodes on the boundary are contained in `vertices` so that the boundary partition `elements` now links to `vertices` instead of `coordinates`.

- **Input/Output Parameters:** To allow a partition of Γ into finitely many parts (e.g., a Dirichlet and a Neumann boundary), the formal signature reads

```
[vertices_fine,triangles_fine,varargout] ...
  = refineMesh(vertices,triangles,varargin)
```

  where the refined mesh is stored by the arrays `vertices_fine` and `triangles_fine`. To explain the variable input/output parameters, we consider certain examples.
- Suppose that $\mathcal{T}_\ell$ is a triangulation described by `vertices` and `triangles` and that $\mathcal{E}_\ell$ is a boundary mesh described by `elements`. Then,

```
[vertices_fine,triangles_fine,elements_fine,father2triangles,father2son] ...
  = refineMesh(vertices,triangles,elements)
```

  provides the uniformly refined mesh where each boundary element $E \in \mathcal{E}_\ell$ is split and where each triangle $T \in \mathcal{T}_\ell$ is refined by *three* bisections, see Figure 1.

  The $(N \times 2)$-matrix `father2son` provides a link between the initial boundary mesh and the refined boundary mesh and is discussed in great detail in Section 4.3. The $(n\times)4$-matrix `father2triangles` that provides a link between the initial mesh and the refined mesh. For $T_i \in \mathcal{T}_\ell$, `father2triangles(i,:)` contains the indices of its four sons $t_j \in \widehat{\mathcal{T}}_\ell$.
- Second, suppose that Γ is split into a Dirichlet boundary Γ_D and a Neumann boundary Γ_N. In this case the boundary mesh is described in terms of `dirichlet` and `neumann`. Then,

```
[vertices_fine,triangles_fine,dirichlet_fine,neumann_fine, ...
 father2triangles,father2dirichlet,father2neumann] ...
  = refineMesh(vertices,triangles,dirichlet,neumann)
```

  provides the uniformly refined mesh, i.e. each boundary element is halved and each triangle is refined by three bisections. We stress that the function may deal with a partition of Γ into finitely many boundary conditions.
- Third, suppose that $\mathcal{M}_\ell \subseteq \mathcal{T}_\ell \cup \mathcal{E}_\ell$ is a set of elements and triangles which are marked for refinement. Let `marked_triangles` and `marked_elements` be column vectors containing the indices of the boundary elements and the triangles which have to be refined. Then,

```
[vertices_fine,triangles_fine,elements_fine,father2triangles,father2son] ...
  = refineMesh(vertices,triangles,elements,marked_triangles,marked_elements)
```

  returns a refined meshes $\mathcal{T}_{\ell+1}$ and $\mathcal{E}_{\ell+1} = \mathcal{T}_{\ell+1}|_\Gamma$ such that all marked elements $E_i \in \mathcal{M}_\ell \cap \mathcal{E}_\ell$ have been halved and all marked triangles $T_j \in \mathcal{M}_\ell \cap \mathcal{T}_\ell$ have been refined by (at least) *one* bisection. Recall that, by assumption, the first edge of T_i is the reference edge. Moreover, if T_i is refined, at least the reference edge is refined by definition of newest vertex bisection.

  In this case, $(n \times 4)$-matrix `father2triangles` contains the following entries for $T_i \in \mathcal{T}_\ell$:
  - If T_i is not refined, all entries of `father2triangles(i,:)` coincide and give the index of $T_i = t_j \in \mathcal{T}_{\ell+1}$.
  - If only the reference edge (first edge) of T_i is refined, `father2triangles(i,:)` contains the indices of its two sons, each of which appears twice, and the first and second entry of the array are the same, as well as the third and forth entry.
  - If the first and the second edge of T_i is refined, `father2triangles(i,:)` contains the indices of its three sons and the first two entries of the vector are equal.

  - If the first and the third edge of T_i is refined, `father2triangles(i,:)` contains the indices of its three sons and the last two entries of the vector are equal.
  - If T_i has been refined by three bisections, `father2triangles(i,:)` contains the indices of its four sons.
- Finally, we stress that local refinement also works if Γ is split into finitely many parts, e.g. a Dirichlet boundary Γ_D and and Neumann boundary Γ_N. Here, the according function call reads

```
[vertices_fine,triangles_fine,dirichlet_fine,neumann_fine, ...
 father2triangles,father2dirichlet,father2neumann] ...
  = refineMesh(vertices,triangles,dirichlet,neumann, ...
               marked_triangles,marked_dirichlet,marked_neumann)
```

  As before, all marked boundary elements are halved and all marked triangles are refined by (at least) one bisection.
- We conclude the list of examples with some remarks on restrictions concerning the usage of this function. First, one may omit all `marked` vectors or none of them. For example, it is not allowed to provide a vector for the adaptive refinement of the triangulation but none for the boundary meshes. However, these vectors may be empty. Secondly, one may omit all `father2*` vectors or none of them.

Finally, we give a rough overview on the code:

- In Line 9–45, we process the input arguments and determine if we use adaptive or uniform mesh refinement and if the user wants to retrieve a link between the initial mesh and the refined mesh using the `father2*` vectors.
- In Line 52–84, we generate a numbering of all edges and, in particular, a numbering of the boundary edges.
- In Line 92–119, we translate all marked boundary edges and all marked triangles into marked edges of triangles. The closure step in Line 116–118, guarantees that at least the reference edge of a triangle T_j is marked, if some edge of T_j is marked. This is part of the newest vertex bisection algorithm and guarantees that the resulting mesh $\mathcal{T}_{\ell+1}$ is regular.
- In Line 122–125, we generate the new vertices which are precisely the midpoints of the marked edges.
- In Line 129–149, we refine the marked boundary edges. We also compute the `father2son` vectors if we are asked for.
- In Line 152–189, we refine the triangles. In Line 157–160, we determine the different types of refinement. In, Line 163–168, we prepare the new numbering of the triangulation $\mathcal{T}_{\ell+1}$. We ensure that refined triangles appear right after each other in the `new_volumes` array. Then, we actually refine the triangles (Line 171–189).
- In Line 192–203, we compute the link between the input triangulation $\mathcal{T}_\ell$ and the refined mesh $\mathcal{T}_{\ell+1}$, if the user requests the array `father2son`.
- In Line 206–227, we finally reorder the numbering of the elements and ensure that the vertices on the boundary appear first in the array `vertices`.

5. Symm's Integral Equation

Continuous Model Problem. In the entire section, we consider Symm's integral equation

$$V\phi = (K + 1/2)g \quad \text{on } \Gamma \tag{5.1}$$

with V the simple-layer potential and K the double-layer potential, where $\Gamma = \partial\Omega$ is the piecewise-affine boundary of a polygonal Lipschitz domain $\Omega \subset \mathbb{R}^2$. This integral equation is an equivalent formulation of the Dirichlet problem

$$-\Delta u = 0 \text{ in } \Omega \quad \text{with} \quad u = g \text{ on } \Gamma. \tag{5.2}$$

Formally, the Dirichlet data satisfy $g \in H^{1/2}(\Gamma)$. We will, however, assume additional regularity $g \in H^1(\Gamma) \subset H^{1/2}(\Gamma)$ so that g is, in particular, continuous. The exact solution $\phi \in H^{-1/2}(\Gamma)$ of (5.1) is the normal derivative $\phi = \partial_n u$ of the solution $u \in H^1(\Omega)$ of (5.2).

Note that (5.1) can equivalently be written in variational form

$$\langle V\phi\,,\,\psi\rangle_\Gamma = \langle (K+1/2)g\,,\,\psi\rangle_\Gamma \quad \text{for all } \psi \in H^{-1/2}(\Gamma), \tag{5.3}$$

where $\langle\cdot\,,\,\cdot\rangle_\Gamma$ denotes the extended $L^2(\Gamma)$-scalar product, i.e. $\langle\phi\,,\,\psi\rangle_\Gamma = \int_\Gamma \phi\psi\,d\Gamma$ for $\phi,\psi \in L^2(\Gamma)$ and with $\int_\Gamma d\Gamma$ integration along the boundary. Provided that $\operatorname{diam}(\Omega) < 1$, one can show that the left-hand side

$$\langle\!\langle \phi\,,\psi\rangle\!\rangle_V := \langle V\phi\,,\,\psi\rangle_\Gamma \quad \text{for } \phi,\psi \in H^{-1/2}(\Gamma) \tag{5.4}$$

of (5.3) defines a scalar product on $H^{-1/2}(\Gamma)$, and the induced norm $|\!|\!|\phi|\!|\!|_V := \langle\!\langle \phi\,,\phi\rangle\!\rangle_V^{1/2}$ is an equivalent norm on $H^{-1/2}(\Gamma)$. In particular, the variational form (5.3) has a unique solution $\phi \in H^{-1/2}(\Gamma)$ which depends continuously on the data g with respect to the $H^{1/2}(\Gamma)$-norm.

Galerkin Discretization. Let $\{\zeta_1,\dots,\zeta_N\}$ denote the set of canonical basis functions of $\mathcal{S}^1(\mathcal{E}_\ell)$. To discretize (5.3), we first replace the continuous Dirichlet data $g \in H^1(\Gamma)$ either by its nodal interpolant

$$G_\ell := \sum_{j=1}^N g(z_j)\zeta_j \in \mathcal{S}^1(\mathcal{E}_\ell) \subset H^1(\Gamma) \tag{5.5}$$

or by its L^2-projection onto $\mathcal{S}^1(\mathcal{E}_\ell)$, i.e. $G_\ell \in \mathcal{S}^1(\mathcal{E}_\ell)$ is the (unique) solution of

$$\int_\Gamma G_\ell\zeta_k\,d\Gamma = \int_\Gamma g\,\zeta_k\,d\Gamma \quad \text{for all } k = 1,\dots,N. \tag{5.6}$$

Second, we replace the entire function space $H^{-1/2}(\Gamma)$ in (5.3) by the finite-dimensional space $\mathcal{P}^0(\mathcal{E}_\ell)$. Since the discrete space $\mathcal{P}^0(\mathcal{E}_\ell)$ is a subspace of $H^{-1/2}(\Gamma)$, $\langle\!\langle\cdot\,,\cdot\rangle\!\rangle_V$ from (5.4) is also a scalar product on $\mathcal{P}^0(\mathcal{E}_\ell)$. Consequently, there is a unique Galerkin solution $\Phi_\ell \in \mathcal{P}^0(\mathcal{E}_\ell)$ of

$$\langle V\Phi_\ell\,,\,\Psi_\ell\rangle_\Gamma = \langle (K+1/2)G_\ell\,,\,\Psi_\ell\rangle_\Gamma \quad \text{for all } \Psi_\ell \in \mathcal{P}^0(\mathcal{E}_\ell). \tag{5.7}$$

According to Linear Algebra, (5.7) holds for all $\Psi_\ell \in \mathcal{P}^0(\mathcal{E}_\ell)$ if and only if it holds for all (canonical) basis functions $\chi_k \in \mathcal{B}_\ell = \{\chi_1,\dots,\chi_N\}$ of $\mathcal{P}^0(\mathcal{E}_\ell)$. With the coefficient vector $\mathbf{x} \in \mathbb{R}^N$ of the ansatz

$$\Phi_\ell = \sum_{j=1}^N \mathbf{x}_j\chi_j \tag{5.8}$$

and the vector $\mathbf{g} \in \mathbb{R}^N$ from

$$G_\ell = \sum_{j=1}^N \mathbf{g}_j\zeta_j, \tag{5.9}$$

the Galerkin formulation (5.7) is thus equivalent to

$$\sum_{j=1}^N \mathbf{x}_j\langle V\chi_j\,,\,\chi_k\rangle_\Gamma = \langle V\Phi_\ell\,,\,\chi_k\rangle_\Gamma = \langle (K+1/2)G_\ell\,,\,\chi_k\rangle_\Gamma = \sum_{j=1}^N \mathbf{g}_j\langle (K+1/2)\zeta_j\,,\,\chi_k\rangle_\Gamma$$

for all $k = 1,\dots,N$. If we define matrices $\mathbf{V},\mathbf{K},\mathbf{M} \in \mathbb{R}^{N\times N}$ by

$$\mathbf{V}_{kj} = \langle V\chi_j\,,\,\chi_k\rangle_\Gamma, \quad \mathbf{K}_{kj} = \langle K\zeta_j\,,\,\chi_k\rangle_\Gamma, \quad \mathbf{M}_{kj} = \langle \zeta_j\,,\,\chi_k\rangle_\Gamma \quad \text{for all } j,k = 1,\dots,N, \tag{5.10}$$

the last equation becomes

$$(\mathbf{V}\mathbf{x})_k = \sum_{j=1}^N \mathbf{x}_j\mathbf{V}_{kj} = \sum_{j=1}^N \mathbf{g}_j\Big(\mathbf{K}_{kj} + \frac{1}{2}\,\mathbf{M}_{kj}\Big) = \Big(\mathbf{K}\mathbf{g} + \frac{1}{2}\,\mathbf{M}\mathbf{g}\Big)_k \quad \text{for all } k = 1,\dots,N.$$

Altogether, the Galerkin formulation (5.7) is thus equivalent to the linear system

$$\mathbf{V}\mathbf{x} = \mathbf{K}\mathbf{g} + \frac{1}{2}\mathbf{M}\mathbf{g}. \tag{5.11}$$

We stress that $\mathbf{V}$ is symmetric and positive definite since it stems from a scalar product. In particular, the linear system (5.11) has a unique solution $\mathbf{x} \in \mathbb{R}^N$.

5.1. Discretization of Dirichlet Data and Computation of Corresponding Dirichlet Data Oscillations. Instead of solving the correct variational form (5.3), we solve

$$\langle V\phi_\ell \,,\, \psi\rangle_\Gamma = \langle (K+1/2)G_\ell \,,\, \psi\rangle_\Gamma \quad \text{for all } \psi \in H^{-1/2}(\Gamma) \tag{5.12}$$

with perturbed right-hand side, where we use the approximation $G_\ell \approx g$. For nodal interpolation (5.5), it is an analytical observation that the error between the exact solution $\phi \in H^{-1/2}(\Gamma)$ of (5.3) and the exact solution $\phi_\ell \in H^{-1/2}(\Gamma)$ of the perturbed formulation (5.12) is controlled by

$$|||\phi - \phi_\ell|||_V \lesssim \|h_\ell^{1/2}(g - G_\ell)'\|_{L^2(\Gamma)} =: \mathrm{osc}_{D,\ell}, \tag{5.13}$$

where $(\cdot)'$ denotes the arclength-derivative, cf. [5]. Since the proof of [5] only requires that $g - G_\ell$ has certain zeros (on each elements resp. on each patch), the same estimate also holds in case of the L^2-projection of (5.6).

Listing 5. Computation of Data Oscillations for nodal interpolation of Dirichlet Data

```
function [osc,uDh] = computeOscDirichlet(coordinates,elements,uD)
%*** compute midpoints of all elements
midpoints = 0.5*( coordinates(elements(:,1),:) + coordinates(elements(:,2),:) );

%*** evaluate Dirichlet data at element midpoints
uD_midpoints = uD(midpoints);

%*** evaluate Dirichlet data at all Dirichlet nodes
uDh = zeros(size(coordinates(:,1),1),1);
idx = unique(elements);
uDh(idx) = uD(coordinates(idx,:));

%*** compute oscillations of Dirichlet data via adapted Newton–Cotes formula
osc = 4/3*( uDh(elements(:,1))+uDh(elements(:,2))-2*uD_midpoints ).^2;
```

5.1.1. Nodal interpolation of Dirichlet data (Listing 5). In this subsection, we aim for a numerical approximation of the local contributions

$$\mathrm{osc}_{D,\ell}(E_j) = \|h_\ell^{1/2}(g - G_\ell)'\|_{L^2(E_j)} = \mathrm{length}(E_j)^{1/2}\|(g - G_\ell)'\|_{L^2(E_j)} \quad \text{for all } E_j \in \mathcal{E}_\ell$$

of the oscillations defined in (5.13) in case of nodal interpolation. For $E_j = [a_j, b_j] \in \mathcal{E}_\ell$ and $h := \mathrm{length}(E_j) = |b_j - a_j|$, let $\gamma_j : [-1,1] \to E_j$ denote the reference parametrization from (2.1). Recall that $|\gamma_j'| = h/2$. With the definition of a boundary integral from Section 2.2 and the definition of the arclength derivative from Section 2.3, we obtain

$$\|v'\|_{L^2(E_j)}^2 = \int_{E_j} (v')^2 \, d\Gamma \overset{\text{Def}}{=} \frac{h}{2}\int_{-1}^1 \big((v'\circ\gamma_j)(s)\big)^2 \, ds \overset{\text{Def}}{=} \frac{2}{h}\int_{-1}^1 \big((v\circ\gamma_j)'(s)\big)^2 \, ds. \tag{5.14}$$

We now approximate $w := v \circ \gamma_j : [-1,1] \to \mathbb{R}$ by some polynomial $p_j \in \mathcal{P}^2[-1,1]$ with

$$p_j(-1) = w(-1) = v(a_j), \quad p_j(0) = w(0) = v(m_j), \quad p_j(1) = w(1) = v(b_j),$$

where $m_j = (a_j + b_j)/2$ denotes the midpoint of E_j. Note that $p_j' \in \mathcal{P}^1[-1,1]$ and $(p_j')^2 \in \mathcal{P}^2[-1,1]$ so that

$$\|v'\|_{L^2(E_j)}^2 = \frac{2}{h}\int_{-1}^1 \big((v\circ\gamma_j)'(s)\big)^2\,ds \approx \frac{2}{h}\int_{-1}^1 (p_j')^2\,ds = \frac{2}{h}\,\mathrm{quad}_2\big((p_j')^2\big),$$

where $\mathrm{quad}_2(\cdot)$ is a quadrature rule on $[-1,1]$ which is exact on $\mathcal{P}^2[-1,1]$. We use a 3-point Newton-Côtes formula with nodes $s_1 = -1$, $s_2 = 0$, and $s_3 = 1$, which is exact on $\mathcal{P}^3[-1,1]$. It thus remains to evaluate $p_j'(s_k)$ by use of $p_j(-1)$, $p_j(0)$, and $p_j(1)$. To that end, we write p_j in terms of the Lagrangian basis

$$p_j = v(a_j)L_1 + v(m_j)L_2 + v(b_j)L_3, \quad \text{whence} \quad p_j' = v(a_j)L_1' + v(m_j)L_2' + v(b_j)L_3'.$$

The Lagrange polynomials L_k associated with $s_k = -1, 0, 1$ read

$$L_1(s) = s(s-1)/2, \quad L_2(s) = 1 - s^2, \quad L_3(s) = s(s+1)/2,$$

and their derivatives are

$$L_1'(s) = (2s-1)/2, \quad L_2'(s) = -2s, \quad L_3'(s) = (2s+1)/2.$$

With the matrix $(L_\ell'(s_k))_{k,\ell=1}^3$, $p_j'(s_k)$ is thus obtained from a matrix-vector multiplication

$$\begin{pmatrix} p_j'(-1) \\ p_j'(0) \\ p_j'(+1) \end{pmatrix} = \begin{pmatrix} L_1'(-1) & L_2'(-1) & L_3'(-1) \\ L_1'(0) & L_2'(0) & L_3'(0) \\ L_1'(+1) & L_2'(+1) & L_3'(+1) \end{pmatrix} \begin{pmatrix} v(a_j) \\ v(m_j) \\ v(b_j) \end{pmatrix} = \begin{pmatrix} -3/2 & +2 & -1/2 \\ -1/2 & 0 & +1/2 \\ +1/2 & -2 & +3/2 \end{pmatrix} \begin{pmatrix} v(a_j) \\ v(m_j) \\ v(b_j) \end{pmatrix}.$$

For the computation of the local Dirichlet data oscillations

$$\mathrm{osc}_{D,\ell}(E_j)^2 = h\,\|(g - G_\ell)'\|_{L^2(E_j)}^2 = 2\int_{-1}^1 \big((g-G_\ell)\circ\gamma_j)'(s)\big)^2\,ds,$$

we have $v = g - G_\ell$. This results in $(g - G_\ell)(a_j) = 0 = (g - G_\ell)(b_j)$ by definition of the nodal interpolant G_ℓ. Consequently, everything simplifies to

$$\begin{pmatrix} p_j'(-1) \\ p_j'(0) \\ p_j'(+1) \end{pmatrix} = \begin{pmatrix} 2\,v(m_j) \\ 0 \\ -2\,v(m_j) \end{pmatrix} = \Big(g(m_j) - \frac{g(a_j)+g(b_j)}{2}\Big) \begin{pmatrix} +2 \\ 0 \\ -2 \end{pmatrix} = \big(g(a_j) + g(b_j) - 2\,g(m_j)\big) \begin{pmatrix} -1 \\ 0 \\ +1 \end{pmatrix}.$$

Note that the weights of the Newton-Côtes formula read

$$\omega_k = \int_{-1}^1 L_k(t)\,dt, \quad \text{whence} \quad \omega_1 = 1/3, \quad \omega_2 = 4/3, \quad \omega_3 = 1/3.$$

Therefore,

$$\begin{aligned} \mathrm{osc}_{D,\ell}(E_j)^2 \approx \widetilde{\mathrm{osc}}_{D,\ell}(E_j)^2 := 2\,\mathrm{quad}_2\big((p_j')^2\big) &= 2\sum_{k=1}^3 \omega_k \big(p_j'(s_k)\big)^2 \\ &= \frac{4}{3}\big(g(a_j) + g(b_j) - 2\,g(m_j)\big)^2. \end{aligned} \tag{5.15}$$

Altogether, the documentation of Listing 5 now reads as follows:

- The function takes the mesh $\mathcal{E}_\ell$ in terms of `coordinates` and `elements` as well as a function handle `uD` for the Dirichlet data g. Besides the local data oscillations, it returns the column vector `g` = `uDh` of the nodal values of the Dirichlet data g (Line 1).
- We first compute all element midpoints (Line 3) and evaluate the Dirichlet data g at all midpoints (Line 6) and all nodes (Line 9–11). In case that `elements` describes a partition of the entire boundary, the latter could simply be performed by

  ```
  uDh = uD(coordinates);
  ```

  However, we aim to reuse this code in case of mixed boundary value problems, where `elements` provides only a partition of the Dirichlet boundary. Our implementation avoids to evaluate the Dirichlet data u_D in Neumann nodes.
- Finally, Formula (5.15) is realized (Line 14) simultaneously for all elements $E_j \in \mathcal{E}_\ell$.

- The function returns the column vector of elementwise Dirichlet data oscillations

$$\mathbf{v} := \big(\widetilde{\operatorname{osc}}_{D,\ell}(E_1)^2, \dots, \widetilde{\operatorname{osc}}_{D,\ell}(E_N)^2\big) \in \mathbb{R}^N$$

  so that $\operatorname{osc}_{D,\ell} \approx \big(\sum_{j=1}^N \mathbf{v}_j\big)^{1/2}$.

Remark 5.1. *For smooth Dirichlet data g and uniform meshes with mesh-size h, there holds*

$$\operatorname{osc}_{D,\ell} = \mathcal{O}(h^{3/2}) \quad \text{and} \quad \Big|\operatorname{osc}_{D,\ell} - \Big(\sum_{j=1}^N \mathbf{v}_j\Big)^{1/2}\Big| = \mathcal{O}(h^{5/2}).$$

Therefore, the quadrature error is of higher order when compared to the discretization order.

LISTING 6. Computation of Data Oscillations for L^2-projection of Dirichlet Data

```
function [osc,uDh] = computeOscDirichletL2(coordinates,elements,uD,varargin)
%*** gauss(3) quadrature
quad_nodes = sqrt(3/5)*[-1;0;1];
quad_weights = [5,8,5]/9;

%*** general constants
nC = size(coordinates,1);
nE = size(elements,1);
nQ = size(quad_nodes,1);

%*** compute mesh-size
a = coordinates(elements(:,1),:);
b = coordinates(elements(:,2),:);
h = sqrt(sum((a-b).^2,2));

%*** build L2 mass matrix with respect to S1
I = elements(:,[1 1 2 2]);
J = elements(:,[1 2 1 2]);
M = sparse(I,J,h*[2 1 1 2]/6);

%*** perform all necessary evaluations of Dirichlet data uD
sx = reshape(a,2*nE,1)*(1-quad_nodes') + reshape(b,2*nE,1)*(1+quad_nodes');
sx = 0.5 * reshape(sx',nQ*nE,2);
uD_sx = reshape(uD(sx,varargin{:}),nQ,nE);

%*** build right-hand side vector for L2 projection onto S1
c = 0.25 * [h;h] .* [ quad_weights*(uD_sx.*repmat(1-quad_nodes,1,nE)), ...
                      quad_weights*(uD_sx.*repmat(1+quad_nodes,1,nE)) ]';
c = accumarray(elements(:),c);

%*** compute (nodal vector of) L2 projection uDh of uD onto S1
uDh = M\c;

%*** evaluate uDh at quadrature nodes
uDh_sx = 0.5 * ( (1-quad_nodes)*uDh(elements(:,1))' ...
                + (1+quad_nodes)*uDh(elements(:,2))' );

%*** compute oscillations of Dirichlet data via gauss(3) formula
D = 0.5 * sqrt(5/3) * [-3 -4 -1 ; -1 0 1 ; 1 4 3];
osc = 2*(quad_weights * ( D*(uD_sx - uDh_sx) ).^2)';
```

5.1.2. L^2-projection of Dirichlet data (Listing 6). With the ansatz $G_\ell = \sum_{j=1}^N \mathbf{g}_j \zeta_j \in \mathcal{S}^1(\mathcal{E}_\ell)$ and an unknown coefficient vector $\mathbf{g} \in \mathbb{R}^N$, Equation (5.6) is equivalent to the matrix equation

$$\widetilde{\mathbf{M}}\mathbf{g} = \mathbf{c}, \quad \text{where} \quad \widetilde{\mathbf{M}}_{jk} = \int_\Gamma \zeta_k \zeta_j \, d\Gamma \text{ and } \mathbf{c}_k = \int_\Gamma g \, \zeta_k \, d\Gamma. \tag{5.16}$$

Clearly, $\widetilde{\mathbf{M}}$ is positive definite and symmetric. This implies that $\mathbf{g} \in \mathbb{R}^N$ is uniquely existing, and hence the function $G_\ell \in \mathcal{S}^1(\mathcal{E}_\ell)$ in (5.6) is uniquely defined.

To compute G_ℓ in practice, we have to assemble the matrix $\widetilde{\mathbf{M}} \in \mathbb{R}^{N\times N}_{sym}$ as well as the right-hand side vector $\mathbf{c} \in \mathbb{R}^N$. The assembly of the mass matrix $\widetilde{\mathbf{M}}$ could be done as follows:

```
nC = size(coordinates,1);
nE = size(elements,1);
M = sparse(nC,nC);
for j = 1:nE
    nodes = elements(j,:);
    h = norm(coordinates(nodes(2),:)-coordinates(nodes(1),:));
    M(nodes,nodes) = M(nodes,nodes) + [2 1;1 2]/6*h;
end
```

The entries $\mathbf{c}_k$ of the right-hand side vector $\mathbf{c} \in \mathbb{R}^N$ are computed by numerical quadrature. We use the 3-point Gaussian quadrature of order 5 on $[-1,1]$: For an element $E_j = [a_j, b_j]$ with $h = \mathrm{length}(E_j)$ and corresponding reference parametrization $\gamma_j : E \to E_j$, it holds

$$\int_{E_j} u\zeta_k \, d\Gamma = \frac{h}{2}\int_{-1}^1 g\circ\gamma_j(s)\ \zeta_k\circ\gamma_j(s)\, ds \approx \frac{h}{4}\begin{cases} 0 & \text{if } z_k \notin \{a_j, b_j\}, \\ \mathtt{gauss}_3\big((1-s)g\circ\gamma_j\big) & \text{for } z_k = a_j, \\ \mathtt{gauss}_3\big((1+s)g\circ\gamma_j\big) & \text{for } z_k = b_j, \end{cases}$$

since

$$\zeta_k\circ\gamma_j(s) = \begin{cases} 0 & \text{if } z_k \notin \{a_j, b_j\}, \\ \frac{1-s}{2} & \text{for } z_k = a_j, \\ \frac{1+s}{2} & \text{for } z_k = b_j. \end{cases}$$

For the numerical approximation of the local contributions

$$\mathrm{osc}_{D,\ell}(E_j) = \|h_\ell^{1/2}(g - G_\ell)'\|_{L^2(E_j)} = \mathrm{length}(E_j)^{1/2}\|(g-G_\ell)'\|_{L^2(E_j)} \quad \text{for all } E_j \in \mathcal{E}_\ell$$

of the oscillations defined in (5.13), we proceed as for the nodal interpolation in Section 5.1.1. However, to reuse the function evaluations of u, we use the 3-point Gaussian quadrature instead of the 3-point Newton-Côtes formula: The nodes of the 3-point Gauss quadrature read

$$s_1 = -\sqrt{3/5}, \quad s_2 = 0, \quad s_3 = \sqrt{3/5},$$

with corresponding weights

$$\omega_1 = 5/9 = \omega_3 \quad \text{and} \quad \omega_2 = 8/9.$$

Direct calculation for the corresponding Lagrange polynomials shows

$$\begin{aligned} L_1(s) &= \frac{5}{6}\, s(s-\sqrt{3/5}), & L_1'(s) &= \frac{5}{6}(2s - \sqrt{3/5}), \\ L_2(s) &= \frac{5}{3}\, s^2 - 1, & L_2'(s) &= \frac{10}{3}\, s, \\ L_3(s) &= \frac{5}{6}\, s(s+\sqrt{3/5}), & L_3'(s) &= \frac{5}{6}\,(2s + \sqrt{3/5}). \end{aligned}$$

We approximate $v := (g - G_\ell) \circ \gamma_j$ by the polynomial $p_j \in \mathcal{P}^2[-1,1]$ with $p_j(s_k) = v \circ \gamma_j(s_k)$. To evaluate $p_j'(s_k)$, we use the representation $p_j'(s) = \sum_{\ell=1}^3 p_j(s_\ell) L_\ell'(s)$. This leads to the matrix

$$\big(L_\ell'(s_k)\big)_{k,\ell=1}^3 \begin{pmatrix} -5/2\,\sqrt{3/5} & -10/3\,\sqrt{3/5} & -5/6\,\sqrt{3/5} \\ -5/6\,\sqrt{3/5} & 0 & 5/6\,\sqrt{3/5} \\ 5/6\,\sqrt{3/5} & 10/3\,\sqrt{3/5} & 5/2\,\sqrt{3/5} \end{pmatrix} = \frac{1}{2}\sqrt{5/3} \begin{pmatrix} -3 & -4 & -1 \\ -1 & 0 & 1 \\ 1 & 4 & 3 \end{pmatrix}$$

to compute $\big(p_j'(s_1), p_j'(s_2), p_j'(s_3)\big)$ from $\big(p_j(s_1), p_j(s_2), p_j(s_3)\big)$.

Arguing as in the previous section, we see

$$\operatorname{length}(E_j)\|v'\|_{L^2(E_j)}^2 = 2\int_{-1}^1 \big(v \circ \gamma_j)'(s)\big)^2\,ds \approx 2\int_{-1}^1 \big(p_j'(s)\big)^2\,ds = 2\,\operatorname{gauss}_2\big((p_j')^2\big).$$

Our implementation then returns

$$\widetilde{\operatorname{osc}}_{D,\ell}(E_j)^2 := 2\,\operatorname{gauss}_2\big((p_j')^2\big) \approx \operatorname{osc}_{D,\ell}(E_j)^2 \quad \text{for all } j = 1, \dots, N. \tag{5.17}$$

Altogether, the documentation of Listing 6 now reads as follows:

- The function takes the mesh in terms of `coordinates` and `elements` as well as a function handle `uD` for the Dirichlet data g. Besides the local data oscillations, it returns the column vector `g` = `uDh` of the nodal values of the L^2-projection G_ℓ of the Dirichlet data g (Line 1).
- All occuring integrals are transformed to $[-1,1]$ and computed by the 3-point Gauss-Legendre quadrature on $[-1,1]$ which is exact for polynomials of degree 5 (Line 3–4).
- The local mesh-width is computed (Line 12–14), i.e. `h` is a column vector with `h`(j) = $\operatorname{length}(E_j)$.
- The mass matrix $\widetilde{\mathbf{M}}$ from (5.16) is built (Line 17–19).
- To avoid functions calls of `uD`, all necessary evaluations of `uD` are done simultaneously and stored in an `nQ` $\times$ `nE` array `uD_sx`, i.e. the j-th column `uD_s`$(:,j)$ contains the evaluations of `uD` in the quadrature nodes related to E_j (Line 22–24).
- The right-hand side vector $\mathbf{c}$ from (5.16) is build (Line 27–29) by use of elementwise quadrature, and the nodal values of G_ℓ are computed and stored in `uDh` (Line 32).
- For $s \in [-1,1]$, it holds that $G_\ell \circ \gamma_j(s) = \frac{1}{2}\big((1-s)G_\ell(a_j) + (1+s)G_\ell(b_j)\big)$, since G_ℓ is affine on $E_j = [a_j, b_j]$. Therefore, the `nQ` $\times$ `nE` array `uDh_sx` contains the evaluation of G_ℓ in all quadrature nodes analogously to the array `uD_sx` (Line 35).
- Finally, formula (5.17) is realized for all elements $E_j \in \mathcal{E}_\ell$, and the function returns a column vector

  $$\mathbf{v} = \big(\widetilde{\operatorname{osc}}_{D,\ell}(E_1)^2, \dots, \widetilde{\operatorname{osc}}_{D,\ell}(E_N)^2\big)$$

  of approximate elementwise Dirichlet data oscillations.

Remark 5.2. *For smooth Dirichlet data g and uniform meshes with mesh-size h, there holds*

$$\operatorname{osc}_{D,\ell} = \mathcal{O}(h^{3/2}) \quad \textit{and} \quad \Big|\operatorname{osc}_{D,\ell} - \Big(\sum_{j=1}^N \mathbf{v}_j\Big)^{1/2}\Big| = \mathcal{O}(h^{5/2}).$$

Therefore, the quadrature error is of higher order when compared to the discretization order.

5.2. Computation of Discrete Integral Operators V and K. The matrices $\mathbf{V}, \mathbf{K} \in \mathbb{R}^{N\times N}$ defined in (5.10) are implemented in the programming language C via the MATLAB-MEX-Interface. The simple-layer potential matrix $\mathbf{V}$ is returned by call of

```
V = buildV(coordinates,elements [,eta]);
```

In general, all matrix entries of $\mathbf{V}$ can be computed analytically by use of anti-derivatives found in [22]. However, analytic integration leads to cancellation effects if the integration domain is small, i.e. $\int_a^b \cdot\, dx$ with $a \approx b$. In this case, the (continuous) integrand is generically of one sign so that Gaussian quadrature (with positive weights) appears to be more stable.

Let $\eta \geq 0$ be given. Recall that

$$\mathbf{V}_{kj} = -\frac{1}{2\pi} \int_{E_k} \int_{E_j} \log|x-y| \, d\Gamma(y) \, d\Gamma(x).$$

A pair of elements (E_j, E_k) is called *admissible* provided that

$$\min\{\text{length}(E_j), \text{length}(E_k)\} \leq \eta \, \text{dist}(E_j, E_k) \tag{5.18}$$

with $\text{dist}(\cdot,\cdot)$ the distance of E_j and E_k. Otherwise, the pair (E_j, E_k) is called inadmissible. Note that for $\mathbf{V}_{kj}$, the Fubini theorem applies and proves that one can assume w.l.o.g. that $\text{length}(E_k) \leq \text{length}(E_j)$. Note that the cancellation effects from the outer integration are thus generically higher than those of the inner integration. For fixed $x \in E_k$, the inner integral

$$\int_{E_j} \log|x-y| \, d\Gamma(y)$$

is computed analytically [22]. If the pair (E_j, E_k) is admissible, we parametrize E_k and approximate

$$\begin{aligned}\int_{E_k} \int_{E_j} \log|x-y| \, d\Gamma(y) \, d\Gamma(x) &= \int_{-1}^{1} \int_{E_j} \log|\gamma_k(s)-y| \, d\Gamma(y) \, ds \\ &\approx \sum_{m=1}^{p} \omega_m \int_{E_j} \log|\gamma_k(s_m)-y| \, d\Gamma(y)\end{aligned}$$

with a Gaussian quadrature on $[-1,1]$ of length p.

For fixed $\eta > 0$, the described procedure leads to some approximate matrix $\mathbf{V}_p \approx \mathbf{V}$. It is proven in [24, Satz 3.13] that $\mathbf{V}_p$ converges exponentially to $\mathbf{V}$ with respect to the Frobenius norm (and hence the ℓ_2-operator norm) as $p \to \infty$.

In `HILBERT`, we choose $\eta = 1/2$, if the optional parameter `eta` is not specified. If `eta` is given by the user, we set $\eta =$ `eta`. Note that for given `eta` ≤ 0 all entries of $\mathbf{V}$ are inadmissible and thus computed analytically. For `eta` > 0 or non-specified, certain entries are computed semi-analytically as described before, where we use a Gaussian quadrature of length $p = 16$. Different values of p can be chosen by modification of the file `source/geometry.h` and by re-building the integral operators, see Section 3.2.

The double-layer potential matrix $\mathbf{K}$ is obtained by call of

```
K = buildK(coordinates,elements [,eta]);
```

Note that the entries of $\mathbf{K}$ read

$$\mathbf{K}_{kj} = -\frac{1}{2\pi} \int_{E_k} \int_{\text{supp}(\zeta_j)} \frac{(y-x)\cdot \mathbf{n}_y}{|x-y|^2} \, \zeta_j(y) \, d\Gamma(y) \, d\Gamma(x),$$

where $\text{supp}(\zeta_j)$ denotes the support of ζ_j and where $\mathbf{n}_y \in \mathbb{R}^2$ denotes the (constant) outer normal vector on $y \in \Gamma$. For a mesh on a closed boundary Γ, $\text{supp}(\zeta_j)$ is the union of precisely two elements $E_i \in \mathcal{E}_\ell$. Therefore, the computation of $\mathbf{K}_{kj}$ needs the computation of double integrals of the type

$$\int_{E_k} \int_{E_i} \frac{(y-x)\cdot \mathbf{n}_j}{|x-y|^2} \, \zeta_j(y) \, d\Gamma(y) \, d\Gamma(x).$$

These can be computed analytically by use of anti-derivatives from [22]. For admissible pairs (E_i, E_k), we may proceed as described for $\mathbf{V}$. More precisely, we change the order of integration so that the smaller element corresponds to the outer integration, and we use numerical quadrature to compute the outer integral. As for $\mathbf{V}$, this provides an approximation $\mathbf{K}_p \approx \mathbf{K}$ which converges exponentially to $\mathbf{K}$ as $p \to \infty$.

LISTING 7. Build Mass Matrix

```
1 function M = buildM(coordinates,elements)
```

```
2
3  nE = size(elements,1);
4
5  %*** build vector of local mesh-size
6  h = sqrt(sum((coordinates(elements(:,1),:)-coordinates(elements(:,2),:)).^2,2));
7
8  %*** build coordinate format of sparse matrix M
9  I = reshape(repmat(1:nE,2,1),2*nE,1);
10 J = reshape(elements',2*nE,1);
11 A = reshape(repmat(0.5*h,1,2)',2*nE,1);
12
13 %*** build sparse matrix from coordinate format
14 M = sparse(I,J,A);
```

5.3. Building of the Mass Matrix M (Listing 7). Let $N \in \mathbb{N}$ be the number of nodes z_j in $\mathcal{E}_\ell$ and $M \in \mathbb{N}$ the number of elements E_j in $\mathcal{E}_\ell$. Note that e.g. for open boundaries, there holds $N \neq M$. The mass matrix $\mathbf{M} \in \mathbb{R}^{M \times N}$ is defined by

$$\mathbf{M}_{kj} = \langle \zeta_j \, , \, \chi_k \rangle_\Gamma \quad \text{for all } j = 1, \dots, N, \ k = 1, \dots, M.$$

Note that the entry $\mathbf{M}_{kj} = \langle \zeta_j \, , \, \chi_k \rangle_\Gamma = \int_{E_k} \zeta_j \, ds$ satisfies

$$\mathbf{M}_{kj} = \begin{cases} 0 & \text{if } z_j \notin \{z_m, z_n\}, \\ \text{length}(E_k)/2 & \text{if } z_j \in \{z_m, z_n\}, \end{cases}$$

where $E_k = [z_m, z_n] \in \mathcal{E}_\ell$. We thus may assemble the matrix $\mathbf{M}$ in the following way:

```
nE = size(elements,1);
M = sparse(nE,nE);
for k = 1:nE
    a = coordinates(elements(k,1),:);
    b = coordinates(elements(k,2),:);
    h = norm(b-a);
    M(k,elements(k,:)) = h/2;
end
```

We stress, however, that this implementation will lead to more than quadratic runtime with respect to the number M of elements. The reason for this is the internal storage of sparse matrices in MATLAB by use of the CCS format. This requires to sort the corresponding memory with every update of the sparse matrix and thus leads to a complexity $\mathcal{O}(k \log k)$ for $\mathcal{O}(k)$ non-zero entries. Since this is done for $k = 1, \dots, M$, one consequently expects a computational complexity of order $\mathcal{O}(M^2 \log M)$ which can be observed experimentally, cf. [19].

Building sparse matrices in MATLAB is efficiently done via the built-in function `sparse` which takes the coordinate format $I, J, A \in \mathbb{R}^{2M}$, where $\mathbf{M}_{ij} = \sum_{s=1}^{r} A_{k_s}$ for $i = I_{k_s}$ and $j = J_{k_s}$, $s = 1, \dots, r$. Altogether, the documentation of Listing 7 reads as follows:

- The function takes the mesh $\mathcal{E}_\ell$ described in terms of `coordinates` and `elements`.
- The column vector $h \in \mathbb{R}^N$ contains $h_j = \text{length}(E_j)$ (Line 6). We stress that the Euclidean length `h = norm(b-a)` can also be computed via `h = sqrt(sum((b-a).^2,2))` if $a, b \in \mathbb{R}^2$ are row-vectors. Then, the vectors $I, J, A \in \mathbb{R}^{2M}$ of the coordinate format of $\mathbf{M}$ are computed (Line 9–11), and the matrix $\mathbf{M}$ is built (Line 14).

LISTING 8. Build RHS for Symm's Integral Equation

```
1  function b = buildSymmRHS(coordinates,elements,uDh)
2  %*** compute DLP-matrix for P0 x S1
3  K = buildK(coordinates,elements);
4
5  %*** compute mass-type matrix for P0 x S1
```

```
6 M = buildM(coordinates,elements);
7
8 %*** build right-hand side vector
9 b = K*uDh + M*uDh*0.5;
```

5.4. Building of Right-Hand Side Vector (Listing 8). In this section, we aim at computing the vector

$$\mathbf{b} := \mathbf{K}\mathbf{g} + \frac{1}{2}\mathbf{M}\mathbf{g} \in \mathbb{R}^N \tag{5.19}$$

from (5.11). The documentation of Listing 8 reads as follows:

- The function takes the mesh $\mathcal{E}_\ell$ described in terms of `coordinates` and `elements` as well as column vector $\mathbf{g} =$ `uDh` which contains the nodal values for the discrete Dirichlet data $G_\ell \approx g$.
- We call the functions for the matrices $\mathbf{M}$ (Line 6) and $\mathbf{K}$ (Line 3).
- We assemble the right-hand side vector $\mathbf{b}$ (Line 9).

LISTING 9. Reliable Error Bound for $\|\phi - \phi_\ell\|_V$

```
1  function err = computeErrNeumann(coordinates,elements,p,phi)
2  %*** arbitrary quadrature on [-1,1] with exactness n >= 2, e.g., gauss(2)
3  quad_nodes = [-1 1]/sqrt(3);
4  quad_weights = [1;1];
5
6  %*** the remaining code is independent of the chosen quadrature rule
7  nE = size(elements,1);
8  nQ = length(quad_nodes);
9
10 %*** build vector of evaluations points as (nQ*nE x 2)-matrix
11 a = coordinates(elements(:,1),:);
12 b = coordinates(elements(:,2),:);
13 sx = reshape(a,2*nE,1)*(1-quad_nodes) + reshape(b,2*nE,1)*(1+quad_nodes);
14 sx = 0.5*reshape(sx',nQ*nE,2);
15
16 %*** phi(sx) usually depends on the normal vector, whence phi takes sx and the
17 %*** nodes of the respective element to compute the normal
18 a_sx = reshape(repmat(reshape(a,2*nE,1),1,nQ)',nE*nQ,2);
19 b_sx = reshape(repmat(reshape(b,2*nE,1),1,nQ)',nE*nQ,2);
20
21 %*** perform all necessary evaluations of phi as (nE x nQ)-matrix
22 phi_sx = reshape(phi(sx,a_sx,b_sx),nQ,nE)';
23
24 %*** compute vector of (squared) element-widths
25 h = sum((a-b).^2,2);
26
27 %*** compute Neumann error simultaneously for all elements
28 err_sx = (phi_sx - repmat(reshape(p,nE,1),1,nQ)).^2;
29 err = 0.5*h.*(err_sx*quad_weights);
```

5.5. Computation of Reliable Error Bound for $\|\phi - \Phi_\ell\|_V$ (Listing 9). We assume that the exact Neumann data satisfy $\phi \in L^2(\Gamma)$. Let $\Phi_\ell^* \in \mathcal{P}^0(\mathcal{E}_\ell)$ be the (only theoretically computed) Galerkin solution with respect to the non-perturbed right-hand side $(K + 1/2)g$ instead of $(K + 1/2)G_\ell$. Let Π_ℓ denote the L^2-orthogonal projection onto $\mathcal{P}^0(\mathcal{E}_\ell)$. With the

technique from [18, 5], we obtain

$$\|\!|\phi - \Phi_\ell^*|\!|_V \le |\!|\!|\phi - \Pi_\ell\phi|\!|\!|_V \lesssim \|h_\ell^{1/2}(\phi - \Pi_\ell\phi)\|_{L^2(\Gamma)}$$

as well as

$$|\!|\!|\Phi_\ell^* - \Phi_\ell|\!|\!|_V \lesssim \operatorname{osc}_{D,\ell},$$

where $\operatorname{osc}_{D,\ell}$ denote the Dirichlet data oscillations from Section 5.1.1. Note that Π_ℓ is even the $\mathcal{E}_\ell$-elementwise best approximation operator. With the triangle inequality, we therefore obtain

$$|\!|\!|\phi - \Phi_\ell|\!|\!|_V \lesssim \|h_\ell^{1/2}(\phi - \Phi_\ell)\|_{L^2(\Gamma)} + \operatorname{osc}_{D,\ell} =: \operatorname{err}_{N,\ell} + \operatorname{osc}_{D,\ell}.$$

In this section, we aim to numerically compute

$$\operatorname{err}_{N,\ell} = \Big(\sum_{j=1}^N \operatorname{err}_{N,\ell}(E_j)^2\Big)^{1/2}, \quad \text{where} \quad \operatorname{err}_{N,\ell}(E_j)^2 = \operatorname{length}(E_j)\,\|\phi - \Phi_\ell\|_{L^2(E_j)}^2.$$

With $\mathbf{x} \in \mathbb{R}^N$ the coefficient vector of

$$\Phi_\ell = \sum_{j=1}^N \mathbf{x}_j \chi_j,$$

there holds

$$\begin{aligned}\operatorname{err}_{N,\ell}(E_j)^2 = \operatorname{length}(E_j)\int_{E_j} |\phi - \mathbf{x}_j|^2\,d\Gamma &= \frac{\operatorname{length}(E_j)^2}{2}\int_{-1}^1 |\phi\circ\gamma_j(s) - \mathbf{x}_j|^2\,ds \\ &\approx \frac{\operatorname{length}(E_j)^2}{2}\operatorname{quad}_n\big((\phi\circ\gamma_j - \mathbf{x}_j)^2\big) =: \widetilde{\operatorname{err}}_{N,\ell}(E_j)^2, \qquad (5.20)\end{aligned}$$

where $\operatorname{quad}_n(\cdot)$ denotes a quadrature rule on $[-1,1]$ which is exact for polynomials of degree n, i.e. $\operatorname{quad}_n(p) = \int_{-1}^1 p\,ds$ for all $p \in \mathcal{P}^n[-1,1]$. With the definition $\widetilde{\operatorname{err}}_{N,\ell} := \big(\sum_{j=1}^N \widetilde{\operatorname{err}}_{N,\ell}(E_j)^2\big)^{1/2}$, one can then prove that

$$|\operatorname{err}_{N,\ell} - \widetilde{\operatorname{err}}_{N,\ell}| = \mathcal{O}(h^{n/2+1}).$$

For smooth ϕ, there holds $\operatorname{err}_{N,\ell} = \mathcal{O}(h^{3/2})$. For our implementation, we thus choose the Gauss quadrature with two nodes, which is exact for polynomials of degree $n = 3$. As for the Dirichlet data oscillations, this choice then leads to

$$|\operatorname{err}_{N,\ell} - \widetilde{\operatorname{err}}_{N,\ell}| = \mathcal{O}(h^{5/2}), \quad \text{whereas at most} \quad \operatorname{err}_{N,\ell} = \mathcal{O}(h^{3/2}),$$

i.e. our implementation is accurate up to higher-order terms. The documentation of Listing 9 now simply reads as follows:

- The function takes the given mesh $\mathcal{E}_\ell$ in form of the arrays `coordinates` and `elements`, the coefficient vector $\mathbf{p} = \mathbf{x}$ as well as a function handle `phi` for the Neumann data. The function `phi` is called by
  ```
  y = phi(x,a,b)
  ```
  with $(n \times 2)$-arrays `x`, `a`, and `b`. The j-th rows `x`$(j,:)$, `a`$(j,:)$, and `b`$(j,:)$ correspond to a point $x_j \in [a_j, b_j] \subset \mathbb{R}^2$. The entry `y`$(j)$ of the column vector `y` then contains $\phi(x_j)$.
- As stated above, we use the Gauss quadrature with two nodes (Line 3–4).
- If $s_k \in [-1,1]$ is a quadrature node and $E_j = [a_j, b_j] \in \mathcal{E}_\ell = \{E_1, \dots, E_N\}$ is an element, the function ϕ has to be evaluated at

  $$\gamma_j(s_k) = \frac{1}{2}\big(a_j + b_j + s_k(b_j - a_j)\big) = \frac{1}{2}\big(a_j(1 - s_k) + b_j(1 + s_k)\big).$$

  In Line 11–14, we build the $(2N \times 2)$-array `sx` which contains all necessary evaluation points. Note that the two evaluation points at E_j are stored in `sx`$(2j-1,:)$ and `sx`$(2j,:)$.
- In Line 18–19, we compute the $(2N \times 2)$-arrays `a_sx` and `b_sx` such that, e.g., `a_sx`$(2j-1,:)$ and `a_sx`$(2j,:)$ contain the first node $a_j \in \mathbb{R}^2$ of the boundary element $E_j = [a_j, b_j]$.

- We then evaluate the Neumann data ϕ simultaneously in all evaluation points and we reshape this $(2N \times 1)$-array into a $(N \times 2)$-array `phi_sx` such that `phi_sx`$(j,:)$ contains all ϕ-values related to E_j (Line 22).
- We realize Equation (5.20). We first derive the necessary evaluations of $(\phi - \mathbf{x}_j)^2$ in Line 28. Multiplication with the quadrature weights and coefficient-wise weighting with $\mathrm{length}(E_j)^2/2$ provides the $(N \times 1)$-array `err` such that `err`$(j) \approx \mathrm{length}(E_j)\|\phi - \Phi_\ell\|^2_{L^2(E_j)}$. More precisely, there holds $\mathrm{err}^2_{N,\ell} \approx \sum_{j=1}^N$ `err`$(j) = \widetilde{\mathrm{err}}^2_{N,\ell}$.

Remark 5.3. *In academic experiments, the exact solution ϕ is usually known and has certain regularity $\phi \in L^2(\Gamma)$ which only depends on the geometry of Γ. As explained before, there holds*

$$|||\phi - \Phi_\ell|||_V \lesssim \mathrm{err}_{N,\ell} + \mathrm{osc}_{D,\ell}$$

so that we can control the error reliably. Moreover, the convergence $\mathrm{err}_{N,\ell} \to 0$ as $\ell \to \infty$ might indicate that there are no major bugs in the implementation — since we compare the Galerkin solution with the exact solution. □

5.6. Computation of $(h - h/2)$-Based A Posteriori Error Estimators. In this section, we discuss the implementation of four error estimators introduced and analyzed in [18]. Let $\widehat{\mathcal{E}}_\ell = \{e_1, \dots, e_{2N}\}$ be the uniform refinement of the mesh $\mathcal{E}_\ell = \{E_1, \dots, E_N\}$. Let $\Phi_\ell \in \mathcal{P}^0(\mathcal{E}_\ell)$ and $\widehat{\Phi}_\ell \in \mathcal{P}^0(\widehat{\mathcal{E}}_\ell)$ be the Galerkin solutions (5.7) with respect to $\mathcal{E}_\ell$ and $\widehat{\mathcal{E}}_\ell$ and the same approximate Dirichlet data G_ℓ, i.e., there holds

$$\langle V\Phi_\ell\,,\,\Psi_\ell\rangle_V = \langle (K+1/2)G_\ell\,,\,\Psi_\ell\rangle_\Gamma \quad \text{for all } \Psi_\ell \in \mathcal{P}^0(\mathcal{E}_\ell) \tag{5.21}$$

as well as

$$\langle V\widehat{\Phi}_\ell\,,\,\widehat{\Psi}_\ell\rangle_V = \langle (K+1/2)G_\ell\,,\,\widehat{\Psi}_\ell\rangle_\Gamma \quad \text{for all } \widehat{\Psi}_\ell \in \mathcal{P}^0(\widehat{\mathcal{E}}_\ell). \tag{5.22}$$

With $\phi_\ell \in H^{-1/2}(\Gamma)$ the exact solution of (5.12), one can expect

$$|||\phi_\ell - \Phi_\ell|||_V \approx |||\widehat{\Phi}_\ell - \Phi_\ell|||_V =: \eta_\ell, \tag{5.23}$$

which results in

$$|||\phi - \Phi_\ell|||_V \le |||\phi - \phi_\ell|||_V + |||\phi_\ell - \Phi_\ell|||_V \lesssim \mathrm{osc}_{D,\ell} + \eta_\ell \tag{5.24}$$

according to the triangle inequality and (5.13).

Clearly, the Galerkin solution $\widehat{\Phi}_\ell$ with respect to the uniformly refined mesh $\widehat{\mathcal{E}}_\ell$ is more accurate than Φ_ℓ. Consequently, any algorithm will return $\widehat{\Phi}_\ell$ instead of Φ_ℓ if $\widehat{\Phi}_\ell$ has been computed. From this point of view, Φ_ℓ then becomes a side result and leads to unnecessary computational effort. One can prove that one may replace Φ_ℓ by a cheap (but appropriate) postprocessing $\Pi_\ell\widehat{\Phi}_\ell$ of $\widehat{\Phi}_\ell$. This leads to some error estimator

$$\eta_\ell \sim |||\widehat{\Phi}_\ell - \Pi_\ell\widehat{\Phi}_\ell|||_V =: \widetilde{\eta}_\ell \tag{5.25}$$

which always stays proportional to η_ℓ, indicated by $\eta_\ell \sim \widetilde{\eta}_\ell$. To be more precise, Π_ℓ denotes the L^2-orthogonal projection onto $\mathcal{P}^0(\mathcal{E}_\ell)$, which simply reads

$$(\Pi_\ell\widehat{\Phi}_\ell)|_{E_i} = \frac{1}{\mathrm{length}(E_i)}\int_{E_i}\widehat{\Phi}_\ell\,d\Gamma \quad \text{for all } E_i \in \mathcal{E}_\ell \tag{5.26}$$

in case of the lowest-order discretization, i.e. piecewise constant ansatz and test functions.

However, one essential drawback of the error estimators η_ℓ and $\widetilde{\eta}_\ell$ is that they do not provide an additional information on the local errors, i.e., the error $|||\phi - \Phi_\ell|||_V$ related to some element $E_i \in \mathcal{E}_\ell$. This is different for the error estimators μ_ℓ and $\widetilde{\mu}_\ell$ discussed in the following. For instance, one can prove that

$$\eta_\ell \sim \mu_\ell := \|h_\ell^{1/2}(\widehat{\Phi}_\ell - \Phi_\ell)\|_{L^2(\Gamma)} = \Big(\sum_{i=1}^N \mathrm{length}(E_i)\|\widehat{\Phi}_\ell - \Phi_\ell\|^2_{L^2(E_i)}\Big)^{1/2}. \tag{5.27}$$

Then, the local contributions

$$\mu_\ell(E_i) := \mathrm{length}(E_i)^{1/2} \|\widehat{\Phi}_\ell - \Phi_\ell\|_{L^2(E_i)} \quad \text{for all } E_i \in \mathcal{E}_\ell \tag{5.28}$$

give some measure for the error on E_i.

As the computation of the error estimator η_ℓ, the computation of μ_ℓ needs the computation of two Galerkin solutions Φ_ℓ and $\widehat{\Phi}_\ell$. As before, the computation of the coarse-mesh solution Φ_ℓ can be avoided by use of the projected fine mesh solution $\Pi_\ell \widehat{\Phi}_\ell$. One can mathematically prove that

$$\eta_\ell \sim \widetilde{\mu}_\ell := \|h_\ell^{1/2}(\widehat{\Phi}_\ell - \Pi_\ell \widehat{\Phi}_\ell)\|_{L^2(\Gamma)}. \tag{5.29}$$

In the following subsections, we first discuss the computation of the *global* error estimators η_ℓ and $\widetilde{\eta}_\ell$ from (5.23) and (5.25). Then, we give an implementation of the *local* error estimators μ_ℓ and $\widetilde{\mu}_\ell$ from (5.27) and (5.29), where our functions return the local contributions, see e.g. (5.28), to steer an adaptive mesh-refinement.

Remark 5.4. *If we plot the error estimators η_ℓ, $\widetilde{\eta}_\ell$, μ_ℓ and $\widetilde{\mu}_\ell$ over the number of elements, one can mathematically predict that the corresponding curves, for a sequence of arbitrarily refined meshes, are parallel. In mathematical terms, this reads*

$$\eta_\ell \le \widetilde{\eta}_\ell \lesssim \widetilde{\mu}_\ell \le \mu_\ell \lesssim \eta_\ell, \tag{5.30}$$

cf. [15, 18]. *Empirically, one observes a very good coincidence of η_ℓ and $\widetilde{\eta}_\ell$ in the sense that the corresponding curves almost coincide. The same is observed for the curves of μ_ℓ and $\widetilde{\mu}_\ell$.* □

Remark 5.5. *Mathematically, the error estimate* (5.24) *involves the so-called saturation assumption: Assume that we could compute the Galerkin solutions Φ_ℓ^* and $\widehat{\Phi}_\ell^*$ with respect to $\mathcal{E}_\ell$ and $\widehat{\mathcal{E}}_\ell$ for the non-perturbed variational formulation* (5.3)*, i.e., we formally use the exact Dirichlet data g instead of the interpolated data G_ℓ — although the right-hand side is, in fact, non-computable because of Kg. Then, the saturation assumption states that*

$$|||\phi - \widehat{\Phi}_\ell^*|||_V \le q\, |||\phi - \Phi_\ell^*|||_V \tag{5.31}$$

with some uniform and ℓ-independent constant $q \in (0,1)$. —Put differently, uniform mesh-refinement leads to a uniform improvement of the discretization error.— Provided (5.31)*, one can prove that*

$$\eta_\ell \le |||\phi_\ell - \Phi_\ell|||_V \le (1-q^2)^{-1/2}\, \eta_\ell \tag{5.32}$$

which is the mathematical basis of (5.23)*, cf.* [18].

We stress that this assumption is somewhat natural and can, for instance, be proven for the finite element method [14]*, see also* [17, Section 2.3]*. For the boundary element method, however,* (5.31) *still remains open.*

Finally, one can prove that (5.31) *is sufficient and in some sense even necessary to guarantee* (5.24). □

Remark 5.6. *In academic experiments, the exact solution ϕ is usually known and has certain regularity $\phi \in L^2(\Gamma)$ which only depends on the geometry of Γ. In this case, one can experimentally verify the saturation assumption as follows: In Section 5.5, we derived*

$$|||\phi - \Phi_\ell|||_V \lesssim \mathrm{err}_{N,\ell} + \mathrm{osc}_{D,\ell}.$$

If the right-hand side has the same convergence behaviour as the error estimator $\eta_\ell + \mathrm{osc}_{D,\ell}$, this proves empirically

$$|||\phi - \Phi_\ell|||_V \lesssim \eta_\ell + \mathrm{osc}_{D,\ell}$$

and confirms the saturation assumption. □

LISTING 10. Computation of Estimator η_ℓ

```
function est = computeEstSlpEta(father2son,V_fine,x_fine,x_coarse)
```

```
%*** compute coefficient vector of (phi_fine - phi_coarse) w.r.t. to fine mesh
x_fine(father2son(:,1)) = x_fine(father2son(:,1)) - x_coarse;
x_fine(father2son(:,2)) = x_fine(father2son(:,2)) - x_coarse;

%*** compute energy ||| phi_fine - phi_coarse |||^2
est = x_fine'*(V_fine*x_fine);
```

5.6.1. Computation of Error Estimator η_ℓ (Listing 10). In this section, we aim to compute the error estimator $\eta_\ell = \|\!|\widehat{\Phi}_\ell - \Phi_\ell|\!\|_V$ from (5.23). Let $\widehat{\chi}_j$ denote the characteristic function associated with some fine-mesh element $e_j \in \widehat{\mathcal{E}}_\ell$. Let $\mathbf{x} \in \mathbb{R}^N$ and $\widehat{\mathbf{x}} \in \mathbb{R}^{2N}$ be the coefficient vectors of Φ_ℓ and $\widehat{\Phi}_\ell$ with respect to the canonical bases of $\mathcal{P}^0(\mathcal{E}_\ell)$ and $\mathcal{P}^0(\widehat{\mathcal{E}}_\ell)$, i.e.

$$\Phi_\ell = \sum_{j=1}^{N} \mathbf{x}_j \chi_j \quad \text{and} \quad \widehat{\Phi}_\ell = \sum_{j=1}^{2N} \widehat{\mathbf{x}}_j \widehat{\chi}_j.$$

Because of $\mathcal{P}^0(\mathcal{E}_\ell) \subset \mathcal{P}^0(\widehat{\mathcal{E}}_\ell)$, there is a unique vector $\widehat{\mathbf{y}} \in \mathbb{R}^{2N}$ such that

$$\Phi_\ell = \sum_{j=1}^{2N} \widehat{\mathbf{y}}_j \widehat{\chi}_j.$$

With the vectors $\widehat{\mathbf{x}}, \widehat{\mathbf{y}} \in \mathbb{R}^{2N}$, there holds

$$\begin{aligned}\eta_\ell^2 = \|\!|\widehat{\Phi}_\ell - \Phi_\ell|\!\|_V^2 = \langle\!\langle \widehat{\Phi}_\ell - \Phi_\ell \,, \widehat{\Phi}_\ell - \Phi_\ell \rangle\!\rangle_V &= \sum_{j,k=1}^{2N} (\widehat{\mathbf{x}}_j - \widehat{\mathbf{y}}_j)(\widehat{\mathbf{x}}_k - \widehat{\mathbf{y}}_k) \langle\!\langle \widehat{\chi}_j \,, \widehat{\chi}_k \rangle\!\rangle_V \\ &= (\widehat{\mathbf{x}} - \widehat{\mathbf{y}}) \cdot \widehat{\mathbf{V}} (\widehat{\mathbf{x}} - \widehat{\mathbf{y}}),\end{aligned}$$

where $\widehat{\mathbf{V}}$ is the matrix for the simple-layer potential (5.10) with respect to the fine mesh $\widehat{\mathcal{E}}_\ell$. With these observations, the documentation of Listing 10 reads as follows:

- The function takes the coefficient vectors $\mathbf{x} \in \mathbb{R}^N$ and $\widehat{\mathbf{x}} \in \mathbb{R}^{2N}$ of the Galerkin solutions Φ_ℓ and $\widehat{\Phi}_\ell$ as well as the simple-layer potential matrix $\widehat{\mathbf{V}}$ for the fine mesh $\widehat{\mathcal{E}}_\ell$. Besides this, the $(N \times 2)$-array `father2son` links the indices of elements $E_i \in \mathcal{E}_\ell$ with the indices of the sons $e_j, e_k \in \widehat{\mathcal{E}}_\ell$ in the sense that `father2son(`i`,:) = [`j,k`]` for $E_i = e_j \cup e_k$ and consequently $\widehat{\mathbf{y}}_j = \widehat{\mathbf{y}}_k = \mathbf{x}_i$.
- We overwrite the vector $\widehat{\mathbf{x}}$ by the coefficient vector $\widehat{\mathbf{x}} - \widehat{\mathbf{y}}$ of $\widehat{\Phi}_\ell - \Phi_\ell$ (Line 3–4).
- Finally, the function returns $\eta_\ell^2 = \|\!|\widehat{\Phi}_\ell - \Phi_\ell|\!\|_V^2$ (Line 7).

LISTING 11. Computation of Estimator $\widetilde{\eta}_\ell$

```
function est = computeEstSlpEtaTilde(father2son,V_fine,x_fine)
%*** compute L2-projection Pi_coarse*phi_fine onto coarse mesh
pi_x_fine = 0.5*( x_fine(father2son(:,1)) + x_fine(father2son(:,2)) );

%*** compute coefficient vector of (1-Pi_coarse)*phi_fine
x_fine(father2son(:,1)) = x_fine(father2son(:,1)) - pi_x_fine;
x_fine(father2son(:,2)) = x_fine(father2son(:,2)) - pi_x_fine;

%*** compute energy ||| (1-Pi_coarse)*phi_fine |||^2
est = x_fine'*(V_fine*x_fine);
```

5.6.2. Computation of Error Estimator $\widetilde{\eta}_\ell$ (Listing 11). We adopt the notation of Section 5.6.1 for the computation of η_ℓ, namely $\widehat{\mathbf{x}} \in \mathbb{R}^{2N}$ with

$$\widehat{\Phi}_\ell = \sum_{j=1}^{2N} \widehat{\mathbf{x}}_j \widehat{\chi}_j.$$

Let $e_j, e_k \in \widehat{\mathcal{E}}_\ell$ be the sons of $E_i \in \mathcal{E}_\ell$, i.e. $E_i = e_j \cup e_k$. Then,

$$\int_{E_i} \widehat{\Phi}_\ell \, d\Gamma = \int_{e_j} \widehat{\Phi}_\ell \, d\Gamma + \int_{e_k} \widehat{\Phi}_\ell \, d\Gamma = \operatorname{length}(e_j)\, \widehat{\mathbf{x}}_j + \operatorname{length}(e_k)\, \widehat{\mathbf{x}}_k = \operatorname{length}(E_i)\, \frac{\widehat{\mathbf{x}}_j + \widehat{\mathbf{x}}_k}{2}.$$

Put differently, there holds

$$(\Pi_\ell \widehat{\Phi}_\ell)|_{E_i} = \frac{\widehat{\mathbf{x}}_j + \widehat{\mathbf{x}}_k}{2} \quad \text{for all } E_i \in \mathcal{E}_\ell \quad \text{with} \quad E_i = e_j \cup e_k \text{ and } e_j, e_k \in \widehat{\mathcal{E}}_\ell$$

for the L^2-projection Π_ℓ defined in (5.26). Representing $\Pi_\ell \widehat{\Phi}_\ell \in \mathcal{P}^0(\mathcal{E}_\ell)$ with respect to the fine-mesh $\widehat{\mathcal{E}}_\ell$, we obtain

$$\Pi_\ell \widehat{\Phi}_\ell = \sum_{n=1}^{2N} \widehat{\mathbf{z}}_n \widehat{\chi}_n,$$

where the vector $\widehat{\mathbf{z}} \in \mathbb{R}^{2N}$ satisfies $\widehat{\mathbf{z}}_j = \widehat{\mathbf{z}}_k = \frac{\widehat{\mathbf{x}}_j + \widehat{\mathbf{x}}_k}{2}$ provided that $e_j, e_k \in \widehat{\mathcal{E}}_\ell$ are the sons of some element $E_i \in \mathcal{E}_\ell$. As in Section 5.6.1, there holds

$$\widetilde{\eta}_\ell^2 = \|\widehat{\Phi}_\ell - \Pi_\ell \widehat{\Phi}_\ell\|_V^2 = (\widehat{\mathbf{x}} - \widehat{\mathbf{z}}) \cdot \widehat{\mathbf{V}}(\widehat{\mathbf{x}} - \widehat{\mathbf{z}}).$$

Therefore, the documentation of Listing 11 reads as follows:

- The function takes the simple-layer potential matrix $\widehat{\mathbf{V}}$ for the fine mesh $\widehat{\mathcal{E}}_\ell$ and the coefficient vector $\widehat{\mathbf{x}} \in \mathbb{R}^{2N}$ of $\widehat{\Phi}_\ell$. Moreover, the link between $\mathcal{E}_\ell$ and $\widehat{\mathcal{E}}_\ell$ is provided by means of `father2son`.
- We first compute the coefficient vector of $\Pi_\ell \widehat{\Phi}_\ell$ with respect to the coarse mesh $\mathcal{E}_\ell$ (Line 3).
- We then overwrite $\widehat{\mathbf{x}}$ by the coefficient vector $\widehat{\mathbf{x}} - \widehat{\mathbf{z}} \in \mathbb{R}^{2N}$ of $\widehat{\Phi}_\ell - \Pi_\ell \widehat{\Phi}_\ell$ (Line 6–7).
- Finally, the function returns $\widetilde{\eta}_\ell^2 = \|\widehat{\Phi}_\ell - \Pi_\ell \widehat{\Phi}_\ell\|_V^2$ (Line 10).

LISTING 12. Computation of Estimator μ_ℓ

```
1  function ind = computeEstSlpMu(coordinates,elements,father2son,x_fine,x_coarse)
2  %*** compute (squared) local mesh-size
3  h = sum((coordinates(elements(:,1),:) - coordinates(elements(:,2),:)).^2,2);
4
5  %*** compute coefficient vector of (phi_fine - phi_coarse) w.r.t. to fine mesh
6  x_fine(father2son(:,1)) = x_fine(father2son(:,1)) - x_coarse;
7  x_fine(father2son(:,2)) = x_fine(father2son(:,2)) - x_coarse;
8
9  %*** compute ind(j) = diam(Ej)*|| phi_fine - phi_coarse ||_{L2(Ej)}^2
10 ind = 0.5*h.*( x_fine(father2son(:,1)).^2 + x_fine(father2son(:,2)).^2 );
```

5.6.3. Computation of Error Estimator μ_ℓ (Listing 12). In this section, we discuss the implementation of

$$\mu_\ell^2 = \sum_{i=1}^{N} \mu_\ell(E_i)^2, \quad \text{where} \quad \mu_\ell(E_i)^2 = \operatorname{length}(E_i)\, \|\widehat{\Phi}_\ell - \Phi_\ell\|_{L^2(E_i)}^2.$$

We adopt the notation of Section 5.6.1, namely $\widehat{\mathbf{x}}, \widehat{\mathbf{y}} \in \mathbb{R}^{2N}$ with

$$\widehat{\Phi}_\ell = \sum_{j=1}^{2N} \widehat{\mathbf{x}}_j \widehat{\chi}_j \quad \text{and} \quad \Phi_\ell = \sum_{j=1}^{2N} \widehat{\mathbf{y}}_j \widehat{\chi}_j.$$

For fixed $E_i \in \mathcal{E}_\ell$ and sons $e_j, e_k \in \widehat{\mathcal{E}}_\ell$ with $E_i = e_j \cup e_k$, we obtain

$$\|\widehat{\Phi}_\ell - \Phi_\ell\|^2_{L^2(E_i)} = \int_{e_j} (\widehat{\Phi}_\ell - \Phi_\ell)^2 \, d\Gamma + \int_{e_k} (\widehat{\Phi}_\ell - \Phi_\ell)^2 \, d\Gamma = \frac{\operatorname{length}(E_i)}{2} \big((\widehat{\mathbf{x}}_j - \widehat{\mathbf{y}}_j)^2 + (\widehat{\mathbf{x}}_k - \widehat{\mathbf{y}}_k)^2 \big).$$

This implies

$$\mu_\ell(E_i)^2 = \frac{\operatorname{length}(E_i)^2}{2} \big((\widehat{\mathbf{x}}_j - \widehat{\mathbf{y}}_j)^2 + (\widehat{\mathbf{x}}_k - \widehat{\mathbf{y}}_k)^2 \big). \tag{5.33}$$

Altogether, the documentation of Listing 12 reads as follows:

- As input arguments, the function takes the mesh $\mathcal{E}_\ell$, the link between $\mathcal{E}_\ell$ and $\widehat{\mathcal{E}}_\ell$, and the coefficient vectors $\mathbf{x} \in \mathbb{R}^N$ and $\widehat{\mathbf{x}} \in \mathbb{R}^{2N}$ of the Galerkin solutions Φ_ℓ and $\widehat{\Phi}_\ell$ (Line 1).
- We compute the vector of all squared element-sizes (Line 3).
- We overwrite the coefficient vector $\widehat{\mathbf{x}}$ of $\widehat{\Phi}_\ell$ by the coefficient vector $\widehat{\mathbf{x}} - \widehat{\mathbf{y}}$ of $\widehat{\Phi}_\ell - \Phi_\ell$ (Line 6–7).
- Finally (Line 10), the function realizes (5.33) and returns the vector

  $$\mathbf{v} := (\mu_\ell(E_1)^2, \dots, \mu_\ell(E_N)^2) \in \mathbb{R}^N$$

  so that $\mu_\ell = \big(\sum_{i=1}^N \mathbf{v}_i \big)^{1/2}$.

LISTING 13. Computation of Estimator $\widetilde{\mu}_\ell$

```
1  function ind = computeEstSlpMuTilde(coordinates,elements,father2son,x_fine)
2  %*** compute (squared) local mesh-size
3  h = sum((coordinates(elements(:,1),:) - coordinates(elements(:,2),:)).^2,2);
4
5  %*** compute L2-projection Pi_coarse*phi_fine onto coarse mesh
6  pi_x_fine = 0.5*( x_fine(father2son(:,1)) + x_fine(father2son(:,2)) );
7
8  %*** compute coefficient vector of (1-Pi_coarse)*phi_fine
9  x_fine(father2son(:,1)) = x_fine(father2son(:,1)) - pi_x_fine;
10 x_fine(father2son(:,2)) = x_fine(father2son(:,2)) - pi_x_fine;
11
12 %*** compute ind(j) = diam(Ej)*|| (1-Pi_coarse)*phi_fine ||_{L2(Ej)}^2
13 ind = 0.5*h.*( x_fine(father2son(:,1)).^2 + x_fine(father2son(:,2)).^2 );
```

5.6.4. Computation of Error Estimator $\widetilde{\mu}_\ell$ (Listing 13). In this section, we finally aim to compute

$$\widetilde{\mu}_\ell^2 = \sum_{i=1}^N \widetilde{\mu}_\ell(E_i)^2, \quad \text{where} \quad \widetilde{\mu}_\ell(E_i)^2 = \operatorname{length}(E_i) \, \|\widehat{\Phi}_\ell - \Pi_\ell \widehat{\Phi}_\ell\|^2_{L^2(E_i)}.$$

We adopt the notation of the preceding Sections 5.6.1–5.6.3, namely $\widehat{\mathbf{x}}, \widehat{\mathbf{z}} \in \mathbb{R}^{2N}$ with

$$\widehat{\Phi}_\ell = \sum_{j=1}^{2N} \widehat{\mathbf{x}}_j \widehat{\chi}_j \quad \text{and} \quad \Pi_\ell \widehat{\Phi}_\ell = \sum_{j=1}^{2N} \widehat{\mathbf{z}}_j \widehat{\chi}_j.$$

Based on this, the abbreviate documentation of Listing 13 reads as follows:

- The function takes the mesh $\mathcal{E}_\ell$, the link between $\mathcal{E}_\ell$ and $\widehat{\mathcal{E}}_\ell$, and the coefficient vectors $\widehat{\mathbf{x}} \in \mathbb{R}^{2N}$ of $\widehat{\Phi}_\ell$ (Line 1). It overwrites $\widehat{\mathbf{x}}$ by the coefficient vector $\widehat{\mathbf{x}} - \widehat{\mathbf{z}}$ of $\widehat{\Phi}_\ell - \Pi_\ell \widehat{\Phi}_\ell$ (Line 6–10).

- Finally (Line 13), the function returns the vector

$$\mathbf{v} := (\widetilde{\mu}_\ell(E_1)^2, \dots, \widetilde{\mu}_\ell(E_N)^2) \in \mathbb{R}^N.$$

In particular, there holds $\widetilde{\mu}_\ell = \big(\sum_{i=1}^N \mathbf{v}_i\big)^{1/2}$.

5.7. Adaptive Mesh-Refinement. Usually computing time and memory requirements are limiting quantities for numerical simulations. Therefore, one aims to choose the mesh such that it is coarse, where the (unknown) solution is smooth, and fine, where the (unknown) solution is singular. Based on a local error estimator, e.g. $\widetilde{\mu}_\ell$, such meshes are constructed in an iterative way. In each step, one refines the mesh only locally, i.e. one refines elements E_j, where the error appears to be large, namely, where the local contributions $\widetilde{\mu}_\ell(E_j)$ are large. For the error estimator $\widetilde{\mu}_\ell$ from Section 5.6.4, a possible adaptive algorithm reads as follows:

Input: Initial mesh $\mathcal{E}_0$, Dirichlet data g, adaptivity parameter $0 < \theta < 1$, maximal number $N_{\max} \in \mathbb{N}$ of elements, and counter $\ell = 0$.

(i) Build uniformly refined mesh $\widehat{\mathcal{E}}_\ell$.
(ii) Compute Galerkin solution $\widehat{\Phi}_\ell \in \mathcal{P}^0(\widehat{\mathcal{E}}_\ell)$.
(iii) Compute refinement indicators $\widetilde{\mu}_\ell(E)^2$ and oscillation terms $\operatorname{osc}_{D,\ell}(E)^2$ for all $E \in \mathcal{E}_\ell$.
(iv) Find minimal set $\mathcal{M}_\ell \subseteq \mathcal{E}_\ell$ such that

$$\theta\,(\widetilde{\mu}_\ell^2 + \operatorname{osc}_{D,\ell}^2) = \theta \sum_{E\in\mathcal{E}_\ell} \widetilde{\mu}_\ell(E)^2 + \operatorname{osc}_{D,\ell}(E)^2 \le \sum_{E\in\mathcal{M}_\ell} \widetilde{\mu}_\ell(E)^2 + \operatorname{osc}_{D,\ell}(E)^2. \tag{5.34}$$

(v) Refine at least marked elements $E \in \mathcal{M}_\ell$ and obtain mesh $\mathcal{E}_{\ell+1}$ with $\kappa(\mathcal{E}_{\ell+1}) \le 2\kappa(\mathcal{E}_0)$.
(vi) Stop provided that $\#\mathcal{E}_{\ell+1} \ge N_{\max}$; otherwise, increase counter $\ell \mapsto \ell + 1$ and go to (i).

Output: Adaptively generated mesh $\widehat{\mathcal{E}}_\ell$ and corresponding discrete solution $\widehat{\Phi}_\ell \in \mathcal{P}^0(\widehat{\mathcal{E}}_\ell)$.

The marking criterion (5.34) has been proposed in the context of adaptive finite element methods [13]. Let formally $N_{\max} = \infty$ so that the adaptive algorithm computes a sequence of discrete solutions $\widehat{\Phi}_\ell$ (or even Φ_ℓ, although this is not computed). In [17, Section 3], we prove that the saturation assumption (5.31) implies convergence of $\widehat{\Phi}_\ell$ and Φ_ℓ to ϕ, provided that the right-hand side g is not disturbed, i.e., $g = G_\ell$. The same result also holds for $\widetilde{\mu}_\ell$ replaced by μ_ℓ.

In [4], we changed the notion of convergence and proved that for certain error estimators — amongst them are $\widetilde{\mu}_\ell$ and μ_ℓ— the adaptive algorithm guarantees $\lim_\ell \widetilde{\mu}_\ell = 0$. This concept is followed in [5] to prove that the adaptive algorithm stated above, yields $\lim_\ell(\mu_\ell^2 + \operatorname{osc}_{D,\ell}^2) = 0$. If the saturation assumption (5.31) holds (at least in infinitely many steps), we obtain convergence of Φ_ℓ to ϕ due to $\|\!|\phi - \Phi_\ell|\!\|_V^2 \lesssim \mu_\ell^2 + \operatorname{osc}_{D,\ell}^2$.

For adaptive finite element schemes, it could recently be proven that adaptive algorithms of this type even lead to quasi-optimal meshes [10]. For adaptive BEM, such a result is completely open although numerical experiments give evidence for such an optimality result.

LISTING 14. Implementation of Adaptive Algorithm

```
1  % adaptiveSymm provides the implementation of an adaptive mesh-refining
2  % algorithm for Symm's integral equation.
3  %*** maximal number of elements
4  nEmax = 200;
5
6  %*** adaptivity parameter
7  theta = 0.25;
8  rho = 0.25;
9
10 %*** adaptive mesh-refining algorithm
11 while 1
```

```
12
13      fprintf('number of elements: N = %d\r',size(elements,1))
14
15      %*** build uniformly refined mesh
16      [coordinates_fine,elements_fine,father2son] ...
17          = refineBoundaryMesh(coordinates,elements);
18
19      %*** discretize Dirichlet data and compute data oscillations
20      [osc_fine,uDh_fine] = computeOscDirichlet(coordinates_fine,elements_fine,@g);
21      osc = osc_fine(father2son(:,1)) + osc_fine(father2son(:,2));
22
23      %*** compute fine-mesh solution
24      V_fine = buildV(coordinates_fine,elements_fine);
25      b_fine = buildSymmRHS(coordinates_fine,elements_fine,uDh_fine);
26      x_fine = V_fine\b_fine;
27
28      %*** compute (h-h/2)-error estimator tilde-mu
29      mu_tilde = computeEstSlpMuTilde(coordinates,elements,father2son, ...
30                                              x_fine);
31
32      %*** mark elements for refinement
33      marked = markElements(theta,rho,mu_tilde + osc);
34
35      %*** generate new mesh
36      [coordinates,elements] = refineBoundaryMesh(coordinates,elements,marked);
37
38      if size(elements,1) > nEmax
39        break;
40      end
41  end
42
43  %*** visualize exact and adaptively computed solution
44  plotArclengthP0(coordinates_fine,elements_fine,x_fine,@phi,1);
```

5.7.1. Implementation of Adaptive Algorithm (Listing 14). The MATLAB script of Listing 14 realizes the adaptive algorithm from the beginning of this section.

- We use the adaptivity parameter $\theta = 1/4$ in (5.34) and mark at least the 25% of elements with the largest indicators (Line 7–8).
- Recall that the function **computeEstSlpMuTilde** as well as **computeOscDirichlet** return vectors of quadratic terms $\widetilde{\mu}_\ell(E)^2$ and $\operatorname{osc}_\ell(E)^2$, respectively. Note that (5.34) corresponds to the choice $\varrho_\ell(E) := \widetilde{\mu}_\ell(E)^2 + \operatorname{osc}_\ell(E)^2$ in (4.1). Therefore, the marking criterion (4.1) is provided by means of the function **markElements** (Line 33).

6. Hypersingular Integral Equation

Continuous Model Problem. In the entire section, we consider the hypersingular integral equation

$$Wu = (1/2 - K')\phi \quad \text{on } \Gamma \tag{6.1}$$

with W the hypersingular integral operator and K' the adjoint double-layer potential, where $\Gamma = \partial\Omega$ is the piecewise-affine boundary of a polygonal Lipschitz domain $\Omega \subset \mathbb{R}^2$. For technical reasons, we have to assume that Γ is connected, i.e. Ω is simply-connected. This integral equation is an equivalent formulation of the Neumann problem

$$-\Delta u = 0 \text{ in } \Omega \quad \text{with} \quad \partial_n u = \phi \text{ on } \Gamma. \tag{6.2}$$

Note that due to the Gauss Divergence Theorem there holds

$$\int_\Gamma \phi \, d\Gamma = \int_{\partial\Omega} \partial_n u \, d\Gamma = \int_\Omega \Delta u \, dx = 0.$$

Formally, the Neumann data satisfy $\phi \in H_*^{-1/2}(\Gamma)$, where the subscript abbreviates the constraint $\langle \phi \, , \, 1 \rangle_\Gamma = 0$. We will, however, assume additional regularity $\phi \in C(\mathcal{E}_\ell) \subset L^2(\Gamma) \subset H^{-1/2}(\Gamma)$. The exact solution $u \in H^{1/2}(\Gamma)$ of the integral formulation (6.1) is just the Dirichlet data $u|_\Gamma$ of the solution $u \in H^1(\Omega)$ of (6.2).

Due to the fact that there holds $Wc = 0$ for all constant functions $c \in \mathbb{R}$, the solutions of (6.1) and (6.2) are only unique up to additive constants. To fix the additive constant, one usually assumes integral mean zero for the respective solutions. In this sense, (6.1) can equivalently be formulated in variational form: Find $u_* \in H_*^{1/2}(\Gamma) := \big\{ v \in H^{1/2}(\Gamma) \, : \, \int_\Gamma v \, d\Gamma = 0 \big\}$ such that

$$\langle W u_* \, , \, v_* \rangle_\Gamma = \langle (1/2 - K') \phi \, , \, v_* \rangle_\Gamma \quad \text{for all } v_* \in H_*^{1/2}(\Gamma). \tag{6.3}$$

One can prove that this formulation has a unique solution, since the left-hand side defines a scalar product on $H_*^{1/2}(\Gamma)$ even with equivalent norms. (We stress that here our assumption enters that Γ is connected. Otherwise, the kernel of W strictly contains the constant functions, and W is thus *not* elliptic on $H_*^{1/2}(\Gamma)$.)

From another point of view, one can consider the bilinear form

$$\langle\!\langle u \, , v \rangle\!\rangle_{W+S} := \langle W u \, , \, v \rangle_\Gamma + \Big(\int_\Gamma u \, d\Gamma \Big) \Big(\int_\Gamma v \, d\Gamma \Big) \quad \text{for all } u, v \in H^{1/2}(\Gamma), \tag{6.4}$$

which leads to the following modified variational form: Find $u \in H^{1/2}(\Gamma)$ such that

$$\langle\!\langle u \, , v \rangle\!\rangle_{W+S} = \langle (1/2 - K') \phi \, , \, v \rangle_\Gamma \quad \text{for all } v \in H^{1/2}(\Gamma). \tag{6.5}$$

One can prove that $\langle\!\langle \cdot \, , \cdot \rangle\!\rangle_{W+S}$ from (6.4) defines a scalar product such that the induced norm $|\!|\!| u |\!|\!|_{W+S} := \langle\!\langle u \, , u \rangle\!\rangle_{W+S}^{1/2}$ is an equivalent norm on $H^{1/2}(\Gamma)$. Consequently, (6.5) has a unique solution u which depends continuously on the Neumann data ϕ. Moreover, one can prove that

$$\langle (1/2 - K') \psi \, , \, 1 \rangle_\Gamma = \frac{1}{2} \langle \psi \, , \, 1 \rangle_\Gamma - \langle \psi \, , \, K1 \rangle_\Gamma = \langle \psi \, , \, 1 \rangle_\Gamma = 0 \quad \text{for all } \psi \in H_*^{-1/2}(\Gamma).$$

If we plug in $v = 1$ in (6.5), we thus obtain

$$\begin{aligned} \Big(\int_\Gamma u \, d\Gamma \Big) \operatorname{length}(\Gamma) &= \langle W u \, , \, 1 \rangle_\Gamma + \Big(\int_\Gamma u \, d\Gamma \Big) \Big(\int_\Gamma 1 \, d\Gamma \Big) \\ &= \langle\!\langle u \, , 1 \rangle\!\rangle_{W+S} = \langle (1/2 - K') \phi \, , \, 1 \rangle_\Gamma = 0 \end{aligned}$$

according to the fact that the kernel of the hypersingular integral operator W consists of constant functions. This implies $u \in H_*^{1/2}(\Gamma)$. For a test function $v_* \in H_*^{1/2}(\Gamma)$, the variational formulation (6.5) thus becomes

$$\langle W u \, , \, v_* \rangle_\Gamma = \langle\!\langle u \, , v_* \rangle\!\rangle_{W+S} = \langle (1/2 - K') \phi \, , \, v_* \rangle_\Gamma,$$

i.e. (6.5) reduces to (6.3). Altogether we obtain that the unique solution u of (6.5) is also the unique solution of (6.3), i.e., (6.5) is an equivalent formulation of (6.3).

Galerkin Discretization. To discretize (6.5), we first replace the Neumann data $\phi \in H_*^{-1/2}(\Gamma) \cap L^2(\Gamma)$ by its L^2-projection $\Phi_\ell \in \mathcal{P}^0(\mathcal{E}_\ell)$,

$$\Phi_\ell|_{E_j} = \frac{1}{\operatorname{length}(E_j)} \int_{E_j} \phi \, d\Gamma =: \mathbf{p}_j \quad \text{for all } E_j \in \mathcal{E}_\ell. \tag{6.6}$$

According to this definition, there holds

$$\int_\Gamma \Phi_\ell \, d\Gamma = \sum_{E \in \mathcal{E}_\ell} \int_E \Phi_\ell \, d\Gamma = \sum_{E \in \mathcal{E}_\ell} \int_E \phi \, d\Gamma = \int_\Gamma \phi \, d\Gamma = 0,$$

i.e. there holds $\Phi_\ell \in H_*^{-1/2}(\Gamma)$, too. Second, we replace the function space $H^{1/2}(\Gamma)$ in (6.5) by the finite-dimensional space $\mathcal{S}^1(\mathcal{E}_\ell)$. Since $\mathcal{S}^1(\mathcal{E}_\ell)$ is a subspace of $H^{1/2}(\Gamma)$, $\langle\!\langle \cdot\,, \cdot \rangle\!\rangle_{W+S}$ from (6.4) is also a scalar product on $\mathcal{S}^1(\mathcal{E}_\ell)$. Consequently, there exists a unique Galerkin solution $U_\ell \in \mathcal{S}^1(\mathcal{E}_\ell)$ of the discretized problem

$$\langle\!\langle U_\ell\,, V_\ell \rangle\!\rangle_{W+S} = \langle (1/2 - K')\Phi_\ell\,,\, V_\ell\rangle_\Gamma \quad \text{for all } V_\ell \in \mathcal{S}^1(\mathcal{E}_\ell). \tag{6.7}$$

As in the continuous case, the discrete solution U_ℓ automatically satisfies $\int_\Gamma U_\ell \, d\Gamma = 0$ which follows from $1 = \sum_{j=1}^N \zeta_j \in \mathcal{S}^1(\mathcal{E}_\ell)$, which allows us to plug in $V_\ell = 1$ in (6.7). Indeed,

$$\begin{aligned}\Big(\int_\Gamma U_\ell \, d\Gamma\Big)\,\mathrm{length}(\Gamma) &= \langle W U_\ell\,,\, 1\rangle_\Gamma + \Big(\int_\Gamma U_\ell \, d\Gamma\Big)\Big(\int_\Gamma 1 \, d\Gamma\Big)\\ &= \langle\!\langle U_\ell\,, 1\rangle\!\rangle_{W+S} = \langle (1/2 - K')\Phi_\ell\,,\, 1\rangle_\Gamma = 0.\end{aligned}$$

According to Linear Algebra, (6.7) holds for all $V_\ell \in \mathcal{S}^1(\mathcal{E}_\ell)$ if and only if it holds for all basis functions $\zeta_k \in \{\zeta_1, \dots, \zeta_N\}$ of $\mathcal{S}^1(\mathcal{E}_\ell)$. With $\mathbf{p} \in \mathbb{R}^N$ from (6.6) and the coefficient vector $\mathbf{x} \in \mathbb{R}^N$ of the ansatz

$$U_\ell = \sum_{j=1}^N \mathbf{x}_j \zeta_j, \tag{6.8}$$

the Galerkin formulation (6.7) is thus equivalent to

$$\sum_{j=1}^N \mathbf{x}_j \langle\!\langle \zeta_j\,, \zeta_k \rangle\!\rangle_{W+S} = \langle\!\langle U_\ell\,, \zeta_k\rangle\!\rangle_{W+S} = \langle (1/2 - K')\Phi_\ell\,,\, \zeta_k\rangle_\Gamma = \sum_{j=1}^N \mathbf{p}_j \langle (1/2 - K')\chi_j\,,\, \zeta_k\rangle_\Gamma \tag{6.9}$$

for all $k = 1, \dots, N$. In the context of Symm's integral equation of Section 5, we have already defined the matrices $\mathbf{K}, \mathbf{M} \in \mathbb{R}^{N\times N}$ by

$$\mathbf{K}_{jk} = \langle K\zeta_k\,,\, \chi_j\rangle_\Gamma \quad \text{and} \quad \mathbf{M}_{jk} = \langle \zeta_k\,,\, \chi_j\rangle_\Gamma,$$

cf. (5.10). The right-hand side of the last equation thus reads

$$\sum_{j=1}^N \mathbf{p}_j \langle (1/2 - K')\chi_j\,,\, \zeta_k\rangle_\Gamma = \frac{1}{2}\sum_{j=1}^N \mathbf{p}_j \langle \chi_j\,,\, \zeta_k\rangle_\Gamma - \sum_{j=1}^N \mathbf{p}_j \langle \chi_j\,,\, K\zeta_k\rangle_\Gamma = \frac{1}{2}\,(\mathbf{M}^T\mathbf{p})_k - (\mathbf{K}^T\mathbf{p})_k.$$

To compute the left-hand side of (6.9), we define matrices $\mathbf{W}, \mathbf{S} \in \mathbb{R}^{N\times N}$ by

$$\mathbf{W}_{kj} = \langle W\zeta_j\,,\, \zeta_k\rangle_\Gamma, \quad \mathbf{S}_{kj} = \Big(\int_\Gamma \zeta_j \, d\Gamma\Big)\Big(\int_\Gamma \zeta_k \, d\Gamma\Big) \quad \text{for all } j,k = 1, \dots, N. \tag{6.10}$$

Then there holds

$$\big((\mathbf{W} + \mathbf{S})\mathbf{x}\big)_k = \sum_{j=1}^N \mathbf{x}_j(\mathbf{W}_{kj} + \mathbf{S}_{kj}) = \langle W U_\ell\,,\, \zeta_k\rangle_\Gamma + \Big(\int_\Gamma U_\ell \, d\Gamma\Big)\Big(\int_\Gamma \zeta_k \, d\Gamma\Big) = \langle\!\langle U_\ell\,, \zeta_k\rangle\!\rangle_{W+S}.$$

Altogether, the Galerkin system (6.7) is equivalently stated by

$$(\mathbf{W} + \mathbf{S})\mathbf{x} = \frac{1}{2}\mathbf{M}^T\mathbf{p} - \mathbf{K}^T\mathbf{p}. \tag{6.11}$$

Note that the matrix $\mathbf{S}$ has rank 1 since it can be written in the form

$$\mathbf{S} = \mathbf{c}\mathbf{c}^T \quad \text{with the column vector} \quad \mathbf{c} \in \mathbb{R}^N \text{ with } \mathbf{c}_j := \int_\Gamma \zeta_j \, d\Gamma. \tag{6.12}$$

Finally, we stress that the matrix $\mathbf{W} + \mathbf{S}$ from (6.11) is symmetric and positive definite since it stems from a scalar product. Consequently, the linear system (6.11) has a unique solution $\mathbf{x} \in \mathbb{R}^N$.

Listing 15. Computation of Data Oscillations for Neumann Data

```
1 function [osc,phih] = computeOscNeumann(coordinates,elements,phi)
```

```
%*** arbitrary quadrature on [-1,1] with exactness n >= 2, e.g., gauss(2)
quad_nodes = [-1 1]/sqrt(3);
quad_weights = [1;1];

%*** the remaining code is independent of the chosen quadrature rule
nE = size(elements,1);
nQ = length(quad_nodes);

%*** build vector of evaluations points as (nQ*nE x 2)-matrix
a = coordinates(elements(:,1),:);
b = coordinates(elements(:,2),:);
sx = reshape(a,2*nE,1)*(1-quad_nodes) + reshape(b,2*nE,1)*(1+quad_nodes);
sx = 0.5*reshape(sx',nQ*nE,2);

%*** phi(sx) usually depends on the normal vector, whence phi takes sx and the
%*** nodes of the respective element to compute the normal
a_sx = reshape(repmat(reshape(a,2*nE,1),1,nQ)',nE*nQ,2);
b_sx = reshape(repmat(reshape(b,2*nE,1),1,nQ)',nE*nQ,2);

%*** perform all necessary evaluations of phi as (nE x nQ)-matrix
phi_sx = reshape(phi(sx,a_sx,b_sx),nQ,nE)';

%*** compute elementwise integral mean of phi
phih = phi_sx*quad_weights*0.5;

%*** compute vector of (squared) element-widths
h = sum((a-b).^2,2);

%*** compute oscillation terms
osc_sx = (phi_sx - repmat(phih,1,nQ)).^2;
osc = 0.5*h.*(osc_sx*quad_weights);
```

6.1. Discretization of Neumann Data and Computation of Corresponding Neumann Data Oscillations (Listing 15). Instead of solving the correct variational form (6.5), we solve

$$\langle\!\langle u_\ell \,,\, v \rangle\!\rangle_{W+S} = \langle (1/2 - K')\Phi_\ell \,,\, v \rangle_\Gamma \quad \text{for all } v \in H^{1/2}(\Gamma) \tag{6.13}$$

with perturbed right-hand side, where we use the approximation $\Phi_\ell \approx \phi$. Analytically, the error between the exact solution $u \in H^{1/2}(\Gamma)$ of (6.5) and the exact solution $u_\ell \in H^{1/2}(\Gamma)$ of the perturbed formulation (6.13) is controlled by

$$|\!|\!| u - u_\ell |\!|\!|_{W+S} \lesssim \| h_\ell^{1/2} (\phi - \Phi_\ell) \|_{L^2(\Gamma)} =: \mathrm{osc}_{N,\ell}, \tag{6.14}$$

see [5]. We now aim for a numerical approximation of the local contributions

$$\mathrm{osc}_{N,\ell}(E_j) := \| h_\ell^{1/2} (\phi - \Phi_\ell) \|_{L^2(E_j)} = \mathrm{length}(E_j)^{1/2} \, \| \phi - \mathbf{p}_j \|_{L^2(E_j)} \quad \text{for all } E_j \in \mathcal{E}_\ell,$$

where —as for the computation of the right-hand side vector $\mathbf{b}$ in Section 6.4— $\mathbf{p}_j$ abbreviates the integral mean

$$\mathbf{p}_j := \frac{1}{\mathrm{length}(E_j)} \int_{E_j} \phi \, d\Gamma = \frac{1}{2} \int_{-1}^{1} \phi \circ \gamma_j \, ds \approx \frac{1}{2} \, \mathrm{quad}_n(\phi \circ \gamma_j) =: \widetilde{\mathbf{p}}_j. \tag{6.15}$$

Here, we use the parametrization $\gamma_j : [-1,1] \to E_j$ from (2.1). Moreover, $\mathrm{quad}_n(\cdot)$ denotes the same quadrature rule as for the computation of the right-hand side vector $\mathbf{b}$ which is exact of order $n \in \mathbb{N}$, i.e., $\mathrm{quad}_n(p) = \int_{-1}^{1} p \, ds$ for all $p \in \mathcal{P}^n[-1,1]$. With this quadrature rule, the local

Neumann oscillations are approximated by

$$\operatorname{osc}_{N,\ell}(E_j)^2 = \operatorname{length}(E_j) \int_{E_j} |\phi - \mathbf{p}_j|^2 \, d\Gamma = \frac{\operatorname{length}(E_j)^2}{2} \int_{-1}^{1} |\phi \circ \gamma_j(s) - \mathbf{p}_j|^2 \, ds$$

$$\approx \frac{\operatorname{length}(E_j)^2}{2} \operatorname{quad}_{\mathrm{n}}\big((\phi \circ \gamma_j - \widetilde{\mathbf{p}}_j)^2\big) =: \widetilde{\operatorname{osc}}_{N,\ell}(E_j)^2. \tag{6.16}$$

With the definition $\widetilde{\operatorname{osc}}_{N,\ell} := \big(\sum_{j=1}^{N} \widetilde{\operatorname{osc}}_{N,\ell}(E_j)^2\big)^{1/2}$, one can then prove that

$$|\operatorname{osc}_{N,\ell} - \widetilde{\operatorname{osc}}_{N,\ell}| = \mathcal{O}(h^{n/2+1}).$$

Since $\operatorname{osc}_{N,\ell} = \mathcal{O}(h^{3/2})$, we should thus choose $n \geq 2$. For our implementation, we use the Gauss quadrature rule with two nodes, which is exact for polynomials of degree $n = 3$. As for the Dirichlet data oscillations, this choice leads to

$$|\operatorname{osc}_{N,\ell} - \widetilde{\operatorname{osc}}_{N,\ell}| = \mathcal{O}(h^{5/2}), \quad \text{whereas} \quad \operatorname{osc}_{N,\ell} = \mathcal{O}(h^{3/2}).$$

The documentation of Listing 15 reads as follows:

- The function takes the given mesh $\mathcal{E}_\ell$ in form of the arrays `coordinates` and `elements` as well as a function handle `phi` for the Neumann data. A call of the function `phi` is done by

  ```
  y = phi(x,a,b)
  ```

  with $(n \times 2)$-arrays `x`, `a`, and `b`. The j-th rows `x`$(j,:)$, `a`$(j,:)$, and `b`$(j,:)$ correspond to a point $x_j \in [a_j, b_j] \subset \mathbb{R}^2$. The entry `y`$(j)$ of the column vector `y` then contains $\phi(x_j)$.
- As stated above, we use the Gauss quadrature with two nodes (Line 3–4).
- If $s_k \in [-1, 1]$ is a quadrature node and $E_j = [a_j, b_j] \in \mathcal{E}_\ell = \{E_1, \dots, E_N\}$ is an element, the function ϕ has to be evaluated at

  $$\gamma_j(s_k) = \frac{1}{2}\big(a_j + b_j + s_k(b_j - a_j)\big) = \frac{1}{2}\big(a_j(1 - s_k) + b_j(1 + s_k)\big).$$

  In Line 11–14, we build the $(2N \times 2)$-array `sx` which contains all necessary evaluation points. Note that the two evaluation points at E_j are stored in `sx`$(2j-1,:)$ and `sx`$(2j,:)$.
- In Line 18–19, we compute the $(2N \times 2)$-arrays `a_sx` and `b_sx` such that, e.g., `a_sx`$(2j-1,:)$ and `a_sx`$(2j,:)$ contain the first node $a_j \in \mathbb{R}^2$ of the boundary element $E_j = [a_j, b_j]$.
- We then evaluate the Neumann data ϕ simultaneously in all evaluation points and we reshape this $(2N \times 1)$-array into an $(N \times 2)$-array `phi_sx` such that `phi_sx`$(j,:)$ contains all ϕ-values related to E_j (Line 22).
- As a next step, we compute the $(N \times 1)$-array `phih` of all integral means along the lines of (6.15), namely `phih`$(j) = \operatorname{quad}_n(\phi \circ \gamma_j)/2$ (Line 25).
- We realize Equation (6.16). Since we are using the same quadrature rule as for the computation of the integral mean, all necessary evaluations of ϕ have already been computed. Therefore, we derive the necessary evaluations of $(\phi - \widetilde{\mathbf{p}}_j)^2$ in Line 31. Multiplication with the quadrature weights and coefficient-wise weighting with $\operatorname{length}(E_j)^2/2$ provides the $(N \times 1)$-array `osc` such that `osc`$(j) \approx \operatorname{length}(E_j)\|\phi - \Phi_\ell\|^2_{L^2(E_j)}$. More precisely, there holds $\operatorname{osc}_{N,\ell}^2 \approx \widetilde{\operatorname{osc}}_{N,\ell}^2 = \sum_{j=1}^{N}$ `osc`(j).

6.2. Computation of Discrete Integral Operator W. The matrix $\mathbf{W} \in \mathbb{R}^{N \times N}_{sym}$ defined in (6.10) is implemented in the programming language C via the MATLAB-MEX-Interface. It is returned by call of

```
W = buildW(coordinates,elements [,eta]);
```

The entries of the matrix $\mathbf{W}$ are computed with the help of Nédélec's formula which is presented in the following identity

$$\langle Wu\,,\, v\rangle_\Gamma = \langle Vu'\,,\, v'\rangle_\Gamma \quad \text{for all } u, v \in H^1(\Gamma). \tag{6.17}$$

Since $\zeta_j' \in \mathcal{P}^0(\mathcal{E}_\ell)$, this gives a direct link between the matrices $\mathbf{W}$ and $\mathbf{V}$, namely, each entry of $\mathbf{W}$ is the weighted sum of four entries of $\mathbf{V}$. The optional parameter `eta` decides whether all entries of $\mathbf{W}$ are computed analytically or if certain double integrals are computed by numerical quadrature. We refer to Section 5.2 for details.

Listing 16. Compute Stabilization for Hypersingular Integral Equation

```
1  function S = buildHypsingStabilization(coordinates,elements)
2  nE = size(elements,1);
3
4  %*** compute local mesh-size
5  h = sqrt(sum((coordinates(elements(:,1),:)-coordinates(elements(:,2),:)).^2,2));
6
7  %*** build vector with entries c(j) = int_Gamma hatfunction(j) ds
8  c = 0.5*accumarray(reshape(elements,2*nE,1),[h;h]);
9
10 %*** build stabilization matrix
11 S = c*c';
```

6.3. Compute Stabilization for Hypersingular Integral Equation (Listing 16). The kernel of the hypersingular integral operator W is the space of constant functions. Since $1 = \sum_{j=1}^N \zeta_j \in \mathcal{S}^1(\mathcal{E}_\ell)$, the corresponding matrix $\mathbf{W}$ defined by $\mathbf{W}_{kj} = \langle W\zeta_j \,,\, \zeta_k\rangle_\Gamma$ for all $j,k \in \{1,\dots,N\}$ cannot be regular. One can prove, however, that it is semi-positive definite. As we have figured out in the introduction, one remedy is to consider the extended bilinear form $\langle\!\langle \cdot\,,\cdot \rangle\!\rangle_{W+S}$ from (6.4). It thus remains to assemble the rank-1-matrix $\mathbf{S} = \mathbf{c}\mathbf{c}^T \in \mathbb{R}^{N\times N}$ from (6.12). For building the vector $\mathbf{c}$ with

$$\mathbf{c}_k := \int_\Gamma \zeta_k \, d\Gamma = \sum_{i=1}^N \int_{E_i} \zeta_k \, d\Gamma,$$

note that the support of ζ_k consists precisely of the elements $E_i \in \mathcal{E}_\ell$ which include $z_k \in \mathcal{K}_\ell$ as a node. The vector $\mathbf{c}$ can be assembled $\mathcal{E}_\ell$-elementwise, and for each element E_i two entries of $\mathbf{c}$ are updated. Moreover, there holds

$$\int_{E_i} \zeta_k \, d\Gamma = \begin{cases} 0, & \text{if } z_k \notin E_i, \\ \operatorname{length}(E_i)/2, & \text{else.} \end{cases}$$

Consequently, the assembly of the vector $\mathbf{c}$ can be done as follows, where `h`(j) contains the element-width $\operatorname{length}(E_j)$.

```
nE = size(elements,1);
h = sqrt(sum((coordinates(elements(:,1),:)-coordinates(elements(:,2),:)).^2,2));
c = zeros(nE,1);
for j = 1:nE
    nodes = elements(j,:);
    c(nodes) = c(nodes) + 0.5*h(j);
end
```

For the final implementation of **`buildHypsingStabilization`** in Listing 16, the `for`-loop is eliminated by use of `accumarray`:

- The function takes the mesh $\mathcal{E}_\ell$ described by the arrays `coordinates` and `elements`.
- We compute the vector of all element-widths (Line 5).
- The former `for`-loop is written in compact form (Line 8).
- Finally, the function builds and returns the stabilization matrix $\mathbf{S}$ (Line 11).

Listing 17. Build Right-Hand Side for Hypersingular Integral Equation

```
function b = buildHypsingRHS(coordinates,elements,phih)
%*** compute DLP-matrix for P0 x S1
K = buildK(coordinates,elements);

%*** compute mass-type matrix for P0 x S1
M = buildM(coordinates,elements);

%*** build right-hand side vector
b =  (phih'*M*0.5 - phih'*K)';
```

6.4. Build Right-Hand Side for Hypersingular Integral Equation (Listing 17). With the representation

$$\Phi_\ell = \sum_{j=1}^{N} \mathbf{p}_j \chi_j$$

and the transposed matrices of $\mathbf{K}$ and $\mathbf{M}$, the right-hand side vector for (6.11) reads

$$\mathbf{b} := \frac{1}{2}\mathbf{M}^T\mathbf{p} - \mathbf{K}^T\mathbf{p} = \Big(\frac{1}{2}\mathbf{p}^T\mathbf{M} - \mathbf{p}^T\mathbf{K}\Big)^T, \tag{6.18}$$

where we identify the vector $\mathbf{p} \in \mathbb{R}^N$ with a matrix $\mathbf{p} \in \mathbb{R}^{N\times 1}$.

The documentation of Listing 17 reads as follows:

- The function takes as input the given mesh $\mathcal{E}_\ell$ in form of the arrays `coordinates` and `elements` as well as a columnvector $\mathbf{p} =$ `phih` with the $\mathcal{T}_\ell$-elementwise values of Φ_ℓ.
- We build $\mathbf{K}$ (Line 3) and $\mathbf{M}$ (Line 6), cf. Section 5.3 above.
- Finally (Line 9), the function computes and returns the vector $\mathbf{b}$ as described in (6.18).

LISTING 18. Reliable Error Bound for $\|\!|u - u_\ell|\!\|_{W+S}$

```
function err = computeErrDirichlet(coordinates,elements,g,uD)
%*** compute midpoints of all elements
midpoints = 0.5*( coordinates(elements(:,1),:) + coordinates(elements(:,2),:) );

%*** compute p = (uD - uDh) at element midpoints
p_midpoints = uD(midpoints) - 0.5*sum(g(elements),2);

%*** compute p = (uD - uDh) at all nodes
p_nodes = uD(coordinates) - g;

%*** evaluate derivative p' at all elements (left,midpoint,right)
p_prime = [p_nodes(elements) p_midpoints] * [-3 -1 1 ; -1 1 3 ; 4 0 -4]*0.5;

%*** compute Dirichlet error simultaneously for all elements
err = 2*p_prime.^2*[1;4;1]/3;
```

6.5. Computation of Reliable Error Bound for $\|\!|u-U_\ell|\!\|_{W+S}$ (Listing 18). We assume that the exact Dirichlet data satisfy additional regularity $u \in H^1(\Gamma)$. Let $U_\ell^* \in \mathcal{S}^1(\mathcal{E}_\ell)$ be the (only theoretically computed) Galerkin solution with respect to the non-perturbed right-hand side $(1/2 - K')\phi$ instead of $(1/2 - K')\Phi_\ell$. Moreover, let I_ℓ denote the nodal interpolation operator onto $\mathcal{S}^1(\mathcal{E}_\ell)$. With the technique from [16, 5], we obtain

$$\|\!|u - U_\ell^*|\!\|_{W+S} \le \|\!|u - I_\ell u|\!\|_{W+S} \lesssim \|h_\ell^{1/2}(u - I_\ell u)'\|_{L^2(\Gamma)} \le \|h_\ell^{1/2}(u - U_\ell)'\|_{L^2(\Gamma)}$$

as well as

$$\|\!|U_\ell^* - U_\ell|\!\|_{W+S} \lesssim \mathrm{osc}_{N,\ell},$$

where $\operatorname{osc}_{N,\ell}$ denotes the Neumann data oscillations from Section 6.1. We therefore obtain

$$\begin{aligned}\|\!|u - U_\ell|\!\|_{W+S} &\le \|\!|u - U_\ell^*|\!\|_{W+S} + \|\!|U_\ell^* - U_\ell|\!\|_{W+S} \\ &\lesssim \|h_\ell^{1/2}(u - U_\ell)'\|_{L^2(\Gamma)} + \operatorname{osc}_{N,\ell} =: \operatorname{err}_{D,\ell} + \operatorname{osc}_{N,\ell}.\end{aligned}$$

For the numerical realization of

$$\operatorname{err}_{D,\ell} = \Big(\sum_{j=1}^N \operatorname{err}_{D,\ell}(E_j)^2\Big)^{1/2}, \quad \text{where} \quad \operatorname{err}_{D,\ell}(E_j)^2 = \operatorname{length}(E_j)\,\|(u - U_\ell)'\|_{L^2(E_j)}^2,$$

we use the same ideas as for the Dirichlet data oscillations in Section 5.1.1, where

$$\operatorname{err}_{D,\ell}(E_j)^2 = 2\int_{-1}^1 \big((u - U_\ell)\circ\gamma_j\big)'(s)^2\,ds \approx \operatorname{quad}_2\big((p_j')^2\big) =: \widetilde{\operatorname{err}}_{D,\ell}(E_j). \tag{6.19}$$

Here, $p_j \in \mathcal{P}^2[-1,1]$ is the unique polynomial with $p_j(-1) = v(a_j)$, $p_j(1) = v(b_j)$, and $p_j(0) = v(m_j)$, where $v = u - U_\ell$ as well as $E_j = [a_j, b_j]$ and $m_j = (a_j + b_j)/2$. Recall that

$$\begin{pmatrix} p_j'(-1) \\ p_j'(0) \\ p_j'(+1)\end{pmatrix} = \begin{pmatrix} -3/2 & +2 & -1/2 \\ -1/2 & 0 & +1/2 \\ +1/2 & -2 & +3/2\end{pmatrix}\begin{pmatrix} v(a_j) \\ v(m_j) \\ v(b_j)\end{pmatrix} = \begin{pmatrix} -3/2 & -1/2 & +2 \\ -1/2 & +1/2 & 0 \\ +1/2 & +3/2 & -2\end{pmatrix}\begin{pmatrix} v(a_j) \\ v(b_j) \\ v(m_j)\end{pmatrix}.$$

As we are at last targeted on vectorization, we write the linear system row-wise as

$$\big(p_j'(-1), p_j'(0), p_j'(+1)\big) = \big(v(a_j), v(b_j), v(m_j)\big)\begin{pmatrix} -3/2 & -1/2 & +1/2 \\ -1/2 & +1/2 & +3/2 \\ +2 & 0 & -2\end{pmatrix}. \tag{6.20}$$

For the numerical quadrature, we use a Newton-Côtes formula with three nodes $s_k \in \{-1, 0, +1\}$ and corresponding weights $\omega_k = \{1/3, 4/3, 1/3\}$. The documentation of Listing 18 now reads as follows:

- The function takes the mesh $\mathcal{E}_\ell$ in terms of `coordinates` and `elements` as well as the nodal vector $\mathbf{g} \in \mathbb{R}^N$ of $U_\ell = \sum_{j=1}^N \mathbf{g}_j \zeta_j$ and the function handle `uD` for the exact solution u (Line 1).
- We first compute all element midpoints (Line 3) and evaluate the solution $u - U_\ell$ at all midpoints (Line 6) and all nodes (Line 9).
- Using (6.20), we provide all necessary evaluations of $p_j'(s_k)$ in form of the $(N \times 3)$-array `p_prime` (Line 12).
- Finally, Line 15 realizes (6.19), and the function returns the column vector `err`, where $\texttt{err}(j) = \widetilde{\operatorname{err}}_{D,\ell}(E_j)^2$. In particular, there holds $\operatorname{err}_{D,\ell} \approx \widetilde{\operatorname{err}}_{D,\ell} := \big(\sum_{j=1}^N \texttt{err}(j)\big)^{1/2}$.

Remark 6.1. *In academic experiments, the exact solution u is usually known and has certain regularity $u \in H^1(\Gamma)$ which only depends on the geometry of Γ. As explained before, there holds*

$$\|\!|u - U_\ell|\!\|_{W+S} \lesssim \operatorname{err}_{D,\ell} + \operatorname{osc}_{N,\ell},$$

so that we can control the error reliably. Moreover, the convergence $\operatorname{err}_{D,\ell} \to 0$ as $\ell \to \infty$ might indicate that there are no major bugs in the implementation — since we compare the Galerkin solution with the exact solution. □

6.6. Computation of $(h - h/2)$-Based A Posteriori Error Estimators. In this section, we discuss the implementation of four error estimators which are introduced and analyzed in [16]. Let $\widehat{\mathcal{E}}_\ell = \{e_1, \dots, e_{2N}\}$ be the uniform refinement of the mesh $\mathcal{E}_\ell$. Let $U_\ell \in \mathcal{S}^1(\mathcal{E}_\ell)$ and $\widehat{U}_\ell \in \mathcal{S}^1(\widehat{\mathcal{E}}_\ell)$ be the Galerkin solutions of (6.7) with respect to $\mathcal{E}_\ell$ and $\widehat{\mathcal{E}}_\ell$ and the same approximate Neumann data Φ_ℓ, i.e. there holds

$$\langle\!\langle U_\ell\,, V_\ell\rangle\!\rangle_{W+S} = \langle (1/2 - K')\Phi_\ell\,,\, V_\ell\rangle_\Gamma \quad \text{for all } V_\ell \in \mathcal{S}^1(\mathcal{E}_\ell) \tag{6.21}$$

and

$$\langle\!\langle \widehat{U}_\ell \,,\, \widehat{V}_\ell \rangle\!\rangle_{W+S} = \langle (1/2 - K')\Phi_\ell \,,\, \widehat{V}_\ell \rangle_\Gamma \quad \text{for all } \widehat{V}_\ell \in \mathcal{S}^1(\widehat{\mathcal{E}}_\ell). \tag{6.22}$$

As for Symm's integral equation, one can expect

$$|\!|\!| u_\ell - U_\ell |\!|\!|_{W+S} \approx |\!|\!| \widehat{U}_\ell - U_\ell |\!|\!|_{W+S} = |\!|\!| \widehat{U}_\ell - U_\ell |\!|\!|_W =: \eta_\ell, \tag{6.23}$$

where $u_\ell \in H^{1/2}(\Gamma)$ denotes the exact solution of

$$\langle\!\langle u_\ell \,,\, v \rangle\!\rangle_{W+S} = \langle (1/2 - K')\Phi_\ell \,,\, v \rangle_\Gamma \quad \text{for all } v \in H^{1/2}(\Gamma). \tag{6.24}$$

According to (6.14), (6.23), and the triangle inequality, there holds

$$|\!|\!| u - U_\ell |\!|\!|_{W+S} \le |\!|\!| u - u_\ell |\!|\!|_{W+S} + |\!|\!| u_\ell - U_\ell |\!|\!|_{W+S} \lesssim \operatorname{osc}_{N,\ell} + \eta_\ell. \tag{6.25}$$

Clearly, the Galerkin solution $\widehat{U}_\ell$ with respect to $\mathcal{S}^1(\widehat{\mathcal{E}}_\ell)$ is more accurate than U_ℓ. Consequently, any algorithm will return $\widehat{U}_\ell$ instead of U_ℓ if $\widehat{U}_\ell$ has been computed. From this point of view U_ℓ becomes a side result and leads to unnecessary computational effort. Similar to Section 5, one can prove that one may replace U_ℓ by a cheap (but appropriate) postprocessing $I_\ell \widehat{U}_\ell$ of $\widehat{U}_\ell$. This leads to some error estimator

$$\eta_\ell \sim |\!|\!| \widehat{U}_\ell - I_\ell \widehat{U}_\ell |\!|\!|_{W+S} =: \widetilde{\eta}_\ell \tag{6.26}$$

which always stays proportional to η_ℓ, indicated by $\eta_\ell \sim \widetilde{\eta}_\ell$, cf. [16]. To be more precise, I_ℓ denotes the nodal interpolation operator on $\mathcal{S}^1(\mathcal{E}_\ell)$, which is given by

$$I_\ell U_\ell := \sum_{z \in \mathcal{K}_\ell} U_\ell(z) \zeta_z,$$

where $\mathcal{K}_\ell$ denotes the set of all nodes of $\mathcal{E}_\ell$ and where ζ_z denotes the hat-function associated with some node $z \in \mathcal{K}_\ell$.

As a matter of fact, the error estimators η_ℓ and $\widetilde{\eta}_\ell$ do not provide any information about the local errors, i.e., the error $|\!|\!| u_\ell - U_\ell |\!|\!|_{W+S}$ related to some element $E_i \in \mathcal{E}_\ell$. This is different for the error estimators μ_ℓ and $\widetilde{\mu}_\ell$ discussed in the following. For instance, one can prove that

$$\eta_\ell \sim \mu_\ell := \|h_\ell^{1/2} (\widehat{U}_\ell - U_\ell)'\|_{L^2(\Gamma)} = \Big(\sum_{i=1}^N \operatorname{length}(E_i) \|(\widehat{U}_\ell - U_\ell)'\|_{L^2(E_i)}^2 \Big)^{1/2}. \tag{6.27}$$

The local contributions

$$\mu_\ell(E_i) := \operatorname{length}(E_i)^{1/2} \|(\widehat{U}_\ell - U_\ell)'\|_{L^2(E_i)} \quad \text{for all } E_i \in \mathcal{E}_\ell \tag{6.28}$$

give some measure for the error on E_i.

As the computation of the error estimator η_ℓ, the computation of μ_ℓ requires two Galerkin solutions U_ℓ and $\widehat{U}_\ell$. As before, the computation of the coarse-mesh solution U_ℓ can be avoided by use of the nodal interpolant $I_\ell \widehat{U}_\ell$. One can mathematically prove that

$$\eta_\ell \sim \widetilde{\mu}_\ell := \|h_\ell^{1/2} (\widehat{U}_\ell - I_\ell \widehat{U}_\ell)'\|_{L^2(\Gamma)}. \tag{6.29}$$

In the following subsections, we first discuss the computation of the *global* error estimators η_ℓ and $\widetilde{\eta}_\ell$ from (6.23) and (6.26). Then, we give an implementation of the *local* error estimators μ_ℓ and $\widetilde{\mu}_\ell$ from (6.27) and (6.29), where our functions return the local contributions, see e.g. (6.28), to steer an adaptive mesh-refinement.

Remark 6.2. *If we plot the error estimators η_ℓ, $\widetilde{\eta}_\ell$, μ_ℓ and $\widetilde{\mu}_\ell$ over the number of elements, one can mathematically predict that the corresponding curves, for a sequence of arbitrarily refined meshes, are parallel. In mathematical terms, this reads*

$$\eta_\ell \le \widetilde{\eta}_\ell \lesssim \widetilde{\mu}_\ell \le \mu_\ell \lesssim \eta_\ell, \tag{6.30}$$

cf. [16]. *Empirically, one observes a very good coincidence of η_ℓ and $\widetilde{\eta}_\ell$ in the sense that the corresponding curves almost coincide. The same is observed for the curves of μ_ℓ and $\widetilde{\mu}_\ell$.* □

Remark 6.3. *Mathematically, the error estimate* (6.23) *respectively* (6.25) *involves the so-called saturation assumption: Assume that we could compute the Galerkin solutions* U_ℓ^* *and* $\widehat{U}_\ell^*$ *with respect to* $\mathcal{E}_\ell$ *and* $\widehat{\mathcal{E}}_\ell$ *for the non-perturbed variational formulation* (6.5)*, i.e., we formally use the exact Neumann data* ϕ *instead of the interpolated data* Φ_ℓ *— although the right-hand side is, in practice, non-computable because of* $K'\phi$*. Then, the saturation assumption states that*

$$|||u - \widehat{U}_\ell^*|||_{W+S} \le q\, |||u - U_\ell^*|||_{W+S} \tag{6.31}$$

with some uniform and ℓ*-independent constant* $q \in (0,1)$*. Put differently, uniform mesh-refinement leads to a uniform improvement of the discretization error. Provided* (6.31)*, one can prove that*

$$\eta_\ell \le |||u_\ell - U_\ell|||_{W+S} \le (1-q^2)^{-1/2}\, \eta_\ell. \tag{6.32}$$

We stress that this assumption is somewhat natural and can, for instance, be proven for the finite element method [14, 17]*. For the boundary element method, however,* (6.31) *still remains open.*

Finally, one can prove that (6.31) *is sufficient and in some sense even necessary to guarantee* (6.25)*.* □

Remark 6.4. *In academic experiments, the exact solution* u *of the hypersingular integral equation is usually known and has certain regularity* $u \in H^1(\Gamma)$ *which only depends on the geometry of* Γ*. In this case, one can experimentally verify the saturation assumption as follows: In Section 6.5, we derived*

$$|||u - U_\ell|||_W \lesssim \mathrm{err}_{D,\ell} + \mathrm{osc}_{N,\ell}.$$

If the right-hand side has the same convergence behaviour as the error estimator $\eta_\ell + \mathrm{osc}_{N,\ell}$*, this proves empirically*

$$|||u - U_\ell|||_W \lesssim \eta_\ell + \mathrm{osc}_{N,\ell}$$

and confirms the saturation assumption. □

LISTING 19. Computation of Estimator η_ℓ

```
function est = computeEstHypEta(elements_fine,elements_coarse,father2son,...
                                                        W_fine,x_fine,x_coarse)
nC = length(x_coarse);

%*** build index field k = idx(j) such that j-th node of coarse mesh coincides
%*** with k-th node of fine mesh
idx = zeros(nC,1);
idx(elements_coarse) = [ elements_fine(father2son(:,1),1), ...
                         elements_fine(father2son(:,2),2) ];

%*** build index field k = mid(j) such that midpoint of j-th element of coarse
%*** mesh is k-th node of fine mesh
mid = elements_fine(father2son(:,1),2);

%*** compute coefficient vector of (u_fine - u_coarse) w.r.t. fine mesh
x_fine(idx) = x_fine(idx) - x_coarse;
x_fine(mid) = x_fine(mid) - 0.5*sum(x_coarse(elements_coarse),2);

%*** compute energy ||| u_fine - u_coarse |||^2
est = x_fine'*(W_fine*x_fine);
```

6.6.1. Computation of Error Estimator η_ℓ (Listing 19). In this section, we aim to compute the error estimator $\eta_\ell = \|\widehat{U}_\ell - U_\ell\|_{W+S}$ from (6.23). Let $\widehat{\zeta}_j$ denote the hat-function associated with some fine-mesh node $z_j \in \widehat{\mathcal{K}}_\ell$. Let $\mathbf{x} \in \mathbb{R}^N$ and $\widehat{\mathbf{x}} \in \mathbb{R}^{2N}$ be the coefficient vectors of U_ℓ and $\widehat{U}_\ell$ with respect to the canonical bases of $\mathcal{S}^1(\mathcal{E}_\ell)$ and $\mathcal{S}^1(\widehat{\mathcal{E}}_\ell)$, i.e.

$$U_\ell = \sum_{j=1}^{N} \mathbf{x}_j \zeta_j \quad \text{and} \quad \widehat{U}_\ell = \sum_{j=1}^{2N} \widehat{\mathbf{x}}_j \widehat{\zeta}_j.$$

Similar to Section 5.6.1, there holds $\mathcal{S}^1(\mathcal{E}_\ell) \subset \mathcal{S}^1(\widehat{\mathcal{E}}_\ell)$ which provides a unique vector $\widehat{\mathbf{y}} \in \mathbb{R}^{2N}$ such that

$$U_\ell = \sum_{j=1}^{2N} \widehat{\mathbf{y}}_j \widehat{\zeta}_j.$$

With the vectors $\widehat{\mathbf{x}}, \widehat{\mathbf{y}} \in \mathbb{R}^{2N}$, there holds

$$\begin{aligned}\eta_\ell^2 = \|\widehat{U}_\ell - U_\ell\|_{W+S}^2 = \langle\!\langle \widehat{U}_\ell - U_\ell \,, \widehat{U}_\ell - U_\ell \rangle\!\rangle_{W+S} &= \sum_{j,k=1}^{2N} (\widehat{\mathbf{x}}_j - \widehat{\mathbf{y}}_j)(\widehat{\mathbf{x}}_k - \widehat{\mathbf{y}}_k) \langle\!\langle \widehat{\zeta}_j \,, \widehat{\zeta}_k \rangle\!\rangle_{W+S} \\ &= (\widehat{\mathbf{x}} - \widehat{\mathbf{y}}) \cdot (\widehat{\mathbf{W}} + \widehat{\mathbf{S}})(\widehat{\mathbf{x}} - \widehat{\mathbf{y}}),\end{aligned}$$

where $\widehat{\mathbf{W}}$ denotes the matrix of the hypersingular integral operator and $\widehat{\mathbf{S}}$ the matrix of the stabilization term contributions (6.10) with respect to the fine mesh, cf. Section 6. The documentation of Listing 19 now reads as follows:

- The function takes the coefficient vectors $\mathbf{x} \in \mathbb{R}^N$ and $\widehat{\mathbf{x}} \in \mathbb{R}^{2N}$ of the Galerkin solutions U_ℓ and $\widehat{U}_\ell$ as well as the sum $\widehat{\mathbf{W}} + \widehat{\mathbf{S}}$ of the hypersingular operator matrix $\widehat{\mathbf{W}}$ and the stabilization term matrix $\widehat{\mathbf{S}}$ for the fine mesh $\widehat{\mathcal{E}}_\ell$ stored in `W_fine`. Besides this, the function takes the coarse mesh described by the $(N \times 2)$-array `elements_coarse` and the fine mesh described by the $(2N \times 2)$-array `elements_fine`. Finally, the $(N \times 2)$-array `father2son` links the indices of elements $E_i \in \mathcal{E}_\ell$ with the indices of its sons $e_j, e_k \in \widehat{\mathcal{E}}_\ell$ in the sense that `father2son(`i`,:) = [`j,k`]` for $E_i = e_j \cup e_k$.
- We build an array `k = idx(i)` such that the i-th node of the coarse mesh coincides with the k-th node of the fine mesh (Line 7–9).
- Furthermore, we build an array `k = mid(j)` such that the midpoint of the j-th element of the coarse mesh is the k-th node of the fine mesh (Line 13).
- Then we overwrite successively the vector $\widehat{\mathbf{x}}$ by the coefficient vector $\widehat{\mathbf{x}} - \widehat{\mathbf{y}}$ of $\widehat{U}_\ell - U_\ell$. We first calculate this difference for any node belonging to $\mathcal{K}_\ell$ (Line 16) and in a next step for any node occurring in $\widehat{\mathcal{K}}_\ell \backslash \mathcal{K}_\ell$ by interpolating the coarse vector (Line 17).
- Finally, the function returns $\eta_\ell^2 = \|\widehat{U}_\ell - U_\ell\|_{W+S}^2$ (Line 20).

LISTING 20. Computation of Estimator $\widetilde{\eta}_\ell$

```
1  function est = computeEstHypEtaTilde(elements_fine,elements_coarse,...
2                                                    father2son,W_fine,x_fine)
3  nC = max(elements_coarse(:));
4
5  %*** build index field k = idx(j) such that j-th node of coarse mesh coincides
6  %*** with k-th node of fine mesh
7  idx = zeros(nC,1);
8  idx(elements_coarse) = [ elements_fine(father2son(:,1),1), ...
9                           elements_fine(father2son(:,2),2) ];
10
11 %*** build index field k = mid(j) such that midpoint of j-th element of coarse
12 %*** mesh is k-th node of fine mesh
13 mid = elements_fine(father2son(:,1),2);
```

```
14
15 %*** build index field [i j] = e2n(k) such that fine-mesh nodes zi and zj are
16 %*** the nodes of the coarse-mesh element Ek
17 e2n = [ elements_fine(father2son(:,1),1) elements_fine(father2son(:,2),2) ];
18
19 %*** compute coefficient vector of (1 - I_coarse)*u_fine w.r.t. fine mesh
20 x_fine(mid) = x_fine(mid) - 0.5*sum(x_fine(e2n),2);
21 x_fine(idx) = 0;
22
23 %*** compute energy ||| (1 - I_coarse)*u_fine |||^2
24 est = x_fine'*(W_fine*x_fine);
```

6.6.2. Computation of Error Estimator $\widetilde{\eta}_\ell$ (Listing 20). In this section, we aim to compute the error estimator $\widetilde{\eta}_\ell$ which is defined by

$$\widetilde{\eta}_\ell := \|\widehat{U}_\ell - I_\ell \widehat{U}_\ell\|_{W+S}.$$

We adopt the notation of Section 6.6.1 for the computation of η_ℓ, namely $\widehat{\mathbf{x}} \in \mathbb{R}^{2N}$ with

$$\widehat{U}_\ell = \sum_{j=1}^{2N} \widehat{\mathbf{x}}_j \widehat{\zeta}_j.$$

Let $z_i \in \widehat{\mathcal{K}}_\ell \backslash \mathcal{K}_\ell$. Then, there are two elements $e_j, e_k \in \widehat{\mathcal{E}}_\ell$ being the sons of $E_i \in \mathcal{E}_\ell$, i.e. $E_i = e_j \cup e_k$, which share z_i as a common node. Since $I_\ell \widehat{U}_\ell$ restricted to some element $E_i = e_j \cup e_k$ is affine, there holds

$$I_\ell \widehat{U}_\ell(z_i) = \frac{1}{2}\Big(I_\ell \widehat{U}_\ell(z_j) + I_\ell \widehat{U}_\ell(z_k)\Big) = \frac{1}{2}\Big(\widehat{U}_\ell(z_j) + \widehat{U}_\ell(z_k)\Big), \tag{6.33}$$

where $z_j, z_k \in \mathcal{K}_\ell$ denote the outer nodes of the elements e_j, e_k. On the other hand, there holds $I_\ell \widehat{U}_\ell(z_i) = \widehat{U}_\ell(z_i)$ provided that $z_i \in \mathcal{K}_\ell$. Altogether, representing $I_\ell \widehat{U}_\ell \in \mathcal{S}^1(\mathcal{E}_\ell)$ with respect to the fine-mesh $\widehat{\mathcal{E}}_\ell$, we obtain

$$I_\ell \widehat{U}_\ell = \sum_{n=1}^{2N} \widehat{\mathbf{z}}_n \widehat{\zeta}_n, \tag{6.34}$$

where $\widehat{\mathbf{z}} \in \mathbb{R}^{2N}$ denotes the coefficient vector. As in Section 6.6.1, there holds

$$\widetilde{\eta}_\ell^2 = \|\widehat{U}_\ell - I_\ell \widehat{U}_\ell\|_{W+S}^2 = (\widehat{\mathbf{x}} - \widehat{\mathbf{z}}) \cdot (\widehat{\mathbf{W}} + \widehat{\mathbf{S}})(\widehat{\mathbf{x}} - \widehat{\mathbf{z}}).$$

Therefore, the documentation of Listing 20 reads as follows:

- The function takes the coefficient vector $\widehat{\mathbf{x}} \in \mathbb{R}^{2N}$ of the Galerkin solutions $\widehat{U}_\ell$ as well as the sum $\widehat{\mathbf{W}} + \widehat{\mathbf{S}}$ of the hypersingular operator matrix $\widehat{\mathbf{W}}$ and the stabilization term matrix $\widehat{\mathbf{S}}$ for the fine mesh $\widehat{\mathcal{E}}_\ell$ stored in `W_fine`. Besides this, the function takes the coarse mesh described by the $(N \times 2)$-array `elements_coarse` and the fine mesh described by the $(2N \times 2)$-array `elements_fine`. Moreover, the link between $\mathcal{E}_\ell$ and $\widehat{\mathcal{E}}_\ell$ is provided by means of `father2son`.
- We first build an array `k = idx(i)` such that the i-th node of the coarse mesh coincides with the k-th node of the fine mesh (Line 7–9).
- Furthermore, we build an array `k = mid(j)` such that the midpoint of the j-th element of the coarse mesh is the k-th node of the fine mesh (Line 13).
- Next we build an array `[i j] = e2n(k)` such that the fine-mesh nodes z_i and z_j are the nodes of the coarse mesh elements E_k (Line 17).
- We successively overwrite $\widehat{\mathbf{x}}$ by the coefficient vector $\widehat{\mathbf{x}} - \widehat{\mathbf{z}} \in \mathbb{R}^{2N}$ of $\widehat{U}_\ell - I_\ell \widehat{U}_\ell$ (Line 20–21).
- Finally, the function returns $\widetilde{\eta}_\ell^2 = \|\widehat{U}_\ell - I_\ell \widehat{U}_\ell\|_{W+S}^2$ (Line 24).

LISTING 21. Computation of Estimator μ_ℓ

```
function ind = computeEstHypMu(elements_fine,elements_coarse,father2son,...
                                                              x_fine,x_coarse)
nC = length(x_coarse);

%*** build index field k = idx(j) such that j-th node of coarse mesh coincides
%*** with k-th node of fine mesh
idx = zeros(nC,1);
idx(elements_coarse) = [ elements_fine(father2son(:,1),1), ...
                         elements_fine(father2son(:,2),2) ];

%*** build index field k = mid(j) such that midpoint of j-th element of coarse
%*** mesh is k-th node of fine mesh
mid = elements_fine(father2son(:,1),2);

%*** compute coefficient vector of (u_fine - u_coarse) w.r.t. fine mesh
x_fine(idx) = x_fine(idx) - x_coarse;
x_fine(mid) = x_fine(mid) - 0.5*sum(x_coarse(elements_coarse),2);

%*** compute h^2*|(u_fine - u_coarse)'|^2 for all fine-mesh elements
%*** where h denotes the diameters of the fine-mesh elements
grad = (x_fine(elements_fine)*[-1;1]).^2;

%*** compute (squared) indicators w.r.t. coarse mesh as described above
ind = 2*( grad(father2son(:,1)) + grad(father2son(:,2)) );
```

6.6.3. Computation of Error Estimator μ_ℓ (Listing 21). In this section, we discuss the implementation of

$$\mu_\ell^2 := \sum_{i=0}^{N} \mu_\ell(E_i)^2, \quad \text{where} \quad \mu_\ell(E_i)^2 := \operatorname{length}(E_i)\|(\widehat{U}_\ell - U_\ell)'\|_{L^2(E_i)}^2.$$

Actually we calculate the squared entries $\mu_\ell(E_i)^2$ for all $E_i \in \mathcal{E}_\ell$.

We adopt the notation of Section 6.6.1, namely $\widehat{\mathbf{x}}, \widehat{\mathbf{y}} \in \mathbb{R}^{2N}$ with

$$\widehat{U}_\ell = \sum_{j=1}^{2N} \widehat{\mathbf{x}}_j \widehat{\zeta}_j \quad \text{and} \quad U_\ell = \sum_{j=1}^{2N} \widehat{\mathbf{y}}_j \widehat{\zeta}_j.$$

For fixed $E_i \in \mathcal{E}_\ell$ and sons $e_j, e_k \in \widehat{\mathcal{E}}_\ell$ with $E_i = e_j \cup e_k$, we obtain

$$\|(\widehat{U}_\ell - U_\ell)'\|_{L^2(E_i)}^2 = \int_{E_i} \left|(\widehat{U}_\ell - U_\ell)'\right|^2 d\Gamma = \int_{e_j} \left|(\widehat{U}_\ell - U_\ell)'\right|^2 d\Gamma + \int_{e_k} \left|(\widehat{U}_\ell - U_\ell)'\right|^2 d\Gamma.$$

As $(\widehat{U}_\ell - U_\ell) \in \mathcal{S}^1(\widehat{\mathcal{E}}_\ell)$ is piecewise affine, its arc-length derivative $(\widehat{U}_\ell - U_\ell)' \in \mathcal{P}^0(\widehat{\mathcal{E}}_\ell)$ is piecewise constant. Consequently, the above formula reduces to

$$\begin{aligned}\|(\widehat{U}_\ell - U_\ell)'\|_{L^2(E_i)}^2 &= \operatorname{length}(e_j)\left|(\widehat{U}_\ell - U_\ell)'|_{e_j}\right|^2 + \operatorname{length}(e_k)\left|(\widehat{U}_\ell - U_\ell)'|_{e_k}\right|^2 \\ &= \frac{\operatorname{length}(E_i)}{2}\Big(\left|(\widehat{U}_\ell - U_\ell)'|_{e_j}\right|^2 + \left|(\widehat{U}_\ell - U_\ell)'|_{e_k}\right|^2\Big).\end{aligned}$$

With $e_j = [z_{j_1}, z_{j_2}] \in \widehat{\mathcal{E}}_\ell$, we obtain

$$\left|(\widehat{U}_\ell - U_\ell)'|_{e_j}\right|^2 = \frac{\left|(\widehat{U}_\ell - U_\ell)(z_{j_2}) - (\widehat{U}_\ell - U_\ell)(z_{j_1})\right|^2}{\operatorname{length}(e_j)^2}.$$

This implies

$$\mu_\ell(E_i)^2 = 2\Big(\big|(\widehat{U}_\ell - U_\ell)(z_{j_2}) - (\widehat{U}_\ell - U_\ell)(z_{j_1})\big|^2 + \big|(\widehat{U}_\ell - U_\ell)(z_{k_2}) - (\widehat{U}_\ell - U_\ell)(z_{k_1})\big|^2 \Big) \tag{6.35}$$

Altogether, the documentation of Listing 21 reads as follows:

- As input arguments, the function takes the mesh $\mathcal{E}_\ell$ represented by the $(N \times 2)$-array `elements_coarse`, the mesh $\widehat{\mathcal{E}}_\ell$ represented by the $(2N \times 2)$-array `elements_fine`, the link between $\mathcal{E}_\ell$ and $\widehat{\mathcal{E}}_\ell$, and the coefficient vectors $\mathbf{x} \in \mathbb{R}^N$ and $\widehat{\mathbf{x}} \in \mathbb{R}^{2N}$ of the Galerkin solutions U_ℓ and $\widehat{U}_\ell$ (Line 1–2).
- We overwrite the vector $\widehat{\mathbf{x}}$ by coefficient vector $\widehat{\mathbf{x}} - \widehat{\mathbf{y}}$ of $\widehat{U}_\ell - U_\ell$ in exactly the same way as we did in Section 6.6.1 for the error estimator η_ℓ (Line 7–17).
- Next (Line 21), we compute the coefficient vector of the squared arc-length derivative of $\widehat{U}_\ell - U_\ell$ multiplied by the diameter of the fine-mesh elements to avoid needless computations.
- Finally (Line 24), the function realizes (6.35) and returns the vector

$$\mathbf{v} := (\mu_\ell(E_1)^2, \dots, \mu_\ell(E_N)^2) \in \mathbb{R}^N$$

so that $\mu_\ell = \big(\sum_{i=1}^N \mathbf{v}_i\big)^{1/2}$.

LISTING 22. Computation of Estimator $\widetilde{\mu}_\ell$

```
1  function ind = computeEstHypMuTilde(elements_fine,elements_coarse,...
2                                                          father2son,x_fine)
3  nC = max(elements_coarse(:));
4
5  %*** build index field k = idx(j) such that j-th node of coarse mesh coincides
6  %*** with k-th node of fine mesh
7  idx = zeros(nC,1);
8  idx(elements_coarse) = [ elements_fine(father2son(:,1),1), ...
9                           elements_fine(father2son(:,2),2) ];
10
11 %*** build index field k = mid(j) such that midpoint of j-th element of coarse
12 %*** mesh is k-th node of fine mesh
13 mid = elements_fine(father2son(:,1),2);
14
15 %*** build index field [i j] = e2n(k) such that fine-mesh nodes zi and zj are
16 %*** the nodes of the coarse-mesh element Ek
17 e2n = [ elements_fine(father2son(:,1),1) elements_fine(father2son(:,2),2) ];
18
19 %*** compute coefficient vector of (1 - I_coarse)*u_fine w.r.t. fine mesh
20 x_fine(mid) = x_fine(mid) - 0.5*sum(x_fine(e2n),2);
21 x_fine(idx) = 0;
22
23 %*** compute h^2*| ((1 - I_coarse)*u_fine)' |^2 for all fine-mesh elements
24 %*** where h denotes the diameters of the fine-mesh elements
25 grad = (x_fine(elements_fine)*[-1;1]).^2;
26
27 %*** compute (squared) indicators w.r.t. coarse mesh as described above
28 ind = 2*( grad(father2son(:,1)) + grad(father2son(:,2)) );
```

6.6.4. Computation of Error Estimator $\widetilde{\mu}_\ell$ (Listing 22). In this section, we finally aim to compute

$$\widetilde{\mu}_\ell^2 := \sum_{i=0}^N \widetilde{\mu}_\ell(E_i)^2, \quad \text{where} \quad \widetilde{\mu}_\ell(E_i)^2 := \operatorname{length}(E_i)\|(\widehat{U}_\ell - I_\ell \widehat{U}_\ell)'\|^2_{L^2(E_i)}.$$

We adopt the notation of Section 6.6.1, namely $\widehat{\mathbf{x}} \in \mathbb{R}^{2N}$ with

$$\widehat{U}_\ell = \sum_{j=1}^{2N} \widehat{\mathbf{x}}_j \widehat{\zeta}_j.$$

Based on the same ideas as for the realization of the local contributions from the preceding Sections 6.6.2 and 6.6.3, a concise documentation of Listing 22 reads as follows:

- The function takes the meshes $\mathcal{E}_\ell$ and $\widehat{\mathcal{E}}_\ell$, the link between $\mathcal{E}_\ell$ and $\widehat{\mathcal{E}}_\ell$, and the coefficient vector $\widehat{\mathbf{x}} \in \mathbb{R}^{2N}$ of $\widehat{U}_\ell$ (Line 1–2).
- Adopting the ideas of Section 6.6.2, we compute the coefficient vector of $\widehat{U}_\ell - I_\ell \widehat{U}_\ell$ (Line 7–21).
- According to Section 6.6.3, we compute the local contributions $\mathrm{length}(e_j)^2 |(\widehat{U}_\ell - I_\ell \widehat{U}_\ell)'|^2$ for all elements $e_j \in \widehat{\mathcal{E}}_\ell$ (Line 25).
- Finally (Line 28), the function returns the vector

  $$\mathbf{v} := (\widetilde{\mu}_\ell(E_1)^2, \dots, \widetilde{\mu}_\ell(E_N)^2) \in \mathbb{R}^N.$$

  In particular, there holds $\widetilde{\mu}_\ell = \big(\sum_{i=1}^N \mathbf{v}_i\big)^{1/2}$.

6.7. Adaptive Mesh-Refinement for Hypersingular Integral Equation.

Usually computing time and memory requirements are limiting quantities for numerical simulations. Therefore, one aims to choose the mesh such that it is coarse, where the (unknown) solution is smooth, and fine, where the (unknown) solution is singular. Based on a local error estimator, e.g. $\widetilde{\mu}_\ell$, such meshes are constructed in an iterative way. In each step, one refines the mesh only locally, i.e. one refines elements E_j, where the error appears to be large, namely, where the local contributions $\widetilde{\mu}_\ell(E_j)$ are large. For the error estimator $\widetilde{\mu}_\ell$ from Section 6.6.4, a possible adaptive algorithm reads as follows:

Input: Initial mesh $\mathcal{E}_0$, Neumann data ϕ, adaptivity parameter $0 < \theta < 1$, maximal number $N_{\max} \in \mathbb{N}$ of elements, and counter $\ell = 0$.

(i) Build uniformly refined mesh $\widehat{\mathcal{E}}_\ell$.
(ii) Compute Galerkin solution $\widehat{U}_\ell \in \mathcal{S}^1(\widehat{\mathcal{E}}_\ell)$.
(iii) Compute refinement indicators $\widetilde{\mu}_\ell(E)^2$ and oscillation terms $\mathrm{osc}_{N,\ell}(E)^2$ for all $E \in \mathcal{E}_\ell$.
(iv) Find minimal set $\mathcal{M}_\ell \subseteq \mathcal{E}_\ell$ such that

$$\theta\,(\widetilde{\mu}_\ell^2 + \mathrm{osc}_{N,\ell}^2) = \theta \sum_{E \in \mathcal{E}_\ell} \Big(\widetilde{\mu}_\ell(E)^2 + \mathrm{osc}_{N,\ell}(E)^2\Big) \le \sum_{E \in \mathcal{M}_\ell} \Big(\widetilde{\mu}_\ell(E)^2 + \mathrm{osc}_{N,\ell}(E)^2\Big). \tag{6.36}$$

(v) Refine at least marked elements $E \in \mathcal{M}_\ell$ and obtain mesh $\mathcal{E}_{\ell+1}$ with $\kappa(\mathcal{E}_{\ell+1}) \le 2\kappa(\mathcal{E}_0)$.
(vi) Stop provided that $\#\mathcal{E}_{\ell+1} \ge N_{\max}$; otherwise, increase counter $\ell \mapsto \ell + 1$ and go to (i).

Output: Adaptively generated mesh $\widehat{\mathcal{E}}_\ell$ and corresponding discrete solution $\widehat{U}_\ell \in \mathcal{S}^1(\widehat{\mathcal{E}}_\ell)$.

The marking criterion (6.36) has been proposed in the context of adaptive finite element methods [13]. Let formally $N_{\max} = \infty$ so that the adaptive algorithm computes a sequence of discrete solutions $\widehat{U}_\ell$ (or even U_ℓ, although this is not computed). With the same techniques as in [17], one can prove that the saturation assumption (6.31) implies convergence of $\widehat{U}_\ell$ and U_ℓ to u, provided that the right-hand side ϕ is not disturbed, i.e., $\phi = \Phi_\ell$. The same result also holds for $\widetilde{\mu}_\ell$ replaced by μ_ℓ.

In [4], we changed the notion of convergence and proved that for certain error estimators — amongst them are $\widetilde{\mu}_\ell$ and μ_ℓ for Symm's integral equation — the adaptive algorithm guarantees $\lim_\ell \widetilde{\mu}_\ell = 0$. This concept is followed in [5] to prove that the adaptive algorithm for Symm's integral equation stated above, yields $\lim_\ell(\widetilde{\mu}_\ell^2 + \mathrm{osc}_{D,\ell}^2) = 0$. The same ideas are applicable to the hypersingular integral equation. Therefore, if the saturation assumption (6.31) holds (at least in infinitely many steps), we obtain convergence of U_ℓ to u due to $\|u - U_\ell\|_{W+S}^2 \lesssim \widetilde{\mu}_\ell^2 + \mathrm{osc}_{N,\ell}^2$.

LISTING 23. Implementation of Adaptive Algorithm

```
% adaptiveHypsing provides the implementation of an adaptive mesh-refining
% algorithm for the hypersingular integral equation.

%*** maximal number of elements
nEmax = 200;

%*** adaptivity parameter
theta = 0.25;
rho = 0.25;

%*** adaptive mesh-refining algorithm
while 1

    fprintf('number of elements: N = %d\r',size(elements,1))

    %*** build uniformly refined mesh
    [coordinates_fine,elements_fine,father2son] ...
        = refineBoundaryMesh(coordinates,elements);

    %*** discretize Neumann data and compute data oscillations
    [osc_fine,phih_fine] ...
        = computeOscNeumann(coordinates_fine,elements_fine,@phi);
    osc = osc_fine(father2son(:,1)) + osc_fine(father2son(:,2));

    %*** compute fine-mesh solution
    W_fine = buildW(coordinates_fine,elements_fine) ...
             + buildHypsingStabilization(coordinates_fine,elements_fine);
    b_fine = buildHypsingRHS(coordinates_fine,elements_fine,phih_fine);
    x_fine = W_fine\b_fine;

    %*** compute (h-h/2)-error estimator tilde-mu
    mu_tilde = computeEstHypMuTilde(elements_fine,elements,father2son,...
                                        x_fine);

    %*** mark elements for refinement
    marked = markElements(theta,rho,mu_tilde + osc);

    %*** generate new mesh
    [coordinates,elements] = refineBoundaryMesh(coordinates,elements,marked);

    if size(elements,1) > nEmax
      break;
    end
end

%*** visualize exact and adaptively computed solution
plotArclengthS1(coordinates_fine,elements_fine,x_fine,@g);
```

6.7.1. Implementation of Adaptive Algorithm (Listing 23). The MATLAB script of Listing 23 realizes the adaptive Algorithm from the beginning of this section.

- We use the adaptivity parameter $\theta = 1/4$ in (6.36) and mark at least the 25% of elements with the largest indicators (Line 8–9). The marking criterion is explained in Section 4.1.

7. Mixed Problem

Continuous Model Problem. Let $\Gamma = \partial\Omega$ be the piecewise affine boundary of a polygonal Lipschitz domain $\Omega \subset \mathbb{R}^2$. We assume that Γ is split into two disjoint and relatively open sets Γ_D and Γ_N with $\Gamma = \overline{\Gamma}_N \cup \overline{\Gamma}_D$. Moreover, we assume positive surface measure $|\Gamma_D| > 0$ to avoid treating the pure Neumann problem from Section 6, and we assume that Γ_N is *not* closed. This is satisfied if, for instance, Ω is simply connected. For given Dirichlet data $u_D \in H^{1/2}(\Gamma_D)$ and Neumann data $\phi_N \in H^{-1/2}(\Gamma_N)$, we consider the mixed boundary value problem

$$\begin{aligned} -\Delta u &= 0 && \text{in } \Omega, \\ u &= u_D && \text{on } \Gamma_D, \\ \partial_n u &= \phi_N && \text{on } \Gamma_N. \end{aligned} \tag{7.1}$$

For the equivalent integral formulation of (7.1), we choose (and fix) arbitrary extensions $\overline{u}_D \in H^{1/2}(\Gamma)$ and $\overline{\phi}_N \in H^{-1/2}(\Gamma)$ of the given data from Γ_D resp. Γ_N to the entire boundary Γ. The missing boundary data, which have to be computed, are

$$u_N := u - \overline{u}_D \quad \text{and} \quad \phi_D := \partial_n u - \overline{\phi}_N. \tag{7.2}$$

One can show that this definition yields $u_N \in \widetilde{H}^{1/2}(\Gamma_N)$ and $\phi_D \in \widetilde{H}^{-1/2}(\Gamma_D)$.

Let V denote the simple-layer potential, K the double-layer potential with adjoint K', and W the hypersingular integral operator. With the so-called Calderón projector

$$A = \begin{pmatrix} -K & V \\ W & K' \end{pmatrix}, \tag{7.3}$$

which is an operator matrix, the unknown data u_N and ϕ_D satisfy the following system of integral equations

$$A \begin{pmatrix} u_N \\ \phi_D \end{pmatrix} = (1/2 - A) \begin{pmatrix} \overline{u}_D \\ \overline{\phi}_N \end{pmatrix} =: \mathcal{F} \quad \text{on } \Gamma_D \times \Gamma_N. \tag{7.4}$$

One can prove that (7.4) is, in fact, an equivalent formulation of the mixed boundary value problem (7.1). With the spaces

$$\mathcal{H} := \widetilde{H}^{1/2}(\Gamma_N) \times \widetilde{H}^{-1/2}(\Gamma_D) \quad \text{and} \quad \mathcal{H}^* := H^{1/2}(\Gamma_D) \times H^{-1/2}(\Gamma_N), \tag{7.5}$$

one can show that $A : \mathcal{H} \to \mathcal{H}^*$ is a linear and continuous mapping. Moreover, $\mathcal{H}^*$ is the dual space of $\mathcal{H}$ with duality understood via the formula

$$\langle (v_D, \psi_N)\,,\, (v_N, \psi_D) \rangle_{\mathcal{H}^* \times \mathcal{H}} := \langle \psi_N\,,\, v_N \rangle_{\Gamma_N} + \langle \psi_D\,,\, v_D \rangle_{\Gamma_D} \tag{7.6}$$

for all $(v_N, \psi_D) \in \mathcal{H}$ and $(v_D, \psi_N) \in \mathcal{H}^*$, where the duality brackets $\langle \cdot\,,\, \cdot \rangle_{\Gamma_N}$ and $\langle \cdot\,,\, \cdot \rangle_{\Gamma_D}$ on the right-hand side denote the extended L^2-scalar products. Now, the operator A induces a continuous bilinear form on $\mathcal{H}$ via

$$\begin{aligned} \langle\!\langle (u_N, \phi_D)\,, (v_N, \psi_D) \rangle\!\rangle_A &:= \langle A(u_N, \phi_D)\,,\, (v_N, \psi_D) \rangle_{\mathcal{H}^* \times \mathcal{H}} \\ &= \langle W u_N + K' \phi_D\,,\, v_N \rangle_{\Gamma_N} + \langle -K u_N + V \phi_D\,,\, \psi_D \rangle_{\Gamma_D}. \end{aligned} \tag{7.7}$$

Note that this bilinear form is non-symmetric because of the entries $-K$ and K' on the right-hand side. Nevertheless, the definition

$$|\!|\!| (u_N, \phi_D) |\!|\!|_A^2 := \langle\!\langle (u_N, \phi_D)\,, (u_N, \phi_D) \rangle\!\rangle_A = \langle W u_N\,,\, u_N \rangle_{\Gamma_N} + \langle V \phi_D\,,\, \phi_D \rangle_{\Gamma_D} \tag{7.8}$$

provides a norm on $\mathcal{H}$ which is equivalent to the usual product norm. Therefore, the bilinear form $\langle\!\langle \cdot\,, \cdot \rangle\!\rangle_A$ is uniformly elliptic, and we are in the framework of the Lax-Milgram lemma. Consequently, the variational form of (7.4)

$$\langle\!\langle (u_N, \phi_D)\,, (v_N, \psi_D) \rangle\!\rangle_A = \langle F\,,\, (v_N, \psi_D) \rangle_{\mathcal{H}^* \times \mathcal{H}} \quad \text{for all } (v_N, \psi_D) \in \mathcal{H} \tag{7.9}$$

has a unique solution $(u_N, \phi_D) \in \mathcal{H}$. To abbreviate notation, we will now use the vector-valued unknown $\mathbf{u} := (u_N, \phi_D) \in \mathcal{H}$.

Mesh Restriction and Discrete Spaces. Let $\mathcal{E}_\ell$ be a mesh of Γ. By definition, $\mathcal{E}_\ell$ then resolves Γ_D and Γ_N, cf. Section 1.1. Consequently,

$$\mathcal{E}_\ell|_{\Gamma_D} := \big\{E \in \mathcal{E}_\ell \,:\, E \subseteq \overline{\Gamma}_D\big\} \quad \text{and} \quad \mathcal{E}_\ell|_{\Gamma_N} := \big\{E \in \mathcal{E}_\ell \,:\, E \subseteq \overline{\Gamma}_N\big\}$$

define meshes of Γ_D and Γ_N, respectively. By now, we have thus defined the discrete spaces $\mathcal{P}^0(\mathcal{E}_\ell)$, $\mathcal{P}^0(\mathcal{E}_\ell|_{\Gamma_D})$, $\mathcal{P}^0(\mathcal{E}_\ell|_{\Gamma_N})$, $\mathcal{S}^1(\mathcal{E}_\ell)$, $\mathcal{S}^1(\mathcal{E}_\ell|_{\Gamma_D})$, and $\mathcal{S}^1(\mathcal{E}_\ell|_{\Gamma_N})$. In addition, we now define the discrete space

$$\mathcal{S}^1_0(\mathcal{E}_\ell|_{\Gamma_N}) := \big\{V_\ell|_{\Gamma_N} \,:\, V_\ell \in \mathcal{S}^1(\mathcal{E}_\ell) \text{ with } V_\ell|_{\Gamma_D} = 0\big\},$$

i.e., $V_\ell \in \mathcal{S}^1_0(\mathcal{E}_\ell|_{\Gamma_N})$ is a continuous and piecewise affine function which vanishes at the tips of Γ_N. One can then show, that $\mathcal{S}^1_0(\mathcal{E}_\ell|_{\Gamma_N})$ is a discrete subspace of $\widetilde{H}^{1/2}(\Gamma_N)$, whereas $\mathcal{P}^0(\mathcal{E}_\ell|_{\Gamma_D})$ is a subspace of $\widetilde{H}^{-1/2}(\Gamma_D)$.

Extension of the Given Dirichlet and Neumann Data. By Definition (7.2), the solution $\mathbf{u} = (u_N, \phi_D)$ of (7.9) depends on the chosen extensions $\overline{u}_D$ of u_D and $\overline{\phi}_N$ of ϕ_N. We assume additional regularity

$$u_D \in H^1(\Gamma_D) \subset H^{1/2}(\Gamma_D) \quad \text{and} \quad \phi_N \in L^2(\Gamma_N) \subset H^{-1/2}(\Gamma_N). \tag{7.10}$$

Let $\mathcal{E}_0$ be the initial mesh for our numerical computation. We then define $\overline{\phi}_N \in L^2(\Gamma)$ by

$$\overline{\phi}_N|_{\Gamma_N} = \phi_N \quad \text{and} \quad \overline{\phi}_N|_{\Gamma_D} = 0 \tag{7.11}$$

as well as $\overline{u}_D \in H^1(\Gamma)$ by

$$\overline{u}_D|_{\Gamma_D} = u_D \quad \text{and} \quad \overline{u}_D|_{\Gamma_N} \in \mathcal{S}^1(\mathcal{E}_0|_{\Gamma_N}) \text{ with } \overline{u}_D(z) = 0 \text{ for all } z \in \mathcal{K}_0 \cap \Gamma_N. \tag{7.12}$$

As a consequence of the inclusion $H^1(\Gamma) \subset C(\Gamma)$, this extension is unique.

Galerkin Discretization. Let $\mathcal{E}_\ell = \{E_1, \dots, E_N\}$ be a mesh of Γ obtained by certain refinements of the initial mesh $\mathcal{E}_0$. To discretize (7.9), we replace the continuous Dirichlet data $\overline{u}_D \in H^1(\Gamma) \subset C(\Gamma)$ by the nodal interpolant

$$U_{D,\ell} := \sum_{j=1}^N \overline{u}_D(z_j)\zeta_j \in \mathcal{S}^1(\mathcal{E}_\ell) \subset H^1(\Gamma) \tag{7.13}$$

and the Neumann data by its L^2-projection

$$\Phi_{N,\ell} \in \mathcal{P}^0(\mathcal{E}_\ell), \quad \Phi_{N,\ell}|_{E_i} := \frac{1}{\operatorname{length}(E_i)} \int_{E_i} \overline{\phi}_N \, d\Gamma =: \mathbf{p}_i. \tag{7.14}$$

With the vector $\mathbf{g}_i := \overline{u}_D(z_i)$, this leads to the representations

$$U_{D,\ell} = \sum_{i=1}^N \mathbf{g}_i\zeta_i \quad \text{and} \quad \Phi_{N,\ell} = \sum_{i=1}^N \mathbf{p}_i\chi_i = \sum_{\substack{j=1 \\ E_j \subseteq \overline{\Gamma}_N}}^N \mathbf{p}_j\chi_j. \tag{7.15}$$

Here, the representation for $\Phi_{N,\ell}$ shrinks to a sum over all elements on the Neumann boundary by definition (7.11) of the extended Neumann data. The representation of $U_{D,\ell}$, however, takes into account all nodes. This is due to the fact that the extension $\overline{u}_D$ of u_D has to be continuous. This leads to $\operatorname{supp}(\overline{u}_D) \cap \Gamma_N \neq \emptyset$ in general. Restricting the sum for $U_{D,\ell}$ to Dirichlet nodes, would thus correspond to a change of the extension $\overline{u}_D$, whence the first component u_N of the solution $\mathbf{u} \in \mathcal{H}$ in every step ℓ!

We now consider the lowest-order Galerkin scheme and replace $\mathcal{H}$ by the discrete space

$$X_\ell := \mathcal{S}^1_0(\mathcal{E}_\ell|_{\Gamma_N}) \times \mathcal{P}^0(\mathcal{E}_\ell|_{\Gamma_D}) \subset \mathcal{H}. \tag{7.16}$$

Altogether, this leads to the following discrete version of the integral equation (7.4): Find $\mathbf{U}_\ell \in X_\ell$ with

$$\langle\!\langle \mathbf{U}_\ell \,, \mathbf{V}_\ell \rangle\!\rangle_A = \langle \mathcal{F}_\ell \,,\, \mathbf{V}_\ell \rangle_{\mathcal{H}^* \times \mathcal{H}} \quad \text{for all } \mathbf{V}_\ell \in X_\ell, \tag{7.17}$$

where the approximated right-hand side is given by

$$\mathcal{F}_\ell := (1/2 - A) \begin{pmatrix} U_{D,\ell} \\ \Phi_{N,\ell} \end{pmatrix}. \tag{7.18}$$

We use (7.18) here because the right hand side of (7.4) can hardly be evaluated numerically. In order to write (7.17) as a linear system of equations

$$\mathbf{A}\mathbf{x} = \mathbf{b}, \tag{7.19}$$

we have to fix a basis of the discrete space X_ℓ:

- Let $\mathcal{E}_\ell = \{E_1, \dots, E_N\}$ and assume that $\overline{\Gamma}_D = \bigcup_{j=1}^d E_j$. Then, $\{\chi_1, \dots, \chi_N\}$ is a basis of $\mathcal{P}^0(\mathcal{E}_\ell)$ and $\{\chi_1, \dots, \chi_d\}$ is a basis of $\mathcal{P}^0(\mathcal{E}_\ell|_{\Gamma_D})$.
- Let $\mathcal{K}_\ell = \{z_1, \dots, z_N\}$ and assume that $\{z_1, \dots, z_n\} = \mathcal{K}_\ell \cap \Gamma_N$. Then, $\{\zeta_1, \dots, \zeta_N\}$ is a basis of $\mathcal{S}^1(\mathcal{E}_\ell)$ and $\{\zeta_1, \dots, \zeta_n\}$ is a basis of $\mathcal{S}^1_0(\mathcal{E}_\ell|_{\Gamma_N})$.
- In particular, $\{(\zeta_1, 0), \dots, (\zeta_n, 0), (0, \chi_1), \dots, (0, \chi_d)\}$ is a basis of X_ℓ, and we fix this ordering for the implementation.

With this basis, the assembly of the the Galerkin data $\mathbf{A} \in \mathbb{R}^{(n+d)\times(n+d)}$ and $\mathbf{b} \in \mathbb{R}^{n+d}$ from (7.19) reads as follows: According to Linear Algebra, the Galerkin system (7.17) holds for all $\mathbf{V}_\ell \in X_\ell$ if it holds for all basis functions $(\zeta_j, 0)$ and $(0, \chi_k)$ of X_ℓ. Consequently, we need to compute the vector

$$\mathbf{b} \in \mathbb{R}^{n+d}, \quad \text{where} \quad \mathbf{b}_j := \langle F_\ell \,,\, (\zeta_j, 0)\rangle_{\mathcal{H}^*\times\mathcal{H}}, \quad \mathbf{b}_{n+k} := \langle F_\ell \,,\, (0, \chi_k)\rangle_{\mathcal{H}^*\times\mathcal{H}}, \tag{7.20}$$

for all $j = 1, \dots, n$ and $k = 1, \dots, d$. Recall the matrices $\mathbf{M}, \mathbf{K} \in \mathbb{R}^{N\times N}$ defined in (5.10) and the matrix $\mathbf{W} \in \mathbb{R}^{N\times N}$ from (6.10). With the data representation (7.15), there holds

$$\begin{aligned}
\mathbf{b}_j &= \langle -WU_{D,\ell} + (1/2 - K')\Phi_{N,\ell} \,,\, \zeta_j\rangle_\Gamma \\
&= \frac{1}{2}\,\langle \Phi_{N,\ell} \,,\, \zeta_j\rangle_\Gamma - \langle \Phi_{N,\ell} \,,\, K\zeta_j\rangle_\Gamma - \langle WU_{D,\ell} \,,\, \zeta_j\rangle_\Gamma \\
&= \frac{1}{2}\sum_{i=1}^N \mathbf{p}_i \langle \chi_i \,,\, \zeta_j\rangle_\Gamma - \sum_{i=1}^N \mathbf{p}_i \langle \chi_i \,,\, K\zeta_j\rangle_\Gamma - \sum_{i=1}^N \mathbf{g}_i \langle W\zeta_i \,,\, \zeta_j\rangle_\Gamma \\
&= \Big(\frac{1}{2}\,\mathbf{M}^T\mathbf{p} - \mathbf{K}^T\mathbf{p} - \mathbf{W}\mathbf{g}\Big)_j \\
&= \Big(\frac{1}{2}\,\mathbf{M}^T\mathbf{p} - \mathbf{K}^T\mathbf{p} - \mathbf{W}^T\mathbf{g}\Big)_j,
\end{aligned}$$

where we have finally used the symmetry of $\mathbf{W}$. Now, also recall the matrix $\mathbf{V} \in \mathbb{R}^{N\times N}$ from (5.10). The same type of arguments leads to

$$\begin{aligned}
\mathbf{b}_{n+k} &= \langle (1/2 + K)U_{D,\ell} - V\Phi_{N,\ell} \,,\, \chi_k\rangle_\Gamma \\
&= \frac{1}{2}\,\langle U_{D,\ell} \,,\, \chi_k\rangle_\Gamma + \langle KU_{D,\ell} \,,\, \chi_k\rangle_\Gamma - \langle V\Phi_{N,\ell} \,,\, \chi_k\rangle_\Gamma \\
&= \frac{1}{2}\sum_{i=1}^N \mathbf{g}_i \langle \zeta_i \,,\, \chi_k\rangle_\Gamma + \sum_{i=1}^N \mathbf{g}_i \langle K\zeta_i \,,\, \chi_k\rangle_\Gamma - \sum_{i=1}^N \mathbf{p}_i \langle V\chi_i \,,\, \chi_k\rangle_\Gamma \\
&= \Big(\frac{1}{2}\,\mathbf{M}\mathbf{g} + \mathbf{K}\mathbf{g} - \mathbf{V}\mathbf{p}\Big)_k.
\end{aligned}$$

For the right-hand side vector $\mathbf{b}$, we thus obtain the short-hand notation

$$\mathbf{b} = \begin{pmatrix} \big(\frac{1}{2}\,\mathbf{p}^T\mathbf{M} - \mathbf{p}^T\mathbf{K} - \mathbf{g}^T\mathbf{W}\big)^T|_{\Gamma_N} \\ \big(\frac{1}{2}\,\mathbf{M}\mathbf{g} + \mathbf{K}\mathbf{g} - \mathbf{V}\mathbf{p}\big)|_{\Gamma_D} \end{pmatrix}. \tag{7.21}$$

To compute the entries of the Galerkin matrix $\mathbf{A}$, we proceed in the same way. With the coefficient vector $\mathbf{x} \in \mathbb{R}^{n+d}$ of the ansatz

$$\mathbf{U}_\ell = (U_{N,\ell}, \Phi_{D,\ell}) \in X_\ell, \quad U_{N,\ell} = \sum_{i=1}^{n} \mathbf{x}_i \zeta_i, \quad \Phi_{D,\ell} = \sum_{i=1}^{d} \mathbf{x}_{n+i} \chi_i,$$

it is easily seen that the entries of $\mathbf{A}$ read

$$\begin{aligned} \mathbf{A}_{ij} &= \langle\!\langle (\zeta_j, 0), (\zeta_i, 0) \rangle\!\rangle_A, & \mathbf{A}_{i,n+k} &= \langle\!\langle (0, \chi_k), (\zeta_i, 0) \rangle\!\rangle_A, \\ \mathbf{A}_{n+k,i} &= \langle\!\langle (\zeta_i, 0), (0, \chi_k) \rangle\!\rangle_A, & \mathbf{A}_{n+k,n+m} &= \langle\!\langle (0, \chi_m), (0, \chi_k) \rangle\!\rangle_A, \end{aligned}$$

for all $i, j = 1, \dots, n$ and $k, m = 1, \dots, d$. Now, a direct computation leads to

$$\begin{aligned} \mathbf{A}_{ij} &= \langle\!\langle (\zeta_j, 0), (\zeta_i, 0) \rangle\!\rangle_A = \langle A(\zeta_j, 0), (\zeta_i, 0) \rangle = \langle (-K\zeta_j, W\zeta_j), (\zeta_i, 0) \rangle \\ &= \langle W\zeta_j, \zeta_i \rangle \\ \mathbf{A}_{i,n+k} &= \langle\!\langle (0, \chi_k), (\zeta_i, 0) \rangle\!\rangle_A = \langle A(0, \chi_k), (\zeta_i, 0) \rangle = \langle (V\chi_k, K'\chi_k), (\zeta_i, 0) \rangle \\ &= \langle K'\chi_k, \zeta_i \rangle \\ \mathbf{A}_{n+k,i} &= \langle\!\langle (\zeta_i, 0), (0, \chi_k) \rangle\!\rangle_A = \langle A(\zeta_i, 0), (0, \chi_k) \rangle = \langle (-K\zeta_i, W\zeta_i), (0, \chi_k) \rangle \\ &= -\langle K\zeta_i, \chi_k \rangle \\ \mathbf{A}_{n+k,n+m} &= \langle\!\langle (0, \chi_m), (0, \chi_k) \rangle\!\rangle_A = \langle A(0, \chi_m), (0, \chi_k) \rangle = \langle (V\chi_m, K'\chi_k), (0, \chi_k) \rangle \\ &= \langle V\chi_m, \chi_k \rangle. \end{aligned}$$

Altogether, we obtain the short-hand notation

$$\begin{pmatrix} \mathbf{W}|_{\Gamma_N \times \Gamma_N} & \mathbf{K}^T|_{\Gamma_N \times \Gamma_D} \\ -\mathbf{K}|_{\Gamma_D \times \Gamma_N} & \mathbf{V}|_{\Gamma_D \times \Gamma_D} \end{pmatrix} \mathbf{x} = \begin{pmatrix} \left(\frac{1}{2}\mathbf{p}^T\mathbf{M} - \mathbf{p}^T\mathbf{K} - \mathbf{g}^T\mathbf{W}\right)^T|_{\Gamma_N} \\ \left(\frac{1}{2}\mathbf{M}\mathbf{g} + \mathbf{K}\mathbf{g} - \mathbf{V}\mathbf{p}\right)|_{\Gamma_D} \end{pmatrix} \tag{7.22}$$

for the linear system (7.19)

Listing 24. Compute Oscillations and Discrete Date for Mixed Problem

```
function [oscD,oscN,uDh,phiNh] = computeOscMixed(coordinates,dirichlet, ...
                                    neumann, varargin)
uDh = zeros(size(coordinates,1),1);
if nargin == 5
    %*** if current mesh is the initial mesh
    [uD,phiN] = varargin{:};
else
    %*** if current mesh is obtained by refinement
    [father2neumann,neumann_old,uDh_old,uD,phiN] = varargin{:};

    %*** prolongation of uDh_old to uDh on Neumann boundary GammaN
    uDh(neumann(father2neumann(:,1),2)) = 0.5*sum(uDh_old(neumann_old),2);
    uDh(neumann(father2neumann(:,1),1)) = uDh_old(neumann_old(:,1));
    uDh(neumann(father2neumann(:,2),2)) = uDh_old(neumann_old(:,2));
end

%*** discretize Dirichlet data and compute data oscillations on GammaD
[oscD,uDh_dirichlet] = computeOscDirichlet(coordinates,dirichlet,uD);

%*** update uDh on Dirichlet boundary GammaD
idx = unique(dirichlet);
uDh(idx) = uDh_dirichlet(idx);

%*** discretize Neumann data and compute data oscillations on GammaN
[oscN,phiNh] = computeOscNeumann(coordinates,neumann,phiN);

```

```
27 %*** prolongate coefficient vector from GammaN to entire boundary Gamma
28 phiNh = [zeros(size(dirichlet,1),1) ; phiNh];
```

7.1. Discretization of Boundary Data and Computation of Corresponding Data Oscillations (Listing 24). Instead of the correct variational form (7.9), i.e.

$$\langle\!\langle \mathbf{u}\,,\mathbf{v}\rangle\!\rangle_A = \langle \mathcal{F}\,,\,\mathbf{v}\rangle_{\mathcal{H}^*\times\mathcal{H}} \quad \text{for all } \mathbf{v}\in\mathcal{H}$$

with solution $\mathbf{u}=(u_N,\phi_D)\in\mathcal{H}$ and test functions $\mathbf{v}=(v_N,\psi_D)\in\mathcal{H}$, we solve the perturbed formulation

$$\langle\!\langle \mathbf{u}_\ell\,,\mathbf{v}\rangle\!\rangle_A = \langle \mathcal{F}_\ell\,,\,\mathbf{v}\rangle_{\mathcal{H}^*\times\mathcal{H}} \quad \text{for all } \mathbf{v}\in\mathcal{H}. \tag{7.23}$$

In [6], we prove that the error between the continuous solutions $\mathbf{u},\mathbf{u}_\ell\in\mathcal{H}$ is controlled by

$$\begin{aligned}
|\!|\!|\mathbf{u}-\mathbf{u}_\ell|\!|\!|_A &\lesssim \|\overline{u}_D - I_\ell\overline{u}_D\|_{H^{1/2}(\Gamma)} + \|\overline{\phi}_N - \Pi_\ell\overline{\phi}_N\|_{H^{-1/2}(\Gamma)}\\
&\lesssim \|h_\ell^{1/2}(\overline{u}_D - I_\ell\overline{u}_D)'\|_{L^2(\Gamma)} + \|h_\ell^{1/2}(\overline{\phi}_N - \Pi_\ell\overline{\phi}_N)\|_{L^2(\Gamma)}\\
&= \|h_\ell^{1/2}(\overline{u}_D - I_\ell\overline{u}_D)'\|_{L^2(\Gamma_D)} + \|h_\ell^{1/2}(\overline{\phi}_N - \Pi_\ell\overline{\phi}_N)\|_{L^2(\Gamma_N)}\\
&= \|h_\ell^{1/2}(u_D - I_\ell u_D)'\|_{L^2(\Gamma_D)} + \|h_\ell^{1/2}(\phi_N - \Pi_\ell\phi_N)\|_{L^2(\Gamma_N)}\\
&=: \operatorname{osc}_{D,\ell} + \operatorname{osc}_{N,\ell},
\end{aligned}$$

where we have used the definition of the chosen extensions $\overline{\phi}_N$ and $\overline{u}_D$. Since the Dörfler marking below uses Hilbert space structure for the indicators, we rewrite the latter estimate in the form

$$|\!|\!|\mathbf{u}-\mathbf{u}_\ell|\!|\!|_A^2 \lesssim \operatorname{osc}_{D,\ell}^2 + \operatorname{osc}_{N,\ell}^2 =: \operatorname{osc}_\ell^2. \tag{7.24}$$

Note that the right-hand side is computable. The implementation of $\operatorname{osc}_{D,\ell}$ and $\operatorname{osc}_{N,\ell}$ has, in principle, already been discussed in Section 5.1.1 and Section 6.1. We stress, however, that the numerical solution of the mixed boundary value problem includes the appropriate prolongation of the given discrete data since the solution u_N depends on the chosen extension $\overline{u}_D\in H^1(\Gamma)$.

We start with a coarse mesh $\mathcal{E}_0$ and choose the extended Dirichlet data $\overline{u}_D$ to satisfy $\overline{u}_D|_{\Gamma_N}\in\mathcal{S}^1(\mathcal{E}_0|_{\Gamma_N})$ with $\overline{u}_D(z_i)=0$ for all nodes $z_i\in\mathcal{K}_\ell\cap\Gamma_N$. We stress that for all subsequent meshes, which arise by mesh refinement, this extension must not be changed! This is realized in the following way: The data $U_{D,\ell}$ is the point evaluation of u_D on the Dirichlet boundary Γ_D, whereas on the Neumann boundary Γ_N it is just the prolongation of $U_{D,\ell-1}$ to the mesh $\mathcal{E}_\ell$, i.e. $U_{D,\ell-1}|_{\Gamma_N}=U_{D,\ell}|_{\Gamma_N}$. This prolongation is part of function **`computeOscMixed`**:

- On the initial mesh $\mathcal{E}_0$, the function is called by

  ```
  [oscD,oscN,uDh,phiNh] = computeOscMixed(coordinates,dirichlet,neumann,uD,phiN);
  ```

  Here, the mesh $\mathcal{E}_0$ is described in terms of the arrays `coordinates`, `dirichlet`, and `neumann`. The function handles `uD` and `phiN` allow the pointwise evaluation of the given data u_D and ϕ_N.
- If the mesh $\mathcal{E}_\ell$ is obtained by refinement of some mesh $\mathcal{E}_{\ell+1}$, the function is called by

  ```
  [oscD,oscN,uDh,phiNh] = computeOscMixed(coordinates,dirichlet,neumann, ...
                                          father2neumann,neumann_old,uDh_old, ...
                                          uD,phiN);
  ```

  In addition to the prior call, one additionally provides the mesh $\mathcal{E}_{\ell-1}|_{\Gamma_N}$ by `neumann_old` as well as the link between $\mathcal{E}_\ell|_{\Gamma_N}$ and $\mathcal{E}_{\ell-1}|_{\Gamma_N}$ in terms of the array `father2neumann`. Finally, `uDh_old` is the nodal vector of $U_{D,\ell-1}$.
- We initialize the nodal vector `uDh` of $U_{D,\ell}$ by zero (Line 3).
- In the prolongation step, we have to guarantee $U_{D,\ell}|_{\Gamma_N}=U_{D,\ell-1}|_{\Gamma_N}$ on the Neumann boundary. To that end, we proceed as follows: If a node z_i of the mesh $\mathcal{E}_\ell$ is a new node, i.e. $z_i\in\mathcal{K}_\ell\backslash\mathcal{K}_{\ell-1}$, it is thus the midpoint of some element $E=[z_j,z_k]$ of $\mathcal{E}_{\ell-1}$. Then, $U_{D,\ell}(z_i)=(U_{D,\ell-1}(z_j)+U_{D,\ell-1}(z_k))/2$ (Line 12). If a node z_i of the mesh $\mathcal{E}_\ell$ was also a

node of $\mathcal{E}_{\ell-1}$, i.e. $z_i \in \mathcal{K}_{\ell-1} \cap \mathcal{K}_\ell$, there holds $U_{D,\ell}(z_i) = U_{D,\ell-1}(z_i)$ (Line 13–14). Now, `uDh`$(j) = U_{D,\ell}(z_j)$ for $z_j \in \mathcal{K}_\ell \cap \Gamma_N$ is correct, and it remains to define `uDh`$(j) = U_{D,\ell}(z_j) = u_D(z_j)$ for $z_j \in \mathcal{K}_\ell \cap \overline{\Gamma}_D$.

- The Dirichlet data oscillations as well as the nodal values of $U_{D,\ell}$ on Γ_D are returned by call of **`computeOscDirichlet`** (Line 18). These values are used to update `uDh` so that `uDh` provides the nodal values of $U_{D,\ell} \in \mathcal{S}^1(\mathcal{T}_\ell)$ (Line 21–22).
- The Neumann data oscillations as well as the elementwise values of $\Phi_{N,\ell}|_{E_j}$ for $E_j \in \mathcal{E}_\ell|_{\Gamma_N}$ are returned by call of **`computeOscNeumann`** (Line 25). Note that we have assumed the numbering $\mathcal{E}_\ell = \{E_1, \dots, E_N\}$ as well as $\mathcal{E}_\ell|_{\Gamma_N} = \{E_{d+1}, ..., E_N\}$, where d is the number of elements on the Dirichlet boundary. Therefore, `phiNh` provides the coefficient vector of $\Phi_{N,\ell} \in \mathcal{P}^0(\mathcal{E}_\ell)$ (Line 28).

Listing 25. Build RHS for Mixed Problem

```
1  function [b_nodes,b_elements] = buildMixedRHS(coordinates,dirichlet,neumann, ...
2                                                                    V,K,W,uDh,phiNh)
3  elements = [dirichlet;neumann];
4  nE = size(elements,1);
5
6  %*** compute mass-type matrix for P0 x S1
7  M = buildM(coordinates,elements);
8
9  %*** compute full right-hand side
10 b_nodes = (0.5*phiNh'*M - phiNh'*K - uDh'*W)';
11 b_elements = M*uDh*0.5 + K*uDh - V*phiNh;
```

7.2. Build Right-Hand Side Vector (Listing 25). To compute the vector $\mathbf{b}$ from (7.19), we first recall the representation of $\mathbf{b}$ in (7.22),

$$\begin{pmatrix} \mathbf{b}_1 \\ \mathbf{b}_2 \end{pmatrix} := \begin{pmatrix} \left(\frac{1}{2}\mathbf{p}^T\mathbf{M} - \mathbf{g}^T\mathbf{W} - \mathbf{p}^T\mathbf{K}\right)^T |_{\Gamma_N} \\ \left(\frac{1}{2}\mathbf{M}\mathbf{g} + \mathbf{K}\mathbf{g} - \mathbf{V}\mathbf{p}\right)|_{\Gamma_D} \end{pmatrix}, \tag{7.25}$$

where $\mathbf{b}_1$ stems from testing with nodal basis functions $\zeta_j \in \mathcal{S}^1(\mathcal{T}_\ell)$, whereas $\mathbf{b}_2$ stems from testing with characteristic functions $\chi_k \in \mathcal{P}^0(\mathcal{T}_\ell)$.

The documentation of Listing 25 reads as follows:

- As input, the function takes the mesh $\mathcal{E}_\ell$ described in terms of `coordinates`, `dirichlet` and `neumann`. The vector `uDh` contains the nodal vector of $U_{D,\ell} \in \mathcal{S}^1(\mathcal{T}_\ell)$, i.e. `uDh` $= \mathbf{g}$, whereas `phiNh` provides the elementwise values of $\Phi_{N,\ell} \in \mathcal{P}^0(\mathcal{T}_\ell)$, i.e. `phiNh` $= \mathbf{p}$. The matrices `V`, `K`, and `W` are the matrices for the simple-layer potential, the double-layer potential, and the hypersingular integral operator for the mesh $\mathcal{E}_\ell$.
- The return vector `b_nodes` provides the nodal contributions of $\mathbf{b}$ for all nodes, i.e. the vector $\mathbf{b}_1$ corresponds to a certain subvector of `b_nodes`. The return vector `b_elements` provides the element contributions of $\mathbf{b}$, i.e. $\mathbf{b}_2$ corresponds to a certain subvector of `b_elements`.
- First (Line 7), the mass-type matrix `M` is computed, cf. Section 5.3 for details.
- Line 10 computes the nodal contributions `b_nodes` as $(\frac{1}{2}\,\mathbf{p}^T\mathbf{M} - \mathbf{g}^T\mathbf{W} - \mathbf{p}^T\mathbf{K})^T \in \mathbb{R}^N$.
- Line 11 computes the element contributions `b_elements` as $\frac{1}{2}\,\mathbf{M}\mathbf{g} + \mathbf{K}\mathbf{g} - \mathbf{V}\mathbf{p} \in \mathbb{R}^N$.

7.3. Sort Mesh for Mixed Problem. As described above, for implementation we choose

$$\mathcal{B} := \{(\zeta_1, 0), \dots, (\zeta_n, 0), (0, \chi_1), \dots, (0, \chi_d)\}$$

as basis for X_ℓ. Here, $\{\zeta_1, \dots, \zeta_n\}$ is a basis of $\mathcal{S}^1_0(\mathcal{E}_\ell|_{\Gamma_N})$ and $\{\chi_1, \dots, \chi_d\}$ is a basis of $\mathcal{P}^0(\mathcal{E}_\ell|_{\Gamma_D})$.

For $\mathcal{S}^1_0(\mathcal{E}_\ell|_{\Gamma_N}) \subset \widetilde{H}^{1/2}(\Gamma_N)$ we aim to benefit from the functions already written for the hypersingular integral equation. To do so, we have to embed $\mathcal{S}^1_0(\mathcal{E}_\ell|_{\Gamma_N})$ into $\mathcal{S}^1(\mathcal{E}_\ell|_{\Gamma_N})$. We thus enforce an ordering of the nodes such that $\{z_1, \dots, z_m\} = \mathcal{K}_\ell \cap \overline{\Gamma_N} = \mathcal{K}_\ell \backslash \Gamma_D$, i.e., $\{\zeta_1, \dots, \zeta_m\}$

is a basis of $\mathcal{S}^1(\mathcal{E}_\ell|_{\Gamma_N})$ and, in particular, $n < m$. Note that for Γ_N connected, there holds $m = n + 2$.

The described ordering of the nodes enforces to do some reordering of the array `coordinates` and to adapt the indices in `dirichlet` and `neumann`. Both subjects are done by the function **`buildSortedMesh`**, see section 4.2. The right function call in this case is

```
[coordinates,neumann,dirichlet]=...
buildSortedMesh(coordinates,neumann,dirichlet).
```

LISTING 26. Transfer of `father2son` relations

```
1  function [f2s] = generateFather2Son(old2new,old2fine,new2fine)
3  %*** determine the number of elements of T_old_fine
4  f2s = zeros(size(unique(old2fine),1),2);
5
6  %*** mark old elements which are not refined, or refined
7  not_refined = old2new(:,1) == old2new(:,2);
8  refined = not(not_refined);
9
10 %*** transport the father2son arrays for the refined elements
11 f2s(old2fine(refined,1),:) = new2fine(old2new(refined,1),:);
12 f2s(old2fine(refined,2),:) = new2fine(old2new(refined,2),:);
13
14 %*** transport the father2son arrays for the elements which are not refined
15 f2s(old2fine(not_refined,1),:) = [ new2fine(old2new(not_refined,1),1) ...
16                                    new2fine(old2new(not_refined,1),1)];
17 f2s(old2fine(not_refined,2),:) = [ new2fine(old2new(not_refined,1),2) ...
18                                    new2fine(old2new(not_refined,1),2)];
```

7.4. Transfer of `father2son` relations (Listing 26). Although we use the $h - h/2$-strategy for error estimation, we note that it is sufficient to compute only $\widehat{\mathbf{U}}_\ell$, i.e., there is no need to compute the coarse mesh solution $\mathbf{U}_\ell$ (see section 7.6). It follows that in the ℓth step, the data $\overline{\phi}_N$ and $\overline{u}_D$ just need to be approximated on the uniform refined mesh $\widehat{\mathcal{E}}_\ell$. Suppose that the discrete Dirichlet data $\widehat{U}_{D,\ell}$ is available in the ℓth step. In the $\ell + 1$th step, we need to compute $\widehat{U}_{D,\ell+1}$. On Γ_N, this is done by prolongation from $\widehat{\mathcal{E}}_\ell$ to $\widehat{\mathcal{E}}_{\ell+1}$ by the function **`computeOscMixed`**. To that end, the last function needs the `father2son` relation `f2s` between $\widehat{\mathcal{E}}_\ell$ and $\widehat{\mathcal{E}}_{\ell+1}$. However, only the `father2son` relations `old2fine` from $\mathcal{E}_\ell$ to $\widehat{\mathcal{E}}_\ell$, `new2fine` from $\mathcal{E}_{\ell+1}$ to $\widehat{\mathcal{E}}_{\ell+1}$ as well as `old2new` from $\mathcal{E}_\ell$ to $\mathcal{E}_{\ell+1}$ are available from the refinement routines. The function **`generateFather2Son`** determines `f2s` from `old2new`, `old2fine` and `new2fine`.

The documentation of listing 26 reads as follows:

- As input, the function takes the `father2son` relations from a mesh $\mathcal{E}_\ell$ to a refined mesh $\mathcal{E}_{\ell+1}$, which are given by the array `old2new`. Further input parameters are the `father2son` relations between $\mathcal{E}_\ell$ and $\widehat{\mathcal{E}}_\ell$, which are given by the array `old2fine`, as well as the `father2son` relations between $\mathcal{E}_{\ell+1}$ and $\widehat{\mathcal{E}}_{\ell+1}$, which are given by the array `new2fine`. Note that these arrays can also describe parts of a boundary mesh. The output is the array `f2s`, which describes the `father2son` relation between $\widehat{\mathcal{E}}_\ell$ and $\widehat{\mathcal{E}}_{\ell+1}$.
- First, we provide memory for `f2s` (Line 4).
- We determine arrays `not_refined` and `refined`. Here, `refined(i)`$= 1$ and `not_refined(i)`$= 0$ if the element $E_i \in \mathcal{E}_\ell$ is refined, and vice versa (Lines 7 - 8).
- If an element $E_i \in \mathcal{E}_\ell$ is refined, then its uniform sons $\widehat{E}_{i_1}$, $\widehat{E}_{i_2} \in \widehat{\mathcal{E}}_\ell$ are also refined from $\widehat{\mathcal{E}}_\ell$ to $\widehat{\mathcal{E}}_{\ell+1}$ (Lines 11-12).
- If an element $E_i \in \mathcal{E}_\ell$ is not refined, i.e., $E_i = E_j \in \mathcal{E}_{\ell+1}$, then E_is uniform sons $\widehat{E}_{i_1}$, $\widehat{E}_{i_2} \in \widehat{\mathcal{E}}_\ell$, are not refined from $\widehat{\mathcal{E}}_\ell$ to $\widehat{\mathcal{E}}_{\ell+1}$ (Lines 15-18).

7.5. Computation of Reliable Error Bound for $|||\mathbf{u}-\mathbf{U}_\ell|||_A$. We assume that the exact solution has additional regularity $\mathbf{u} = (u_N, \phi_D) \in H^1(\Gamma_N) \times L^2(\Gamma_D)$. Let $\mathbf{U}_\ell^\star \in X_\ell$ be the (unknown, but existing) Galerkin solution with respect to the exact right-hand side F instead of F_ℓ. As in the previous section, one can prove that

$$|||\mathbf{U}_\ell^\star - \mathbf{U}_\ell|||_A \lesssim \mathrm{osc}_\ell.$$

Moreover, the exact Galerkin solution is quasi-optimal. Therefore,

$$\begin{aligned}|||\mathbf{u}-\mathbf{U}_\ell^\star|||_A &\lesssim |||u_N - I_\ell u_N|||_{W(\Gamma_N)} + |||\phi_D - \Pi_\ell \phi_D|||_{V(\Gamma_D)}\\ &\lesssim \|h_\ell^{1/2}(u_N - I_\ell u_N)'\|_{L^2(\Gamma_N)} + \|h_\ell^{1/2}(\phi_D - \Pi_\ell\phi_D)\|_{L^2(\Gamma_D)}\\ &=: \mathrm{err}_{N,\ell} + \mathrm{err}_{D,\ell}.\end{aligned}$$

Altogether, we obtain

$$|||\mathbf{u}-\mathbf{U}_\ell|||_A^2 \lesssim \mathrm{err}_\ell^2 + \mathrm{osc}_\ell^2 \quad \text{with} \quad \mathrm{err}_\ell^2 = \mathrm{err}_{D,\ell}^2 + \mathrm{err}_{N,\ell}^2.$$

Note that the computation of $\mathrm{err}_{N,\ell}$ and $\mathrm{err}_{D,\ell}$ has already been discussed in Section 5.5 and Section 6.5.

7.6. Computation of $(h-h/2)$-Based A Posteriori Error Estimators. Note that the energy norm $|||\cdot|||_A$ induced by the Calderón projector A can be written in terms of the energy norms $|||\cdot|||_{V(\Gamma_D)}$ and $|||\cdot|||_{W(\Gamma_N)}$ induced by the simple-layer potential $V \in L(\widetilde{H}^{-1/2}(\Gamma_D); H^{1/2}(\Gamma_D))$ and the hypersingular integral operator $W \in L(\widetilde{H}^{1/2}(\Gamma_N); H^{-1/2}(\Gamma_N))$. According to (7.8), there holds

$$|||(u_N, \phi_D)|||_A^2 = |||u_N|||_{W(\Gamma_N)}^2 + |||\phi_D|||_{V(\Gamma_D)}^2.$$

For a posteriori error estimation, we may therefore use the estimators introduced above. Suppose that

$$\mathbf{U}_\ell = (U_{N,\ell}, \Phi_{D,\ell}) \in X_\ell \quad \text{and} \quad \widehat{\mathbf{U}}_\ell = (\widehat{U}_{N,\ell}, \widehat{\Phi}_{D,\ell}) \in \widehat{X}_\ell$$

are Galerkin solutions with respect to the mesh $\mathcal{E}_\ell$ and its uniform refinement $\widehat{\mathcal{E}}_\ell$, computed with the same right-hand side $\mathcal{F}_\ell$, i.e. there holds

$$\langle\!\langle \mathbf{U}_\ell\,, \mathbf{V}_\ell\rangle\!\rangle_A = \langle \mathcal{F}_\ell\,,\, \mathbf{V}_\ell\rangle_{\mathcal{H}^*\times\mathcal{H}} \quad \text{for all } \mathbf{V}_\ell \in X_\ell \tag{7.26}$$

as well as

$$\langle\!\langle \widehat{\mathbf{U}}_\ell\,, \widehat{\mathbf{V}}_\ell\rangle\!\rangle_A = \langle \mathcal{F}_\ell\,,\, \widehat{\mathbf{V}}_\ell\rangle_{\mathcal{H}^*\times\mathcal{H}} \quad \text{for all } \widehat{\mathbf{V}}_\ell \in \widehat{X}_\ell. \tag{7.27}$$

As in Section 5.6, we define the following four error estimators for the part of the simple-layer potential:

$$\begin{aligned}\eta_{D,\ell} &:= |||\widehat{\Phi}_{D,\ell} - \Phi_{D,\ell}|||_{V(\Gamma_D)}, & \widetilde{\eta}_{D,\ell} &:= |||\widehat{\Phi}_{D,\ell} - \Pi_\ell\widehat{\Phi}_{D,\ell}|||_{V(\Gamma_D)},\\ \mu_{D,\ell} &:= \|h_\ell^{1/2}(\widehat{\Phi}_{D,\ell} - \Phi_{D,\ell})\|_{L^2(\Gamma_D)}, & \widetilde{\mu}_{D,\ell} &:= \|h_\ell^{1/2}(\widehat{\Phi}_{D,\ell} - \Pi_\ell\widehat{\Phi}_{D,\ell})\|_{L^2(\Gamma_D)}.\end{aligned}$$

In analogy to Section 6.6, we define the following four error estimators for the contribution of the hypersingular integral operator:

$$\begin{aligned}\eta_{N,\ell} &:= |||\widehat{U}_{N,\ell} - U_{N,\ell}|||_{W(\Gamma_N)}, & \widetilde{\eta}_{N,\ell} &:= |||\widehat{U}_{N,\ell} - I_\ell\widehat{U}_{N,\ell}|||_{W(\Gamma_N)},\\ \mu_{N,\ell} &:= \|h_\ell^{1/2}(\widehat{U}_{N,\ell} - U_{N,\ell})'\|_{L^2(\Gamma_N)}, & \widetilde{\mu}_{N,\ell} &:= \|h_\ell^{1/2}(\widehat{U}_{N,\ell} - I_\ell\widehat{U}_{N,\ell})'\|_{L^2(\Gamma_N)}.\end{aligned}$$

Consequently, we obtain (at least) four a posteriori error estimators for the mixed problem:

$$\begin{aligned}\eta_\ell^2 &:= \eta_{D,\ell}^2 + \eta_{N,\ell}^2, & \widetilde{\eta}_\ell^2 &:= \widetilde{\eta}_{D,\ell}^2 + \widetilde{\eta}_{N,\ell}^2,\\ \mu_\ell^2 &:= \mu_{D,\ell}^2 + \mu_{N,\ell}^2, & \widetilde{\mu}_\ell{}^2 &:= \widetilde{\mu}_{D,\ell}^2 + \widetilde{\mu}_{N,\ell}^2.\end{aligned}$$

We remark that the implementation of these error estimators has already been discussed above. With the analytical techniques from [18, 15] and [16], we prove in [6] that there holds equivalence

$$\eta_\ell \lesssim \widetilde{\eta}_\ell \lesssim \widetilde{\mu}_\ell \le \mu_\ell \lesssim \eta_\ell.$$

Moreover, there holds efficiency in the form

$$\eta_\ell \lesssim |||\mathbf{u} - \mathbf{U}_\ell|||_A + \mathrm{osc}_\ell.$$

The constants hidden in the symbol $\lesssim$ depend only on Γ and $\kappa(\mathcal{E}_\ell)$. Under a saturation assumption for the non-perturbed problem, there holds also reliability

$$|||\mathbf{u} - \mathbf{U}_\ell|||_A \lesssim \eta_\ell + \mathrm{osc}_\ell. \tag{7.28}$$

To steer an adaptive mesh-refining algorithm, it is therefore natural to use one of the combined error estimators

$$\begin{aligned}\varrho_\ell^2 &:= \mu_\ell^2 + \mathrm{osc}_\ell^2 = (\mu_{D,\ell}^2 + \mathrm{osc}_{D,\ell}^2) + (\mu_{N,\ell}^2 + \mathrm{osc}_{N,\ell}^2),\\ \widetilde{\varrho}_\ell^2 &:= \widetilde{\mu}_\ell^2 + \mathrm{osc}_\ell^2 = (\widetilde{\mu}_{D,\ell}^2 + \mathrm{osc}_{D,\ell}^2) + (\widetilde{\mu}_{N,\ell}^2 + \mathrm{osc}_{N,\ell}^2).\end{aligned}$$

For the same reasons as above, the usual choice is $\widetilde{\varrho}_\ell$ since it avoids the computation of the coarse-mesh Galerkin solution $\mathbf{U}_\ell \in X_\ell$, but only relies on local postprocessing of $\widehat{\mathbf{U}}_\ell$.

7.7. Adaptive Mesh-Refinement. For $E_j \in \mathcal{E}_\ell = \{E_1, \dots, E_N\}$, we consider the refinement indicators

$$\widetilde{\varrho}_\ell(E_j)^2 := \begin{cases} \widetilde{\mu}_{D,\ell}(E_j)^2 + \mathrm{osc}_{D,\ell}(E_j)^2 & \text{if } E_j \subseteq \overline{\Gamma}_D,\\ \widetilde{\mu}_{N,\ell}(E_j)^2 + \mathrm{osc}_{N,\ell}(E_j)^2 & \text{if } E_j \subseteq \overline{\Gamma}_N. \end{cases} \tag{7.29}$$

Note that there holds

$$\widetilde{\varrho}_\ell^2 = \sum_{j=1}^N \widetilde{\varrho}_\ell(E_j)^2. \tag{7.30}$$

With this notation, the adaptive algorithm takes the same form as before:

Input: Initial mesh $\mathcal{E}_0$, Dirichlet data u_D, Neumann data ϕ_N, adaptivity parameter $0 < \theta < 1$, maximal number $N_{\max} \in \mathbb{N}$ of elements, and counter $\ell = 0$.

(i) Build uniformly refined mesh $\widehat{\mathcal{E}}_\ell$.
(ii) Compute Galerkin solution $\widehat{\mathbf{U}}_\ell \in \widehat{X}_\ell$.
(iii) Compute refinement indicators $\widetilde{\varrho}_\ell(E)^2$ for all $E \in \mathcal{E}_\ell$.
(iv) Find minimal set $\mathcal{M}_\ell \subseteq \mathcal{E}_\ell$ such that

$$\theta\, \widetilde{\varrho}_\ell^2 = \theta \sum_{E \in \mathcal{E}_\ell} \widetilde{\varrho}_\ell(E)^2 \le \sum_{E \in \mathcal{M}_\ell} \widetilde{\varrho}_\ell(E)^2. \tag{7.31}$$

(v) Refine (at least) marked elements $E \in \mathcal{M}_\ell$ and obtain mesh $\mathcal{E}_{\ell+1}$ with $\kappa(\mathcal{E}_{\ell+1}) \le 2\kappa(\mathcal{E}_0)$.
(vi) Stop provided that $\#\mathcal{E}_{\ell+1} \ge N_{\max}$; otherwise, increase counter $\ell \mapsto \ell + 1$ and go to (i).

Output: Adaptively generated mesh $\widehat{\mathcal{E}}_\ell$ and corresponding discrete solution $\widehat{\mathbf{U}}_\ell \in \widehat{X}_\ell$.

The marking criterion (7.31) has been proposed in the context of adaptive finite element methods [13]. Let formally $N_{max} = \infty$ so that the adaptive algorithm computes a sequence of discrete solutions $\widehat{\mathbf{U}}_\ell$ (or even $\mathbf{U}_\ell$, although this is not computed). Based on (7.28), which holds under a saturation assumption for the non-perturbed problem, we can show with techniques introduced in [17] the convergence of $\widehat{\mathbf{U}}_\ell$ and $\mathbf{U}_\ell$ to $\mathbf{u}$, provided that the right hand side (u_D, ϕ_N) is not disturbed, i.e., $(u_D, \phi_N) = (U_{D,\ell}, \Phi_{N,\ell})$.

In [4], we changed the notion of convergence and proved that for certain error estimators — amongst them are $\widetilde{\mu}_\ell$ and μ_ℓ for Symm's integral equation — the adaptive algorithm guarantees convergence $\lim_\ell \widetilde{\mu}_\ell = 0$. This concept is followed in [6] to prove that the adaptive algorithm for the mixed problem stated above, yields $\lim_\ell \widetilde{\varrho}_\ell = 0$. Therefore, if the saturation assumption holds (at least in infinitely many steps), we obtain convergence of $\mathbf{U}_\ell$ to $\mathbf{u}$ due to $|||\mathbf{u} - \mathbf{U}_\ell|||_A^2 \lesssim \varrho_\ell^2$.

LISTING 27. Adaptive Algorithm for Mixed BVP

```
% adaptiveMixed provides the implementation of an adaptive mesh-refining
% algorithm for the symmetric integral formulation of a mixed boundary value
% problem.
%*** adaptivity parameter
theta = 0.25;
rho = 0.25;

%*** rearrange indices such that Neumann nodes are first
[coordinates,neumann,dirichlet] = ...
    buildSortedMesh(coordinates,neumann,dirichlet);

%*** initialize Dirichlet data
uDh = zeros(size(coordinates,1),1);
uDh(unique(dirichlet)) = g(coordinates(unique(dirichlet),:));

%*** Perform uniform refinement before starting the loop
%*** refine mesh uniformly
    [coordinates_fine,dirichlet_fine,neumann_fine,...
        father2dirichlet_fine,father2neumann_fine] ...
        = refineBoundaryMesh(coordinates,dirichlet,neumann);

%*** rearrange indices such that Neumann nodes are first
[coordinates_fine,neumann_fine,dirichlet_fine] = ...
    buildSortedMesh(coordinates_fine,neumann_fine,dirichlet_fine);

%*** discretize data and compute corresponding data oscillations for fine mesh
    [oscD_fine,oscN_fine,uDh_fine,phiNh_fine] ...
        = computeOscMixed(coordinates_fine,dirichlet_fine,neumann_fine, ...
                          father2neumann_fine,neumann,uDh,@g,@phi);

oscD = sum(oscD_fine(father2dirichlet_fine),2);
oscN = sum(oscN_fine(father2neumann_fine),2);

%*** adaptive mesh-refining algorithm
while 1

    fprintf('number of elements: N = %d\r',size(neumann,1)+size(dirichlet,1))

    %*** compute integral operators for fine mesh
    elements_fine = [dirichlet_fine;neumann_fine];
    V_fine = buildV(coordinates_fine,elements_fine);
    K_fine = buildK(coordinates_fine,elements_fine);
    W_fine = buildW(coordinates_fine,elements_fine);

    %*** compute right-hand side for fine mesh
    [b_nodes_fine,b_elements_fine] ...
        = buildMixedRHS(coordinates_fine,dirichlet_fine, ...
        neumann_fine,V_fine,K_fine,W_fine,uDh_fine,phiNh_fine);

    %*** compute degrees of freedom for fine mesh
    nC_fine = size(coordinates_fine,1);
    nD_fine = size(dirichlet_fine,1);
    freeNeumann_fine = setdiff(1:nC_fine,unique(dirichlet_fine));
    freeDirichlet_fine = 1:nD_fine;
    nN_fine = length(freeNeumann_fine);

```

```
57      %*** shrink integral operators and right-hand side
58      W_fine = W_fine(freeNeumann_fine,freeNeumann_fine);
59      K_fine = K_fine(freeDirichlet_fine,freeNeumann_fine);
60      V_fine = V_fine(freeDirichlet_fine,freeDirichlet_fine);
61      b_nodes_fine = b_nodes_fine(freeNeumann_fine);
62      b_elements_fine = b_elements_fine(freeDirichlet_fine);
63
64      %*** compute Galerkin solution for fine mesh
65      x = [W_fine K_fine' ; -K_fine V_fine] \ [b_nodes_fine ; b_elements_fine];
66
67      %*** compute coefficient vectors w.r.t. S1(GammaN) and P0(GammaD)
68      xN_fine = zeros(nC_fine,1);
69      xN_fine(freeNeumann_fine) = x(1:nN_fine);      %** dof on Neumann boundary
70      xD_fine = x((1:nD_fine) + nN_fine);            %** dof on Dirichlet boundary
71
72      %*** stopping criterion
73      if (size(neumann,1) + size(dirichlet,1) > nEmax )
74        break;
75      end
76
77      %*** compute (h-h/2)-error estimator tilde-mu on the associated boundaries
78      muD_tilde = computeEstSlpMuTilde(coordinates,dirichlet, ...
79          father2dirichlet_fine,xD_fine);
80      muN_tilde = computeEstHypMuTilde(neumann_fine,neumann, ...
81          father2neumann_fine,xN_fine);
82
83      %*** mark elements for refinement
84      [marked_dirichlet,marked_neumann] ...
85          = markElements(theta,rho,muD_tilde + oscD,muN_tilde + oscN);
86
87      %*** generate new mesh
88      [coordinates,dirichlet,neumann,father2dirichlet_adap,father2neumann_adap]...
89          = refineBoundaryMesh(coordinates,dirichlet,neumann,...
90                               marked_dirichlet,marked_neumann);
91
92      %*** rearrange indices such that Neumann nodes are first
93      [coordinates,neumann,dirichlet] = ...
94          buildSortedMesh(coordinates,neumann,dirichlet);
95
96      neumann_coarse = neumann_fine;
97      father2neumann_coarse = father2neumann_fine;
98      [coordinates_fine,dirichlet_fine,neumann_fine,father2dirichlet_fine,...
99          father2neumann_fine]= refineBoundaryMesh(coordinates,dirichlet,neumann);
100
101     %*** build coarse2fine array
102     coarse2fine = generateFather2Son(father2neumann_adap, ...
103                                      father2neumann_coarse, ...
104                                      father2neumann_fine);
105
106     %*** rearrange indices such that Neumann nodes are first
107     [coordinates_fine,neumann_fine,dirichlet_fine] = ...
108         buildSortedMesh(coordinates_fine,neumann_fine,dirichlet_fine);
109
110     %*** discretize data and compute corresponding data oscillations for
111     %*** fine mesh
112     [oscD_fine,oscN_fine,uDh_fine,phiNh_fine] ...
113         = computeOscMixed(coordinates_fine,dirichlet_fine,neumann_fine, ...
```

```
114                             coarse2fine,neumann_coarse,uDh_fine,@g,@phi);
115     oscD = sum(oscD_fine(father2dirichlet_fine),2);
116     oscN = sum(oscN_fine(father2neumann_fine),2);
117 end
118
119 %*** xN_fine = dof on Neumann boundary, i.e. update for Dirichlet data
120 %*** to obtain approximation uh of trace of PDE solution u
121 uh =  xN_fine+uDh_fine;
122
123 %*** plot approximate vs exact trace
124 plotArclengthS1(coordinates_fine,[dirichlet_fine;neumann_fine],uh,@g,2)
125
126 %*** xD_fine = dof on Dirichlet boundary, i.e. update for Neumann data
127 %*** to obtain approximation phih of normal derivative of PDE solution u
128 phih = [xD_fine ; phiNh_fine(nD_fine+1:end)];
129
130 %*** plot approximate vs exact normal derivative
131 plotArclengthP0(coordinates_fine,[dirichlet_fine;neumann_fine],phih,@phi,1)
```

7.7.1. Implementation of Adaptive Algorithm (Listing 27). The MATLAB script of Listing 27 realizes the adaptive algorithm from the beginning of this section.

- We use the adaptivity parameter $\theta = 1/4$ in (7.31) and mark at least 25% of elements with the largest indicators (Line 5–6).
- We order the nodes such that nodes on $\overline{\Gamma}_N$ are first (Line 9–10) and compute the nodal vector of $U_{D,0}$ (Line 13–14).

The remainder of the code consists of the adaptive loop, where $\mathcal{E}_\ell$ is a given mesh with associated discrete Dirichlet data $U_{D,\ell}$.

- We generate the mesh $\widehat{\mathcal{E}}_0$ (Line 18–20). Then, we discretize the given data on $\widehat{\mathcal{E}}_0$ and compute the corresponding data oscillations (Line 27–29). The $\widehat{\mathcal{E}}_0$-piecewise data oscillations are linked to the coarse mesh $\mathcal{E}_0$ (Line 31–32).
- We build the discrete integral operators related to $\widehat{\mathcal{E}}_\ell$ (Line 41–43) and the corresponding right-hand side (Line 46–48). Note that the latter is built with respect to the improved data $(\widehat{U}_{D,\ell}, \widehat{\Phi}_{N,\ell})$ instead of $(U_{D,\ell}, \Phi_{N,\ell})$.
- By definition, the degrees of freedom are the elements on the Dirichlet boundary, which are the first N_D elements (Line 54), as well as the nodes $\widehat{\mathcal{K}}_\ell \backslash \overline{\Gamma}_D$, which lie inside of Γ_N (Line 53).
- To lower the storage, we restrict the discrete operators and the right-hand side to the degrees of freedom (Line 58–62). For instance, $\mathbf{V}$ is only needed for elements on Γ_D, and $\mathbf{W}$ is only needed for nodes $z_\ell \in \mathcal{K}_\ell \backslash \overline{\Gamma}_D$.
- Finally (Line 65), we compute the coefficient vector $\widehat{\mathbf{x}}$ of $\widehat{\mathbf{U}}_\ell$ by solving (7.22).
- Next, we aim to obtain the basis vectors $\widehat{x}^N$ and $\widehat{x}^D$ of $U_{N,\ell}$ and $\Phi_{D,\ell}$, respectively. To use the functions from Section 6, we have to represent $U_{N,\ell}$ in the nodal basis of $\mathcal{S}^1(\mathcal{E}_\ell|_{\overline{\Gamma}_N})$. This is done in Line 68–69. The coefficients of $\Phi_{D,\ell}$ with respect to $\mathcal{P}^0(\mathcal{E}_\ell|_{\overline{\Gamma}_D})$ are obtained in Line 70.
- We compute the local contributions of the error estimator $\widetilde{\mu}_\ell^2 = \widetilde{\mu}_{D,\ell}^2 + \widetilde{\mu}_{N,\ell}^2$ (Line 78–81).
- In Line 84–85, the Dörfler marking (7.31) is realized.
- In the next step, the new mesh $\mathcal{E}_{\ell+1}$ is created (Line 88–90) and ordered (Line 93–94).
- In Line 96–97, we save `neumann_fine` and `father2neumann_fine`, as we need it in Line 102, where we generate the father2son relation from $\widehat{\mathcal{E}}_\ell$ to $\widehat{\mathcal{E}}_{\ell+1}$.
- The mesh $\widehat{\mathcal{E}}_{\ell+1}$ is created (Line 98–99) and the nodes are ordered (Line 107–108).
- Finally, we compute the data oscillations on $\widehat{\mathcal{E}}_{\ell+1}$ (Line 112–116).

8. Integral Equations with Non-Homogeneous Volume Force

So far, we have only considered the Laplace equation $-\Delta u = f$ with homogeneous volume force $f = 0$. In this section, we discuss the generalization for $f \neq 0$. Doing so, we have to deal with two integral operators which have not been considered, yet, namely the trace N_0 and the normal derivative $N_1 = \partial_n \widetilde{N}$ of the Newtonian potential $\widetilde{N}$ from (1.3).

Discretization of Volume Force. For the later implementation and to compute the Newtonian potential analytically, we will replace the volume force $f \in \widetilde{H}^{-1}(\Omega)$ by some discrete function. In addition to the boundary element mesh $\mathcal{E}_\ell$, we therefore have to deal with an additional partition $\mathcal{T}_\ell = \{T_1, \dots, T_M\}$ of the domain Ω. Throughout, we will assume that $\mathcal{T}_\ell$ is a regular triangulation of Ω into triangles, i.e.,

- all elements $T_j \in \mathcal{T}_\ell$ are compact non-degenerate triangles, i.e., $T_j = \operatorname{conv}\{z_1, z_2, z_3\} \subset \mathbb{R}^2$ with certain vertices $z_k \in \mathbb{R}^2$ and T_j has positive area $|T_j| > 0$,
- Ω is the union of the volume elements $\overline{\Omega} = \bigcup_{j=1}^M T_j$,
- the intersection of $T_j, T_k \in \mathcal{T}_\ell$, for $j \neq k$, is either disjoint or a common vertex or a common edge.

We extend the definition of the mesh-size function h_ℓ which is —from now on— defined as $h_\ell : \overline{\Omega} \to \mathbb{R}$ with

$$h_\ell(x) = \begin{cases} \operatorname{diam}(E_j) & \text{provided that } x \in \operatorname{interior}(E_j) \text{ for some boundary element } E_j \in \mathcal{E}_\ell, \\ \operatorname{diam}(T_j) & \text{provided that } x \in \operatorname{interior}(T_j) \text{ for some volume element } T_j \in \mathcal{T}_\ell. \end{cases}$$

Note that h_ℓ is pointwise defined almost everywhere with respect to both Ω and Γ and provides functions $h_\ell \in L^\infty(\Gamma)$ as well as $h_\ell \in L^\infty(\Omega)$. Throughout, we will assume (and algorithmically guarantee) that the generated volume meshes $\mathcal{T}_\ell$ are uniformly shape regular, i.e.

$$\sup_{\ell \in \mathbb{N}} \sigma(\mathcal{T}_\ell) < \infty, \quad \text{where} \quad \sigma(\mathcal{T}_\ell) := \max_{T_j \in \mathcal{T}_\ell} \frac{\operatorname{diam}(T_j)^2}{|T_j|} < \infty. \tag{8.1}$$

This is for instance guaranteed by use of any mesh-refinement based on newest vertex bisection (NVB), cf. [32].

By $\mathcal{P}^0(\mathcal{T}_\ell)$, we denote the space of all $\mathcal{T}_\ell$-piecewise constant functions. Since there are no ambiguities between $L^2(\Gamma)$ and $L^2(\Omega)$, we will denote the L^2-orthogonal projections onto $\mathcal{P}^0(\mathcal{E}_\ell)$ and $\mathcal{P}^0(\mathcal{T}_\ell)$ by $\Pi_\ell : L^2 \to \mathcal{P}^0$ in either case.

To discretize the non-homogeneous volume force $f \in \widetilde{H}^{-1}(\Omega)$, we assume additional regularity $f \in L^2(\Omega)$, and replace f by its L^2-projection $F_\ell := \Pi_\ell f \in \mathcal{P}^0(\mathcal{T}_\ell)$. Note that there holds

$$F_\ell|_{T_j} = \frac{1}{|T_j|} \int_{T_j} f \, dx =: \mathbf{f}_j, \tag{8.2}$$

where $|T_j|$ denote the area of T_j. With the characteristic functions χ_{T_j} corresponding to $T_j \in \mathcal{T}_\ell$, there holds

$$F_\ell = \sum_{j=1}^M \mathbf{f}_j \chi_{T_j}, \tag{8.3}$$

and we will use this representation to build the Galerkin data.

8.1. Symm's Integral Equation with Volume Forces. As first model problem, we consider the Dirichlet problem

$$-\Delta u = f \text{ in } \Omega \quad \text{with} \quad u = g \text{ on } \Gamma = \partial\Omega. \tag{8.4}$$

With the trace $N_0 f$ of the Newton potential $\widetilde{N} f$, this problem is equivalently stated in the integral formulation

$$V\phi = (K + 1/2)g - N_0 f \quad \text{on } \Gamma. \tag{8.5}$$

With the notation of Section 5, the data perturbed Galerkin formulation reads as follows: Find $\Phi_\ell \in \mathcal{P}^0(\mathcal{T}_\ell)$ such that

$$\langle V\Phi_\ell\,,\,\Psi_\ell\rangle_\Gamma = \langle (K+1/2)G_\ell\,,\,\Psi_\ell\rangle_\Gamma - \langle N_0 F_\ell\,,\,\Psi_\ell\rangle_\Gamma \quad \text{for all } \Psi_\ell \in \mathcal{P}^0(\mathcal{T}_\ell), \tag{8.6}$$

where $G_\ell \in \mathcal{S}^1(\mathcal{E}_\ell)$ approximates the given Dirichlet data g, cf. Section 5. We proceed as above and define the matrix $\mathbf{N} \in \mathbb{R}^{N\times M}$ by

$$\mathbf{N}_{kj} = \langle N\chi_{T_j}\,,\,\chi_{E_k}\rangle_\Gamma \quad \text{for all } T_j \in \mathcal{T}_\ell \text{ and } E_k \in \mathcal{E}_\ell, \tag{8.7}$$

where χ_{E_k}, $k = 1,\dots,M$ and χ_{T_j}, $j = 1,\dots,N$ denote the characteristic functions of E_k and T_j. Then, the Galerkin formulation (8.6) is equivalent to the linear system

$$\mathbf{V}\mathbf{x} = \Big(\mathbf{K} + \frac{1}{2}\,\mathbf{M}\Big)\mathbf{g} - \mathbf{N}\mathbf{f}, \tag{8.8}$$

where $\mathbf{f}$ is the coefficient vector of F_ℓ from (8.3) and where $\mathbf{g} \in \mathbb{R}^N$ is the nodal vector of G_ℓ from (5.5). In comparison to Section 5, only the right-hand side of (8.8) is modified by the additional vector $-\mathbf{N}\mathbf{f}$.

8.2. Computation of Discrete Newton Potential N. The matrix $\mathbf{N} \in \mathbb{R}^{N\times M}$ for the Newton potential which is defined in (8.7), is implemented in the programming language C via the MATLAB-MEX-Interface. The Newton potential matrix $\mathbf{N}$ is returned by call of

```
N = buildN(coordinates,elements,vertices,triangles [,eta]);
```

Here, the boundary mesh $\mathcal{E}_\ell$ is given in terms of the arrays `coordinates` and `elements`, whereas the volume mesh $\mathcal{T}_\ell$ is described in terms of `vertices` and `triangles`.

In general, the matrix entries of $\mathbf{N}$ can be computed almost analytically, i.e. up to quadrature of a smooth arctan, see [2]. However, analytic integration leads to cancellation effects if the integration domain is small. Recall that

$$\mathbf{N}_{kj} = -\frac{1}{2\pi}\int_{E_k}\int_{T_j} \log|x-y|\,dx\,d\Gamma(y),$$

so that the integration domain here means either the boundary element E_k or the volume element T_j. For fixed $\eta > 0$, a pair of elements $(E_k, T_j) \in \mathcal{E}_\ell \times \mathcal{T}_\ell$ is called *admissible* provided that

$$\operatorname{length}(E_k) \le \eta \operatorname{dist}(E_k, T_j).$$

Otherwise, the pair is called *inadmissible*.

For an inadmissible pair (E_k, T_j), the matrix entry $\mathbf{N}_{kj}$ is computed analytically. For an admissible pair, the outer integration is done by Gaussian quadrature, i.e.,

$$\begin{aligned}\int_{E_k}\int_{T_j} \log|x-y|\,dx\,d\Gamma(y) &= \frac{\operatorname{length}(h_k)}{2}\int_{-1}^{1}\int_{T_j} \log|x-\gamma_k(s)|\,dx\,ds \\ &\approx \frac{\operatorname{length}(h_k)}{2}\sum_{m=1}^{p}\omega_m\int_{T_j}\log|x-\gamma_k(s_m)|\,dx,\end{aligned}$$

whereas the remaining integral is computed analytically. For more information on the standard choices of p and η in `HILBERT` and how to change them, we refer to Section 5.2.

Remark 8.1. *We found empirically that, due to cancellation effects, the matrix* $\mathbf{N}$ *became instable in case that there are boundary elements* $E \in \mathcal{E}_\ell$ *and triangles* $T \in \mathcal{T}_\ell$ *with* $E \subset \partial T$ *and* $\operatorname{length}(E) \ll \operatorname{diam}(T)$. *One remedy to cure this instability is to consider coupled meshes, i.e., the boundary mesh* $\mathcal{E}_\ell = \mathcal{T}_\ell|_\Gamma$ *is the restriction of the volume mesh* $\mathcal{T}_\ell$ *to the boundary. Although one might think that this might lead to inefficiencies, we empirically observed optimal convergence behaviour. Therefore, this concept is followed in all our implementations. We refer to the function* **`buildSortedMesh`** *from Section 4.2 which returns* $\mathcal{T}_\ell|_\Gamma$.

LISTING 28. Computation of Data Oscillations for Volume Data

```
function [osc,fh] = computeOscVolume(vertices,triangles,f)
%*** quadrature rule on reference triangle Tref = conv{(0,0),(1,0),(0,1)}
tmp_pos = [6-sqrt(15) ; 9+2*sqrt(15) ; 6+sqrt(15) ; 9-2*sqrt(15) ; 7]/21;
quad_vertices = tmp_pos([1 1 ; 2 1 ; 1 2 ; 3 4 ; 3 3 ; 4 3 ; 5 5]);
tmp_wts = [155-sqrt(15) 155+sqrt(15) 270]/2400;
quad_weights = tmp_wts([1 1 1 2 2 2 3]);

%*** the remaining code is independent of the chosen quadrature rule
nT = size(triangles,1);
nQ = size(quad_vertices,1);

%*** first vertices of triangles and corresponding edge vectors
v1  = vertices(triangles(:,1),:);
d21 = vertices(triangles(:,2),:) - v1;
d31 = vertices(triangles(:,3),:) - v1;

%*** compute vector of triangle areas 2*area(T)
area2 = d21(:,1).*d31(:,2)-d21(:,2).*d31(:,1);

%*** build matrix of quadrature vertices by use of affine transformation of Tref
jacobian1 = reshape(repmat([d21(:,1);d31(:,1)],1,nQ)',nT*nQ,2);
jacobian2 = reshape(repmat([d21(:,2);d31(:,2)],1,nQ)',nT*nQ,2);
zref = repmat(quad_vertices,nT,1);
z = [ sum(zref.*jacobian1,2) sum(zref.*jacobian2,2) ] ...
    + reshape(repmat(v1(:),1,nQ)',nT*nQ,2);

%*** evaluate volume force f at all quadrature vertices z
fz = reshape(f(z),nQ,nT);

%*** compute integral mean
f_mean = 2*quad_weights*fz;

%*** return OSC(j) = area(Tj) *  || f - f_mean ||_{L2(Tj)}^2
osc = 0.5*area2.^2 .* (quad_weights*(fz - repmat(f_mean,nQ,1)).^2)';

%*** return column vector
fh = f_mean';
```

8.3. Discretization of Volume Data and Computation of Corresponding Volume Data Oscillations (Listing 28). In analogy to the techniques from [11], one can prove that the definition

$$\operatorname{osc}_{\Omega,\ell} := \Big(\sum_{j=1}^{M} \operatorname{osc}_{\Omega,\ell}(T_j)^2\Big)^{1/2} \quad \text{with} \quad \operatorname{osc}_{\Omega,\ell}(T_j)^2 = |T_j|\,\|f - F_\ell\|^2_{L^2(T_j)} \tag{8.9}$$

guarantees

$$\|f - F_\ell\|_{\widetilde{H}^{-1}(\Omega)} \lesssim \|h_\ell(f - F_\ell)\|_{L^2(\Omega)} \lesssim \operatorname{osc}_{\Omega,\ell}, \tag{8.10}$$

where we have used the uniform shape regularity (8.1) in the final estimate. As above, we stress that our numerical scheme aims to approximate the solution $\phi_\ell \in H^{-1/2}(\Gamma)$ of the perturbed formulation

$$V\phi_\ell = (K + 1/2)G_\ell - N_0 F_\ell \quad \text{on } \Gamma \tag{8.11}$$

instead of (8.5). One can, however, prove that

$$|||\phi - \phi_\ell|||_V \lesssim \operatorname{osc}_{D,\ell} + \operatorname{osc}_{\Omega,\ell}, \tag{8.12}$$

see [6].

For the computation of the local contributions $\operatorname{osc}_{\Omega,\ell}(T_j)$, we use an affine transformation Ξ_j from the reference element $T^{\mathrm{ref}} = \operatorname{conv}\{(0,0),(0,1),(1,0)\}$ to $T_j \in \mathcal{T}_\ell$. Let $T_j = \operatorname{conv}\{z_1, z_2, z_3\}$. We then define

$$\Xi_j x^{\mathrm{ref}} := M_j x^{\mathrm{ref}} + z_1 \quad \text{with} \quad M_j = \big(z_2 - z_1\,,\, z_3 - z_1\big) \in \mathbb{R}^{2\times 2}. \tag{8.13}$$

It is easily seen that $\Xi_j : T^{\mathrm{ref}} \to T_j$ is an affine bijection and that the functional determinant satisfies $|\det D\Xi_j(x^{\mathrm{ref}}) = |\det M_j| = 2\,|T_j|$. Consequently, the transformation theorem proves

$$2|T_j| \int_{T^{\mathrm{ref}}} (f \circ \Xi_j)(x^{\mathrm{ref}})\, dx^{\mathrm{ref}} = \int_{T^{\mathrm{ref}}} (f \circ \Xi_j)(x^{\mathrm{ref}})\, |D\Xi_j(x^{\mathrm{ref}})|\, dx^{\mathrm{ref}} = \int_{T_j} f\, dx$$

for any integrable $f \in L^1(T_j)$. Suppose that there are given quadrature nodes $x_1^{\mathrm{ref}}, \dots, x_p^{\mathrm{ref}} \in T^{\mathrm{ref}}$ and corresponding weights $\omega_1, \dots, \omega_p$ such that

$$\int_{T^{\mathrm{ref}}} (f \circ \Xi_j)(x^{\mathrm{ref}})\, dx^{\mathrm{ref}} \approx \sum_{k=1}^{p} \omega_k (f \circ \Xi_j)(x_k^{\mathrm{ref}}).$$

First, we use this quadrature formula to approximate the integral mean

$$F_\ell|_{T_j} = \frac{1}{|T_j|} \int_{T_j} f\, dx = 2 \int_{T^{\mathrm{ref}}} (f \circ \Xi_j)(x^{\mathrm{ref}})\, dx^{\mathrm{ref}} \approx 2 \sum_{k=1}^{p} \omega_k (f \circ \Xi_j)(x_k^{\mathrm{ref}}) =: \widetilde{\mathbf{f}}_j. \tag{8.14}$$

Second, we use the same quadrature formula to approximate the integral

$$\operatorname{osc}_{\Omega,\ell}(T_j)^2 \approx |T_j| \int_{T_j} |f - \widetilde{\mathbf{f}}_j|^2\, dx \approx 2|T_j|^2 \sum_{k=1}^{p} \omega_k\, |f \circ \Xi_j(x_k^{\mathrm{ref}}) - \widetilde{\mathbf{f}}_j|^2 =: \widetilde{\operatorname{osc}}_{\Omega,\ell}(T_j)^2. \tag{8.15}$$

For the implementation, we use a 7-point quadrature rule from [33] which is exact on $\mathcal{P}^5(T_j)$, i.e., for polynomials in $\mathbb{R}^2$ with maximal degree 5. Consequently, our implementation satisfies $\operatorname{osc}_{\Omega,\ell}(T_j) = \widetilde{\operatorname{osc}}_{\Omega,\ell}(T_j)$ for $f \in \mathcal{P}^2(T_j)$. The Bramble-Hilbert lemma thus proves, for smooth volume forces f

$$|\operatorname{osc}_{\Omega,\ell}(T_j) - \widetilde{\operatorname{osc}}_{\Omega,\ell}(T_j)| \lesssim |T_j|^{1/2} h^3 \simeq h^4,$$

whence

$$|\operatorname{osc}_{\Omega,\ell} - \widetilde{\operatorname{osc}}_{\Omega,\ell}| \lesssim (N h^8)^{1/2} = \mathcal{O}(h^3),$$

whereas only $\operatorname{osc}_{\Omega,\ell} = \mathcal{O}(h^2)$ for smooth f.

The vector $\mathbf{v} \in \mathbb{R}^M$ with $\mathbf{v}_j := \widetilde{\operatorname{osc}}_{\Omega,\ell}(T_j)^2$ is computed by the function **computeOscVolume** from Listing 28.

- We provide a symmetric 7-point quadrature on T^{ref} (Lines 2–6), and the remaining code is independent of the chosen quadrature rule.
- The vector `v1` contains all first nodes of the triangles. The vectors `d21` and `d31` contain the corresponding directional vectors which then determine the triangles $T_j \in \mathcal{T}_\ell$ (Line 13–15).
- In Line 18, we compute the column vector `area2` containing $|\det D\Xi_j(x^{\mathrm{ref}})| = 2\,|T_j|$.
- For each element T_j and each quadrature node x_k^{ref}, we have to evaluate $f\big(\Xi_j(x_k^{\mathrm{ref}})\big)$. In a first step, we simultaneously compute all evaluation nodes $z_{(j-1)Q+k} = \Xi_j(x_k^{\mathrm{ref}}) \in \mathbb{R}^2$ for $j = 1, \dots, M$ and $Q = 7$ the order of the quadrature. This leads to some $(MQ \times 2)$-matrix `z` (Lines 21–25). All evaluations of f are performed simultaneously (Line 28), where we assume that the corresponding MATLAB function takes the matrix `z` and returns the $(MQ \times 1)$-column vector `fz` of the corresponding function values. This vector is reshaped into an $(Q \times M)$-matrix such that the j-th column contains the evaluations corresponding to T_j, i.e., $\texttt{fz}(k,j) = f\big(\Xi_j(x_k^{\mathrm{ref}})\big)$.
- In Line 31, we compute the $(1 \times M)$-row vector `f` of the approximate integral means, cf. (8.14).

- Finally, the function returns the column vector $\mathbf{v} \in \mathbb{R}^M$ with $\mathbf{v}_j = \widetilde{\mathrm{osc}}_{\Omega,\ell}(T_j)^2$, cf. (8.15), which is computed in Line 34, as well as the $(M \times 1)$ column vector $\mathbf{f} =$ `fh` of the elementwise integral means (Line 37).

LISTING 29. Build RHS for Symm's IE with Non-Homogeneous Volume Force

```
1 function b = buildSymmVolRHS(coordinates,elements,uDh,vertices,triangles,fh)
2 %*** compute RHS vector for homogeneous volume force
3 b = buildSymmRHS(coordinates,elements,uDh);
4
5 %*** compute N-matrix for P0(Gamma) x P0(Omega)
6 N = buildN(coordinates,elements,vertices,triangles);
7
8 %*** return RHS vector for non-homogeneous volume force
9 b = b - N*fh;
```

8.4. Building of Right-Hand Side Vector for Symm's IE (Listing 29). Equation (8.8) states that the right-hand side of Symm's equation with non-homogeneous volume force is given by

$$\mathbf{b} = \Big(\mathbf{K} + \frac{1}{2}\mathbf{M}\Big)\mathbf{g} - \mathbf{N}\mathbf{f}. \tag{8.16}$$

The algorithm of Listing 29 computes the vector $\mathbf{b}$. In view of Section 5.4, it essentially remains to compute the contribution $-\mathbf{N}\mathbf{f}$ and add it to the right-hand side for Symm's integral equation with homogeneous volume force. A description of the function **`buildSymmVolRHS`** thus is the following.

- The function **`buildSymmVolRHS`** takes as input the boundary mesh $\mathcal{E}_\ell$ in form of the arrays `coordinates` and `elements` as well as the volume mesh $\mathcal{T}_\ell$ in terms of `vertices` and `triangles`. The column vector $\mathbf{f} =$`fh` contains the $\mathcal{T}_\ell$-elementwise values of F_ℓ.
- First, the right-hand side of Symm's integral equation with homogeneous volume force is computed (Line 3).
- The Newton potential matrix $\mathbf{N}$ is built in Line 6.
- Finally, we realize Equation (8.16) in Line 9 and return the vector $\mathbf{b}$.

8.5. A Posteriori Error Estimate for Symm's IE. It is shown in [6] that the error can, for instance, be estimated by

$$\|\phi - \Phi_\ell\|_V \approx \eta_{V,\ell} + \mathrm{osc}_{D,\ell} + \mathrm{osc}_{\Omega,\ell} \sim (\eta_{V,\ell}^2 + \mathrm{osc}_{D,\ell}^2 + \mathrm{osc}_{\Omega,\ell}^2)^{1/2} =: \varrho_\ell, \tag{8.17}$$

where $\eta_{V,\ell}$ can be replaced by any other error estimator for Symm's integral equation, e.g., by the local error estimator $\widetilde{\mu}_{V,\ell}$, see Section 5.6. We stress that the upper bound for $\|\phi - \Phi_\ell\|_V$ in (8.17) is mathematically only guaranteed under a saturation assumption for the non-perturbed problem, whereas a lower bound of the type

$$\varrho_\ell \lesssim \|\phi - \Phi_\ell\|_V + \mathrm{osc}_{D,\ell} + \mathrm{osc}_{\Omega,\ell} \tag{8.18}$$

holds in general.

LISTING 30. Adaptive Algorithm for Symm's IE with Non-Homogeneous Volume Force

```
1 % adaptiveSymmVol provides the implementation of an adaptive mesh-refining
2 % algorithm for Symm's integral equation with volume force.
3 addpath('../../lib');
4
5 %*** use lshape2 for demonstration purposes
6 addpath('lshape2/');
7
```

```
vertices = load('coordinates.dat')/4;
triangles = load('triangles.dat');

%*** rotate domain Omega so that exact solution is symmetric
alpha = 3*pi/4;
vertices = vertices*[cos(alpha) -sin(alpha);sin(alpha) cos(alpha)]';

%*** maximal number of elements
nEmax = 200;

%*** adaptivity parameter
theta = 0.25;
rho = 0.25;

%*** extract boundary mesh
[vertices,triangles,coordinates,elements] = ...
    buildSortedMesh(vertices,triangles);

%*** adaptive mesh-refining algorithm
while 1

    fprintf('number of elements: N = %d (Gamma), %d (Omega)\r', ...
        size(elements,1),size(triangles,1));

    %*** build uniformly refined mesh
    [coordinates_fine,elements_fine,father2son] ...
        = refineBoundaryMesh(coordinates,elements);

    %*** discretize Dirichlet and volume data and compute data oscillations
    [osc_fine,uDh_fine] = computeOscDirichlet(coordinates_fine,elements_fine,@g);
    [oscV,fh] = computeOscVolume(vertices,triangles,@f);
    osc = osc_fine(father2son(:,1)) + osc_fine(father2son(:,2));

    %*** compute fine-mesh solution
    V_fine = buildV(coordinates_fine,elements_fine);
    b_fine = buildSymmVolRHS(coordinates_fine,elements_fine,uDh_fine, ...
        vertices,triangles,fh);
    x_fine = V_fine\b_fine;

    %*** stopping criterion
    if (size(elements,1) + size(triangles,1)) > nEmax
      break;
    end

    %*** compute (h-h/2)-error estimator tilde-mu
    mu_tilde = computeEstSlpMuTilde(coordinates,elements,father2son,x_fine);

    %*** mark elements for refinement
    [marked_elements,marked_triangles] = markElements(theta,rho, ...
        mu_tilde + osc,oscV);

    %*** generate new mesh
    [vertices,triangles,elements] ...
        = refineMesh(vertices,triangles,elements,marked_triangles,marked_elements);
    coordinates = vertices(unique(elements),:);
end

```

```
%*** visualize exact and adaptively computed solution
plotArclengthP0(coordinates_fine,elements_fine,x_fine,@phi,1);

%*** visualize exact and approximated Dirichlet data
plotArclengthS1(coordinates_fine,elements_fine,uDh_fine,@g,2);
```

8.6. Adaptive Mesh-Refinement for Symm's IE. For the presentation, we restrict to the error estimator $\widetilde{\mu}_\ell$ from Section 5.6.4. For $\tau \in \mathcal{E}_\ell \cup \mathcal{T}_\ell$, we define

$$\widetilde{\varrho}_\ell(\tau)^2 := \begin{cases} \widetilde{\mu}_{V,\ell}(\tau)^2 + \operatorname{osc}_{D,\ell}(\tau)^2 & \text{for } \tau = E \in \mathcal{E}_\ell, \\ \operatorname{osc}_{\Omega,\ell}(\tau)^2 & \text{for } \tau = T \in \mathcal{T}_\ell. \end{cases} \tag{8.19}$$

Note that there holds

$$\widetilde{\varrho}_\ell^2 = \widetilde{\mu}_{V,\ell}^2 + \operatorname{osc}_{D,\ell}^2 + \operatorname{osc}_{\Omega,\ell}^2 = \sum_{\tau \in \mathcal{E}_\ell \cup \mathcal{T}_\ell} \widetilde{\varrho}_\ell(\tau)^2. \tag{8.20}$$

As stated before, we found it essential for stability of $\mathbf{N}$ to ensure $\mathcal{E}_\ell = \mathcal{T}_\ell|_\Gamma$. Then, the usual adaptive algorithm takes the following form:

Input: Initial volume mesh $\mathcal{T}_0$, Dirichlet data g, volume force f, adaptivity parameter $0 < \theta < 1$, maximal number $M_{\max} \in \mathbb{N}$ of volume elements, and counter $\ell = 0$.

(i) Build boundary mesh $\mathcal{E}_\ell := \mathcal{T}_\ell|_\Gamma$ and uniform refinement $\widehat{\mathcal{E}}_\ell$.
(ii) Compute Galerkin solution $\widehat{\Phi}_\ell \in \mathcal{P}^0(\widehat{\mathcal{E}}_\ell)$.
(iii) Compute refinement indicators $\widetilde{\varrho}_\ell(\tau)^2$ from (8.19) for all $\tau \in \mathcal{E}_\ell \cup \mathcal{T}_\ell$.
(iv) Find minimal set $\mathcal{M}_\ell \subseteq \mathcal{E}_\ell \cup \mathcal{T}_\ell$ such that

$$\theta\, \widetilde{\varrho}_\ell^2 = \theta \sum_{\tau \in \mathcal{E}_\ell \cup \mathcal{T}_\ell} \widetilde{\varrho}_\ell(\tau)^2 \le \sum_{\tau \in \mathcal{M}_\ell} \widetilde{\varrho}_\ell(\tau)^2. \tag{8.21}$$

(v) For each marked boundary element $E \in \mathcal{M}_\ell \cap \mathcal{E}_\ell$, mark the corresponding edge of the unique triangle $T \in \mathcal{T}_\ell$ with $E \subset \partial T$.
(vi) For each marked triangle $T \in \mathcal{M}_\ell \cap \mathcal{T}_\ell$, mark its reference edge.
(vii) Use newest vertex bisection to halve at least all marked edges and to generate a new volume mesh $\mathcal{T}_{\ell+1}$
(viii) Stop provided that $\#\mathcal{T}_{\ell+1} \ge M_{\max}$; otherwise, increase counter $\ell \mapsto \ell + 1$ and go to (i).

Output: Adaptively generated boundary mesh $\widehat{\mathcal{E}}_\ell$ and volume mesh $\mathcal{T}_\ell$ and corresponding discrete solution $\widehat{\Phi}_\ell \in \mathcal{P}^0(\widehat{\mathcal{E}}_\ell)$.

The MATLAB script of Listing 30 realizes this adaptive algorithm. We refer to section 5.7.1 for further details.

8.7. Hypersingular Integral Equation with Volume Forces. Next, we consider the Neumann problem

$$-\Delta u = f \text{ in } \Omega \quad \text{with} \quad \partial_n u = \phi \text{ on } \Gamma = \partial\Omega, \tag{8.22}$$

where the given data now have to satisfy the compatibility condition

$$\int_\Omega f\, dx + \int_\Gamma \phi\, d\Gamma = 0 \tag{8.23}$$

according to the Gauss divergence theorem. With the normal derivative $N_1 := \partial_n \widetilde{N}$ of the Newtonian potential, the differential equation (8.22) is equivalently stated by

$$Wu = (1/2 - K')\phi - N_1 f \quad \text{on } \Gamma. \tag{8.24}$$

We now adopt the notation of Section 6. Then, this integral equation is equivalently stated in variational form as

$$\langle\!\langle u\,,v\rangle\!\rangle_{W+S} = \langle(1/2-K')\phi\,,\,v\rangle_\Gamma - \langle N_1 f\,,\,v\rangle_\Gamma \quad \text{for all } v\in H^{1/2}(\Gamma), \tag{8.25}$$

and the unique solution $u\in H^{1/2}(\Gamma)$ of which automatically satisfies $\int_\Gamma u\,d\Gamma = 0$, i.e., $u\in H^{1/2}_*(\Gamma)$. Moreover, it is a consequence of the Calderón system that N_1 satisfies the operator equation

$$N_1 = (-1/2+K')V^{-1}N_0 \tag{8.26}$$

with the trace N_0 of the Newtonian potential, see e.g. [31, Lemma 6.20]. We can therefore split the variational formulation (8.25) into two steps: First, we compute $\lambda = V^{-1}N_0 f\in H^{-1/2}(\Gamma)$ by solving the (equivalent) variational form

$$\langle V\lambda\,,\,\psi\rangle_\Gamma = \langle N_0 f\,,\,\psi\rangle_\Gamma \quad \text{for all } \psi\in H^{-1/2}(\Gamma). \tag{8.27}$$

In particular, there holds $-N_1 f = (1/2-K')\lambda$. Second, we compute the solution $u\in H^{1/2}_*(\Gamma)$ of (8.25) by solving the variational form

$$\langle\!\langle u\,,v\rangle\!\rangle_{W+S} = \langle(1/2-K')(\phi+\lambda)\,,\,v\rangle_\Gamma \quad \text{for all } v\in H^{1/2}(\Gamma). \tag{8.28}$$

To solve (8.25) numerically, we now discretize (8.27)–(8.28): First, we replace f by $F_\ell\in\mathcal{P}^0(\mathcal{T}_\ell)$ and seek the unique Galerkin solution $\Lambda_\ell\in\mathcal{P}^0(\mathcal{E}_\ell)$ of

$$\langle V\Lambda_\ell\,,\,\Psi_\ell\rangle_\Gamma = \langle N_0 F_\ell\,,\,\Psi_\ell\rangle_\Gamma \quad \text{for all } \Psi_\ell\in\mathcal{P}^0(\mathcal{E}_\ell). \tag{8.29}$$

Second, we seek the unique Galerkin solution $U_\ell\in\mathcal{S}^1(\mathcal{E}_\ell)$ of

$$\langle\!\langle U_\ell\,,V_\ell\rangle\!\rangle_{W+S} = \langle(1/2-K')(\Phi_\ell+\Lambda_\ell)\,,\,V_\ell\rangle_\Gamma \quad \text{for all } V_\ell\in\mathcal{S}^1(\mathcal{E}_\ell), \tag{8.30}$$

where $\Phi_\ell = \Pi_\ell\phi\in\mathcal{P}^0(\mathcal{E}_\ell)$ denotes the L^2-projected Neumann data, cf. Section 6. We stress that this procedure avoids to implement the matrix corresponding to N_1. Moreover, in [28] it is even empirically observed that this approach is more effective with respect to the computational time. Note that (8.29) is equivalent to the linear system

$$\mathbf{V}\mathbf{y} = \mathbf{N}\mathbf{f}, \tag{8.31}$$

where $\mathbf{f}\in\mathbb{R}^M$ is the coefficient vector of F_ℓ from (8.3) and where the unique solution $\mathbf{y}\in\mathbb{R}^N$ is the coefficient vector of

$$\Lambda_\ell = \sum_{j=1}^N \mathbf{y}_j\chi_{E_j}. \tag{8.32}$$

Moreover, (8.30) is equivalent to the linear system

$$(\mathbf{W}+\mathbf{S})\mathbf{x} = \Big(\frac{1}{2}\mathbf{M}^T - \mathbf{K}^T\Big)(\mathbf{p}+\mathbf{y}), \tag{8.33}$$

cf. (6.11) in case of $f=0$. In comparison with Section 6, the right-hand side in (8.33) has an additional term $+\mathbf{y}$, and we have to solve for $\mathbf{y}$ in a preprocessing step.

LISTING 31. Build RHS for Hypersingular IE with Non-Homogeneous Volume Force

```
function [b,lambdah] = buildHypsingVolRHS(coordinates,elements,phih, ...
                                                    vertices,triangles,fh)
%*** compute N-matrix for P0(Gamma) x P0(Omega)
N = buildN(coordinates,elements,vertices,triangles);

%*** compute N*fh and free unnecessary memory
Nfh = N*fh;
clear N;

%*** compute SLP matrix
```

```
11 V = buildV(coordinates,elements);
12
13 %*** solve Symm's IE for the computation of N_1*f
14 lambdah = V\Nfh;
15 clear V;
16
17 %*** compute DLP matrix for P0 x S1
18 K = buildK(coordinates,elements);
19
20 %*** compute mass-type matrix for P0 x S1
21 M = buildM(coordinates,elements);
22
23 %*** build right-hand side vector
24 phih = phih + lambdah;
25 b = (phih'*M*0.5 - phih'*K)';
```

8.8. Building of Right-Hand Side Vector for Hypersingular IE (Listing 31). Equation (8.33) states that the right-hand side of the hypersingular integral equation with non-homogeneous volume force is given by

$$\mathbf{b} = \Big(\frac{1}{2}\,\mathbf{M}^T - \mathbf{K}^T\Big)(\mathbf{p}+\mathbf{y}) = \Big((\mathbf{p}+\mathbf{y})^T\big(\frac{1}{2}\,\mathbf{M} - \mathbf{K}\big)\Big)^T. \tag{8.34}$$

Here, $\mathbf{p}$ is the coefficient vector of the L^2-projection $\Phi_\ell \in \mathcal{P}^0(\mathcal{E}_\ell)$ of the Neumann data $\phi \in H_*^{-1/2}(\Gamma)$, see (6.6), and the vector $\mathbf{y}$ is the coefficient vector of the solution $\Lambda_\ell \in \mathcal{P}^0(\mathcal{E}_\ell)$ of (8.31). A description of the function **`buildHypsingVolRHS`** reads as follows:

- The function **`buildHypsingVolRHS`** takes as input the boundary mesh $\mathcal{E}_\ell$ in terms of `coordinates` and `elements` as well as the volume mesh $\mathcal{T}_\ell$ in terms of `vertices` and `triangles`. The $(N \times 1)$-column vector $\mathbf{p}$ = `phih` provides the elementwise values of $\Phi_\ell \in \mathcal{P}^0(\mathcal{E}_\ell)$, the $(M \times 1)$-column vector $\mathbf{f}$ = `fh` provides the elementwise values of $F_\ell \in \mathcal{P}^0(\mathcal{T}_\ell)$.
- We first build the Newton potential matrix $\mathbf{N}$ (Line 4) and compute the right-hand side vector $\mathbf{Nf}$ to compute Λ_ℓ, cf. (8.29) (Line 7). This is the only use of $\mathbf{N}$, and the memory can be set free (Line 8);
- Next, we build the simple-layer potential matrix $\mathbf{V}$ (Line 11) and compute the coefficient vector $\mathbf{y}$ = `lambdah` of Φ_ℓ (Line 14). This is the only use of $\mathbf{V}$, and the memory can be set free (Line 15).
- We build the double-layer potential matrix $\mathbf{K}$ (Line 18) and the mass-type matrix $\mathbf{M}$ (Line 21), cf. Section 5.3.
- Finally, we overwrite $\mathbf{p}$ by $\mathbf{p}+\mathbf{y}$ (Line 24) and thus realize Equation (8.34) in Line 25.

Contrary to our implementation of **`buildSymmVolRHS`** in Listing 29, the implementation of **`buildHypsingVolRHS`** avoids the use of **`buildHypsingRHS`**. The reason for this is simply that otherwise the matrices $\mathbf{K}$ and $\mathbf{M}$ would have been computed twice.

8.9. A Posteriori Error Estimate for Hypersingular IE. It is shown in [5] resp. [6] that the error in λ can, for instance, be estimated by

$$\|\lambda - \Lambda_\ell\|_V \approx \eta_{V,\ell} + \mathrm{osc}_{\Omega,\ell}, \tag{8.35}$$

where the error estimator $\eta_{V,\ell} = \|\widehat{\lambda}_\ell - \lambda_\ell\|_V$ from Section 5.6.1 measures the discretization error for λ. Moreover, we show in [6] that the error in u can be estimated by

$$\|u - U_\ell\|_W \approx \eta_{W,\ell} + \mathrm{osc}_{N,\ell} + \|\lambda - \Lambda_\ell\|_V, \tag{8.36}$$

where the error estimator $\eta_{W,\ell} = |||\widehat{U}_\ell - U_\ell|||_W$ from Section 6.6.1 measures the discretization error for u. Finally, this leads to the overall error estimate

$$|||u - U_\ell|||_W + |||\lambda - \Lambda_\ell|||_V \approx \eta_{W,\ell} + \eta_{V,\ell} + \mathrm{osc}_{N,\ell} + \mathrm{osc}_{\Omega,\ell}. \tag{8.37}$$

As above, the error estimator $\eta_{V,\ell}$ can be replaced by any other error estimator for Symm's integral equation, and $\eta_{W,\ell}$ can be replaced by any other error estimator for the hypersingular integral equation, see Section 5.6 and Section 6.6, respectively. We stress that the upper bound in the error estimate (8.37) depends on a saturation assumption for both u and λ, whereas the lower bound

$$\begin{aligned} \varrho_\ell &:= \big(\eta_{W,\ell}^2 + \eta_{V,\ell}^2 + \mathrm{osc}_{N,\ell}^2 + \mathrm{osc}_{\Omega,\ell}^2\big)^{1/2} \\ &\lesssim |||u - U_\ell|||_W + |||\lambda - \Lambda_\ell|||_V + \mathrm{osc}_{N,\ell} + \mathrm{osc}_{\Omega,\ell} \end{aligned} \tag{8.38}$$

holds in general.

LISTING 32. Adaptive Algorithm for Hypersingular IE with Non-Homogeneous Volume Force

```
% adaptiveHypsingVol provides the implementation of an adaptive mesh-refining
% algorithm for the hypersingular integral equation with volume force.

%*** use lshape2 for demonstration purposes
addpath('lshape2/');

vertices = load('coordinates.dat')/4;
triangles = load('triangles.dat');

%*** rotate domain Omega so that exact solution is symmetric
alpha = 3*pi/4;
vertices = vertices*[cos(alpha) -sin(alpha);sin(alpha) cos(alpha)]';

%*** extract boundary mesh
[vertices,triangles,coordinates,elements] = buildSortedMesh(vertices,triangles);

%*** maximal number of elements
nEmax = 200;

%*** adaptivity parameter
theta = 0.25;
rho = 0.25;

%*** adaptive mesh-refining algorithm
while 1

    fprintf('number of elements: N = %d (Gamma), %d (Omega)\r', ...
        size(elements,1),size(triangles,1));

    %*** build uniformly refined mesh
    [coordinates_fine,elements_fine,father2son] ...
        = refineBoundaryMesh(coordinates,elements);

    %*** discretize Neumann and volume data and compute data oscillations
    [osc_fine,phih_fine] ...
        = computeOscNeumann(coordinates_fine,elements_fine,@phi);
    [oscV,fh] = computeOscVolume(vertices,triangles,@f);
    oscN = osc_fine(father2son(:,1)) + osc_fine(father2son(:,2));

    %*** compute fine-mesh solution
```

```
    W_fine = buildW(coordinates_fine,elements_fine) ...
             + buildHypsingStabilization(coordinates_fine,elements_fine);
    [b_fine,y_fine] = buildHypsingVolRHS(coordinates_fine,elements_fine, ...
        phih_fine,vertices,triangles,fh);
    x_fine = W_fine\b_fine;

    %*** stopping criterion
    if ( size(elements,1) + size(triangles,1) ) > nEmax
      break;
    end

    %*** compute (h-h/2)-error estimator tilde-mu
    mu_tilde_ = computeEstHypMuTilde(elements_fine,elements,father2son,...
                                     x_fine);

    %*** compute (h-h/2)-error estimator tilde-mu for Symm's equation to
    %*** measure error induced within buildHypsingVolRHS
    mu_V_tilde_ = computeEstSlpMuTilde(coordinates,elements,father2son,y_fine);

    %*** mark elements for refinement
    [marked_elements,marked_triangles] ...
        = markElements(theta,mu_tilde_+mu_V_tilde_+oscN,oscV);

    %*** generate new mesh
    [vertices,triangles,elements] ...
        = refineMesh(vertices,triangles,elements,marked_triangles,marked_elements);
    coordinates = vertices(unique(elements),:);
end

%*** visualize exact and adaptively computed solution
plotArclengthS1(coordinates_fine,elements_fine,x_fine,@g,1);

%*** visualize exact and approximated Neumann data
plotArclengthP0(coordinates_fine,elements_fine,phih_fine,@phi,2);
```

8.10. Adaptive Mesh-Refinement for Hypersingular IE. We now give an adaptive algorithm for the hypersingular equation with non-homogeneous volume force. We use the error estimator

$$\widetilde{\mu}_{V,\ell} = \|h_\ell^{1/2}(\widehat{\Lambda}_\ell - \Pi_\ell \Lambda_\ell)\|_{L^2(\Gamma)} \tag{8.39}$$

to control the discretization of the auxiliary problem (8.27) and

$$\widetilde{\mu}_{W,\ell} = \|h_\ell^{1/2}(\widehat{U}_\ell - I_\ell \widehat{U}_\ell)'\|_{L^2(\Gamma)} \tag{8.40}$$

to control the discretization of the hypsingular integral equation (8.28). For $\tau \in \mathcal{E}_\ell \cup \mathcal{T}_\ell$, we define

$$\widetilde{\varrho}_\ell(\tau)^2 := \begin{cases} \widetilde{\mu}_{W,\ell}(\tau)^2 + \widetilde{\mu}_{V,\ell}(\tau)^2 + \mathrm{osc}_{N,\ell}(\tau)^2 & \text{for } \tau = E \in \mathcal{E}_\ell \\ \mathrm{osc}_{\Omega,\ell}(\tau)^2 & \text{for } \tau = T \in \mathcal{T}_\ell. \end{cases} \tag{8.41}$$

There holds

$$\widetilde{\varrho}_\ell^2 = \widetilde{\mu}_{V,\ell}^2 + \widetilde{\mu}_{W,\ell}^2 + \mathrm{osc}_{N,\ell}^2 + \mathrm{osc}_{\Omega,\ell}^2 = \sum_{\tau \in \mathcal{E}_\ell \cup \mathcal{T}_\ell} \widetilde{\varrho}_\ell(T)^2. \tag{8.42}$$

Then, the adaptive algorithm takes the following form:

Input: Initial volume mesh $\mathcal{T}_0$, Neumann data ϕ, volume force f, adaptivity parameter $0 < \theta < 1$, maximal number $M_{max} \in \mathbb{N}$ of volume elements, and counter $\ell = 0$.

(i) Build boundary mesh $\mathcal{E}_\ell = \mathcal{T}_\ell|_\Gamma$ and uniform refinement $\widehat{\mathcal{E}}_\ell$.
(ii) Compute Galerkin solution $\widehat{\Lambda}_\ell \in \mathcal{P}^0(\widehat{\mathcal{E}}_\ell)$ of (8.29)
(iii) Compute Galerkin solution $\widehat{U}_\ell \in \mathcal{S}^1(\widehat{\mathcal{E}}_\ell)$.
(iv) Compute refinement indicators $\widetilde{\varrho}_\ell(\tau)$ from (8.41) for all $\tau \in \mathcal{E}_\ell \cup \mathcal{T}_\ell$.
(v) Find minimal set $\mathcal{M}_\ell \subseteq \mathcal{E}_\ell \cup \mathcal{T}_\ell$ such that

$$\theta\, \widetilde{\varrho}_\ell^2 = \theta \sum_{\tau \in \mathcal{E}_\ell \cup \mathcal{T}_\ell} \widetilde{\varrho}_\ell(\tau)^2 \le \sum_{\tau \in \mathcal{M}_\ell} \widetilde{\varrho}_\ell(\tau)^2. \tag{8.43}$$

(vi) For each marked boundary element $E \in \mathcal{M}_\ell \cap \mathcal{E}_\ell$, mark the corresponding edge of the unique triangle $T \in \mathcal{T}_\ell$ with $E \subset \partial T$.
(vii) For each marked triangle $T \in \mathcal{M}_\ell \cap \mathcal{T}_\ell$, mark its reference edge.
(viii) Use newest vertex bisection to halve at least all marked edges and to generate a new volume mesh $\mathcal{T}_{\ell+1}$
(ix) Stop provided that $\#\mathcal{T}_{\ell+1} \ge M_{\max}$; otherwise, increase counter $\ell \mapsto \ell + 1$ and go to (i).

Output: Adaptively generated boundary mesh $\widehat{\mathcal{E}}_\ell$ and volume mesh $\mathcal{T}_\ell$ and corresponding discrete solution $\widehat{U}_\ell \in \mathcal{S}^1(\widehat{\mathcal{E}}_\ell)$.

The MATLAB script of Listing 32 realizes this adaptive algorithm. We refer to section 6.7.1 for further details, since the algorithms are similar. Nevertheless, let us comment on the additional error estimate for $\widehat{\Lambda}_\ell$:

- The function **`buildHypsingVolRHS`** returns the right hand side vector `b_fine` as well as the vector `y_fine`, which is the discrete solution of the equation (8.31).
- The vector `y_fine` is needed for the computation of the error estimate (8.39) (Line 59). The resulting error estimate `mu_V_tilde` is added to the error indicator for the boundary mesh (Line 63).

8.11. Mixed Problem. Now we consider a mixed problem with non-vanishing volume force

$$\begin{aligned} -\Delta u &= f && \text{in } \Omega, \\ u &= u_D && \text{on } \Gamma_D, \\ u &= \phi_N && \text{on } \Gamma_N. \end{aligned} \tag{8.44}$$

With the definitions of Section 7, the integral formulation of this problem is

$$A \begin{pmatrix} u_N \\ \phi_D \end{pmatrix} = (1/2 - A) \begin{pmatrix} \overline{u}_D \\ \overline{\phi}_N \end{pmatrix} - \begin{pmatrix} N_0 f \\ N_1 f \end{pmatrix} =: \mathcal{F} \quad \text{on } \Gamma_D \times \Gamma_N. \tag{8.45}$$

Here, we have chosen arbitrary but fixed extensions of the data $\overline{u}_D \in H^{1/2}(\Gamma)$ and $\overline{\phi}_N \in H^{-1/2}(\Gamma)$ see Section 7. However, the difference to the integral formulation (7.4) of Section 7 is that the right hand side involves the additional term

$$- \begin{pmatrix} N_0 f \\ N_1 f \end{pmatrix},$$

where $N_0 f$ and $N_1 f$ are the trace and the normal derivative of the Newton potential (1.3). For the computation of the perturbed Galerkin formulation, we first replace $N_0 f$ by $N_0 F_\ell$. To compute an approximation of $N_1 f$, we use the well known identity

$$N_1 = (-1/2 + K')V^{-1} N_0,$$

see [25, 31, 30]. The discrete scheme now reads as follows: first, we solve Symm's equation

$$\langle V \Lambda_\ell \,,\, \Psi_\ell \rangle = \langle N_0 F_\ell \,,\, \Psi_\ell \rangle \quad \text{for all } \Psi_\ell \in \mathcal{P}^0(\mathcal{E}_\ell)$$

to obtain an approximation of $\Lambda_\ell \approx N_1 f$. In the next step, we solve the mixed problem

$$\langle\!\langle \mathbf{U}_\ell \,,\, \mathbf{V}_\ell \rangle\!\rangle_A = \langle \mathcal{F}_\ell \,,\, \mathbf{V}_\ell \rangle_{\mathcal{H}^\star \times \mathcal{H}} \text{ for all } \mathbf{V}_\ell \in X_\ell, \tag{8.46}$$

where

$$\mathcal{F}_\ell := (1/2 - A)\begin{pmatrix} I_\ell u_D \\ \Phi_\ell \phi_N \end{pmatrix} - \begin{pmatrix} N_0 F_\ell \\ (-1/2 + K')\Lambda_\ell. \end{pmatrix}$$

The algorithm for solving a mixed problem with volume force thus takes the following form:

- Input: volume mesh $\mathcal{T}_\ell$, boundary mesh $\mathcal{E}_\ell = \mathcal{T}_\ell|_\Gamma$.
- Compute $\mathbf{g} = U_{D,\ell} = I_\ell \overline{u}_D$, $\mathbf{p} = \Phi_{N,\ell} = \Pi_\ell \overline{\phi}_N$ and $\mathbf{f} = F_\ell = \Pi_\ell f$.
- Solve the linear system

$$\mathbf{V}\mathbf{y} = \mathbf{N}\mathbf{f}. \tag{8.47}$$

- Solve the linear system

$$\begin{pmatrix} \mathbf{W}|_{\Gamma_N \times \Gamma_N} & \mathbf{K}^T|_{\Gamma_N \times \Gamma_D} \\ -\mathbf{K}|_{\Gamma_D \times \Gamma_N} & \mathbf{V}|_{\Gamma_D \times \Gamma_D} \end{pmatrix} \mathbf{x} = \begin{pmatrix} \left(\frac{1}{2}\mathbf{p}^T\mathbf{M} - \mathbf{p}^T\mathbf{K} - \mathbf{g}^T\mathbf{W}\right)^T - (\frac{1}{2}\mathbf{M}^T - \mathbf{K}^T)\mathbf{y}|_{\Gamma_N} \\ \left(\frac{1}{2}\mathbf{M}\mathbf{g} + \mathbf{K}\mathbf{g} - \mathbf{V}\mathbf{p} - \mathbf{N}\mathbf{f}\right)|_{\Gamma_D} \end{pmatrix}. \tag{8.48}$$

LISTING 33. Build RHS Vector for Mixed Problem with Non-Homogeneous Volume Force

```
1  function [b_nodes,b_elements,lambdah] ...
2                  = buildMixedVolRHS(coordinates,dirichlet,neumann,uDh,phiNh, ...
3  elements = [dirichlet;neumann];
4
5  %*** compute N-matrix for P0(Gamma) x P0(Omega)
6  N = buildN(coordinates,elements,vertices,triangles);
7
8  %*** compute N*fh and free unnecessary memory
9  Nfh = N*fh;
10 clear N;
11
12 %*** solve Symm's IE for the computation of N_1*f
13 lambdah = V\Nfh;
14
15 %*** compute mass-type matrix for P0 x S1
16 M = buildM(coordinates,elements);
17
18 %*** compute full right-hand side
19 b_elements = M*uDh*0.5 + K*uDh - V*phiNh-Nfh;
20 phiNh = phiNh + lambdah;
21 b_nodes = (0.5*phiNh'*M - phiNh'*K - uDh'*W)';
```

8.12. Building of Right-Hand Side Vector for Mixed Problem (Listing 33). Equation (8.48) states that the right-hand side of the integral formulation of the mixed boundary value problem is given by

$$\begin{pmatrix} \left(\frac{1}{2}\mathbf{p}^T\mathbf{M} - \mathbf{p}^T\mathbf{K} - \mathbf{g}^T\mathbf{W}\right)^T - (\frac{1}{2}\mathbf{M}^T - \mathbf{K}^T)\mathbf{y}|_{\Gamma_N} \\ \left(\frac{1}{2}\mathbf{M}\mathbf{g} + \mathbf{K}\mathbf{g} - \mathbf{V}\mathbf{p} - \mathbf{N}\mathbf{f}\right)|_{\Gamma_D} \end{pmatrix} = \begin{pmatrix} \left(\frac{1}{2}\mathbf{p}^T\mathbf{M} - \mathbf{p}^T\mathbf{K} - \mathbf{g}^T\mathbf{W}\right)^T|_{\Gamma_N} \\ \left(\frac{1}{2}\mathbf{M}\mathbf{g} + \mathbf{K}\mathbf{g} - \mathbf{V}\mathbf{p}\right)|_{\Gamma_D} \end{pmatrix} - \begin{pmatrix} (\frac{1}{2}\mathbf{M}^T - \mathbf{K}^T)\mathbf{y}|_{\Gamma_N} \\ \mathbf{N}\mathbf{f})|_{\Gamma_D} \end{pmatrix}. \tag{8.49}$$

The first term on the right-hand side is returned by call of **buildMixedRHS**, see Section 7.2. However, we do not use this function here because $\mathbf{M}$ would be built twice. The implementation of **buildMixedVolRHS** reads as follows:

- First, the discrete Newton potential $\mathbf{N}$ is built and multiplied with the discrete volume data (Lines 6 and 9).
- The equation (8.47) is solved in order to obtain $\mathbf{y} =$`lambdah` (Line 13).
- The mass-type matrix $\mathbf{M}$ is built (Line 16).
- We compute the second line of the right-hand side of (8.48) (Line 19).

- (Line 20-21) The first line of the right-hand side of (8.48) is computed.

8.13. A Posteriori Error Estimate for Mixed Problem. In [6], we propose to extend the error estimators of Section 7.6 in order to include the discretization error introduced by solving (8.47). To that end, we define

$$\mu_{V,\ell}^2 = \|h_\ell^{1/2}(1-\Pi_\ell)\widehat{N}_\ell\|_{L^2(\Gamma)}^2.$$

In principle, one can define $\widetilde{\mu}_{V,\ell}$, $\eta_{V,\ell}$, $\widetilde{\eta}_{V,\ell}$ in the same fashion, but for ease of presentation we stick to $\mu_{V,\ell}$ at this place. Then, we obtain at least sixteen different a posteriori error estimators for the mixed problem with non-homogeneous volume force. Just considering $\mu_{V,\ell}$, they read

$$\eta_\ell^2 := \eta_{D,\ell}^2 + \eta_{N,\ell}^2 + \mu_{V,\ell}^2, \qquad \widetilde{\eta}_\ell^2 := \widetilde{\eta}_{D,\ell}^2 + \widetilde{\eta}_{N,\ell}^2 + \mu_{V,\ell}^2,$$
$$\mu_\ell^2 := \mu_{D,\ell}^2 + \mu_{N,\ell}^2 + \mu_{V,\ell}^2, \qquad \widetilde{\mu}_\ell{}^2 := \widetilde{\mu}_{D,\ell}^2 + \widetilde{\mu}_{N,\ell}^2 + \mu_{V,\ell}^2.$$

See also Section 7.6 to recall the definitions of the error estimators. In [6], we prove that under a saturation assumption $\rho_\ell^2 := \widetilde{\mu}_\ell^2 + \mathrm{osc}_\ell^2 + \mathrm{osc}_{\Omega,\ell}^2$ is an upper bound of the error, i.e.,

$$|||\mathbf{u} - \mathbf{U}_\ell|||_A \lesssim \rho_\ell.$$

Here, osc_ℓ from (7.24) are boundary data oscillations and $\mathrm{osc}_{\Omega,\ell}$ from (8.9) are volume data oscillations.

8.14. Adaptive Mesh-Refinement for Mixed Problem.

LISTING 34. Adaptive Algorithm for Mixed Problem with Non-homogeneous Volume Force

```
% adaptiveMixedVol provides the implementation of an adaptive mesh–refining
% algorithm for the symmetric integral formulation of a mixed boundary value
% problem with volume force.
%*** use lshape2 for demonstration purposes
addpath('lshape2/');

vertices = load('coordinates.dat')/4;
triangles = load('triangles.dat');

%*** rotate domain Omega so that exact solution is symmetric
alpha = 3*pi/4;
vertices = vertices*[cos(alpha) –sin(alpha);sin(alpha) cos(alpha)]';

%*** split Gamma into Dirichlet and Neumann boundary
dirichlet = [1 2;2 3;3 6;6 5];
neumann = [5 8;8 7;7 4;4 1];

%*** maximal number of elements
nEmax = 200;

%*** adaptivity parameter
theta = 0.25;
rho = 0.25;

%*** extract boundary mesh
[vertices,triangles,coordinates,neumann,dirichlet] = ...
    buildSortedMesh(vertices,triangles,neumann,dirichlet);

%*** initialize Dirichlet data
uDh = zeros(size(coordinates,1),1);
uDh(unique(dirichlet)) = g(coordinates(unique(dirichlet),:));

```

```
%*** Perform uniform refinement before starting the loop
%*** refine mesh uniformly
[coordinates_fine,dirichlet_fine,neumann_fine,...
    father2dirichlet_fine,father2neumann_fine] ...
    = refineBoundaryMesh(coordinates,dirichlet,neumann);

%*** rearrange indices such that Neumann nodes are first
[coordinates_fine,neumann_fine,dirichlet_fine] = ...
    buildSortedMesh(coordinates_fine,neumann_fine,dirichlet_fine);

%*** discretize data and compute corresponding data oscillations for fine mesh
[oscD_fine,oscN_fine,uDh_fine,phiNh_fine] ...
        = computeOscMixed(coordinates_fine,dirichlet_fine,neumann_fine, ...
                          father2neumann_fine,neumann,uDh,@g,@phi);
[oscV,fh] = computeOscVolume(vertices,triangles,@f);
oscD = sum(oscD_fine(father2dirichlet_fine),2);
oscN = sum(oscN_fine(father2neumann_fine),2);

%*** adaptive mesh-refining algorithm
while 1

    fprintf('number of elements: N = %d (Gamma), %d (Omega)\r', ...
        size(neumann,1)+size(dirichlet,1),size(triangles,1));

    %*** compute integral operators for fine mesh
    elements_fine = [dirichlet_fine;neumann_fine];
    V_fine = buildV(coordinates_fine,elements_fine);
    K_fine = buildK(coordinates_fine,elements_fine);
    W_fine = buildW(coordinates_fine,elements_fine);

    %*** compute right-hand side for fine mesh
    [b_nodes_fine,b_elements_fine,Nell_fine] = buildMixedVolRHS( ...
        coordinates_fine,dirichlet_fine,neumann_fine,uDh_fine,phiNh_fine, ...
    vertices,triangles,fh,V_fine,K_fine,W_fine);

    %*** compute degrees of freedom for fine mesh
    nC_fine = size(coordinates_fine,1);
    nD_fine = size(dirichlet_fine,1);
    freeNeumann_fine = setdiff(1:nC_fine,unique(dirichlet_fine));
    freeDirichlet_fine = 1:nD_fine;
    nN_fine = length(freeNeumann_fine);

    %*** shrink integral operators and right-hand side
    W_fine = W_fine(freeNeumann_fine,freeNeumann_fine);
    K_fine = K_fine(freeDirichlet_fine,freeNeumann_fine);
    V_fine = V_fine(freeDirichlet_fine,freeDirichlet_fine);
    b_nodes_fine = b_nodes_fine(freeNeumann_fine);
    b_elements_fine = b_elements_fine(freeDirichlet_fine);

    %*** compute Galerkin solution for fine mesh
    x = [ W_fine K_fine' ; -K_fine V_fine ] \ [ b_nodes_fine ; b_elements_fine ];

    %*** compute coefficient vectors w.r.t. S1(GammaN) and P0(GammaD)
    xN_fine = zeros(nC_fine,1);
    xN_fine(freeNeumann_fine) = x(1:nN_fine);      %** dof on Neumann boundary
    xD_fine = x((1:nD_fine) + nN_fine);            %** dof on Dirichlet boundary

```

```
90      %*** stopping criterion + size(triangles,1)
91      if (size(neumann,1) + size(dirichlet,1) + size(triangles,1) > nEmax )
92        break;
93      end
94
95      %*** compute error estimates
96      muD_tilde = computeEstSlpMuTilde(coordinates,dirichlet,father2dirichlet_fine, ...
97                                       xD_fine);
98      muN_tilde = computeEstHypMuTilde(neumann_fine,neumann,father2neumann_fine, ...
99                                       xN_fine);
100     mu1_tilde_D = computeEstSlpMuTilde(coordinates,dirichlet, ...
101         father2dirichlet_fine,Nell_fine(1:nD_fine));
102     mu1_tilde_N = computeEstSlpMuTilde(coordinates,neumann, ...
103         father2neumann_fine,Nell_fine(nD_fine+1:end));
104
105     %*** mark elements for refinement
106     [marked_dirichlet,marked_neumann,marked_triangles] ...
107         = markElements(theta,muD_tilde + oscD + mu1_tilde_D, ...
108         muN_tilde + oscN + mu1_tilde_N,oscV);
109
110     %*** generate new mesh
111     [vertices,triangles,dirichlet,neumann,father2triangles, ...
112         father2dirichlet,father2neumann_adap] ...
113         = refineMesh(vertices,triangles,dirichlet,neumann,marked_triangles, ...
114                  marked_dirichlet,marked_neumann);
115
116     %*** rearrange indices such that Neumann nodes are first
117     [vertices,triangles,coordinates,neumann,dirichlet] = ...
118         buildSortedMesh(vertices,triangles,neumann,dirichlet);
119
120     %*** generate fine mesh
121     neumann_coarse = neumann_fine;
122     father2neumann_coarse = father2neumann_fine;
123     [coordinates_fine,dirichlet_fine,neumann_fine,father2dirichlet_fine, ...
124         father2neumann_fine] = refineBoundaryMesh(coordinates, ...
125                                                   dirichlet,neumann);
126
127     %*** rearrange indices such that Neumann nodes are first
128     [coordinates_fine,neumann_fine,dirichlet_fine] = ...
129         buildSortedMesh(coordinates_fine,neumann_fine,dirichlet_fine);
130
131     %*** build coarse2fine array
132     coarse2fine = generateFather2Son(father2neumann_adap, ...
133                                      father2neumann_coarse, ...
134                                      father2neumann_fine);
135
136     %*** discretize data and compute corresponding data oscillations for
137     %*** fine mesh
138     [oscD_fine,oscN_fine,uDh_fine,phiNh_fine] ...
139         = computeOscMixed(coordinates_fine,dirichlet_fine,neumann_fine, ...
140                           coarse2fine,neumann_coarse,uDh_fine,@g,@phi);
141     [oscV,fh] = computeOscVolume(vertices,triangles,@f);
142     oscD = sum(oscD_fine(father2dirichlet_fine),2);
143     oscN = sum(oscN_fine(father2neumann_fine),2);
144 end
145
146 %*** xN_fine = dof on Neumann boundary, i.e. update for Dirichlet data
```

```
%*** to obtain approximation uh of trace of PDE solution u
uh =  xN_fine+uDh_fine;

%*** plot approximate vs exact trace
plotArclengthS1(coordinates_fine,[dirichlet_fine;neumann_fine],uh,@g,2)

%*** xD_fine = dof on Dirichlet boundary, i.e. update for Neumann data
%*** to obtain approximation phih of normal derivative of PDE solution u
phih = [xD_fine ; phiNh_fine(nD_fine+1:end)];

%*** plot approximate vs exact normal derivative
plotArclengthP0(coordinates_fine,[dirichlet_fine;neumann_fine],phih,@phi,1)
```

8.14.1. Implementation of Adaptive Algorithm (Listing 34). The MATLAB script of Listing 34 realizes the adaptive algorithm from the beginning of this section. We refer to section 7.7.1 for further details.

9. New Features in HILBERT (Release 3)

9.1. Evaluation of Simple-Layer and Double-Layer Potential. Evaluation of the simple-layer potential $\widetilde{V}$ and the double-layer potential $\widetilde{K}$, c.f. (1.3), for lowest-order functions, i.e., piecewise constant functions in case of $\widetilde{V}$ and piecewise affine, globally continuous functions in case of $\widetilde{K}$, and abritary evaluation points in $\mathbb{R}^2$ is provided by the following MEX-functions:

- `Vphi_x = evaluateV(coordinates, elements, phih, x [, eta]);`
- `Kg_x = evaluateK(coordinates, elements, gh, x [, eta]);`

As usual, `coordinates` and `elements` describe a boundary mesh $\mathcal{E}_\ell$ with nodes $\mathcal{K}_\ell$. The $|\mathcal{E}_\ell| \times 1$ matrix `phih` and the $|\mathcal{K}_\ell| \times 1$ matrix `gh` describe functions $\Phi_\ell \in \mathcal{P}^0(\mathcal{E}_\ell)$ and $G_\ell \in \mathcal{S}^1(\mathcal{E}_\ell)$ respectively, containing their coefficients with respect to the canonical bases of $\mathcal{P}^0(\mathcal{E}_\ell)$ and $\mathcal{S}^1(\mathcal{E}_\ell)$, i.e., $\Phi_\ell = \sum_{i=1}^{|\mathcal{E}_\ell|} \texttt{phih}(i)\, \chi_i$ and $G_\ell = \sum_{i=1}^{|\mathcal{K}_\ell|} \texttt{gh}(i)\, \zeta_i$. The $M \times 2$ matrix `x` contains a total of M evaluation points, one per row. Both functions return an $M \times 1$ vector. The j-th entry contains the value of the potential $\widetilde{V}\Phi_\ell$, or $\widetilde{K}G_\ell$, respectively, evaluated at the j-th evaluation point $x_j = \texttt{x}(j,:)$.

For our implementation, we write the respective integrals over Γ as sum of integrals over the individual boundary elements of the triangulation $\mathcal{E}_\ell$. To increase numerical stability, we employ the same techniques as for the integral operator matrices $\mathbf{V}$ and $\mathbf{K}$. Let $\eta \geq 0$ be fixed. We adapt the admissibility criterion (5.18) and call an evaluation point x_j and a boundary element E_k admissible if

$$\operatorname{diam}(E_k) < \operatorname{dist}(x_j, E_k), \tag{9.1}$$

where $\operatorname{dist}(x_j, E_k)$ denotes the distance between x_j and E_k. If an element and an evaluation point are admissible, we compute the corresponding integral using Gaussian quadrature, otherwise we use analytical formulas.

In case the optional parameter `eta` is specified, we set $\eta = \texttt{eta}$, otherwise we choose $\eta = 1/2$. For $\eta = 0$, all integrals are computed analytically. The default value for η can be changed by recompiling `evaluateV.c` and `evaluateK.c` with different values for the preprocessor constant `DEFAULT_ETA`. For Gaussian quadrature, we use a 16-point rule by default. This can be changed by recompiling `evaluateV.c` and `evaluateK.c` with different values for the preprocessor constant `GAUSS_ORDER`. Possible values include $2, 4, 8, 16$ and 32.

9.2. Evaluation of Adjoint Double-Layer Potential and Hypersingular Integral Operator. Evaluation of the adjoint double-layer potential K' and the hypersingular integral

operator W for lowest-order functions, i.e., piecewise constant functions in case of K' and piecewise affine, globally continuous functions in case of W, and evaluation points almost everywhere on the boundary Γ, is provided by the following MEX-functions:

- `Kaphi_x = evaluateKadj(coordinates, elements, gh, x, n_x [,eta])`
- `Wg_x = evaluateW(coordinates, elements gh, x, n_x [,eta])`

As usual, `coordinates` and `elements` describe a boundary mesh $\mathcal{E}_\ell$ with nodes $\mathcal{K}_\ell$. The $|\mathcal{E}_\ell| \times 1$ matrix `phih` and the $|\mathcal{K}_\ell| \times 1$ matrix `gh` describe functions $\Phi_\ell \in \mathcal{P}^0(\mathcal{E}_\ell)$ and $G_\ell \in \mathcal{S}^1(\mathcal{E}_\ell)$ respectively, containing their coefficients with respect to the canonical bases of $\mathcal{P}^0(\mathcal{E}_\ell)$ and $\mathcal{S}^1(\mathcal{E}_\ell)$, i.e. $\Phi_\ell = \sum_{i=1}^{|\mathcal{E}_\ell|} \texttt{phih}(i)\, \chi_i$ and $G_\ell = \sum_{i=1}^{|\mathcal{K}_\ell|} \texttt{gh}(i)\, \zeta_i$. The $M \times 2$ matrix `x` contains a total of M evaluation points, one per row. The hypersingular integral operator can be evaluated on $\Gamma \backslash \mathcal{K}_\ell$, the adjoint double-layer potential everywhere on Γ except for the corner points of Γ. The $M \times 2$ vector `n_x` contains a list of normal vectors. The j-th row of `n_x` contains the outer normal vector at the evaluation point `x(j,:)`. Both functions return an $M \times 1$ vector. The j-th entry contains the value of $K'\Phi_\ell$, or WG_ℓ, respectively, evaluated at the j-th evaluation point $x_j = \texttt{x}(j,:)$.

The hypersingular integral operator, which is defined as the negative normal derivative of the double-layer potential, formally reads

$$Wg(x) = -\frac{1}{2\pi} \lim_{\varepsilon \to 0} \int_{\Gamma, |y-x| \geq \varepsilon} \left(\frac{n_x \cdot n_y}{|x-y|^2} - 2 \frac{(x-y) \cdot n_x \, (x-y) \cdot n_y}{|x-y|^4} \right) g(y) \, d\Gamma.$$

Cauchy's principal value does not exist for the integral above, but if g is continuous, we may employ regularization techniques as described in [31, Section 6.5] for the case that Γ is closed. We use similar techniques in case that Γ is open. For our implementation, we write the regularized integral over Γ as sum of integrals over the individual boundary elements $E \in \mathcal{E}_\ell$. For the evaluation of integrals over an element E with $x \in E = [z_j, z_k]$, it is essential that x is in the interior of E, i.e. $x \neq z_j$ and $x \neq z_k$.

To increase numerical stability, we employ the same techniques as for the evaluation of the simple-layer potential $\widetilde{V}$ and the double-layer potential $\widetilde{K}$, c.f. Section 9.1. Recall that an evaluation point x_j and a boundary element E_k are admissible if

$$\operatorname{diam}(E_k) < \operatorname{dist}(x_j, E_k), \tag{9.2}$$

where $\operatorname{dist}(x_j, E_k)$ denotes the distance between x_j and E_k. If an element and an evaluation point are admissible, we compute the corresponding integral using Gaussian quadrature, otherwise we use analytical formulas.

In case the optional parameter `eta` is specified, we set $\eta = \texttt{eta}$, otherwise we choose $\eta = 1/2$. For $\eta = 0$, all integrals are computed analytically. The default value for η can be changed by recompiling `evaluateKadj.c` and `evaluateW.c` with different values for the preprocessor constant `DEFAULT_ETA`. For Gaussian quadrature, we use a 16-point rule by default. This can be changed by recompiling `evaluateKadj.c` and `evaluateW.c` with different values for the preprocessor constant `GAUSS_ORDER`. Possible values include $2, 4, 8, 16$ and 32.

9.3. Evaluation of the Newton Potential and its Normal Derivative. In this section, we consider the MEX-function `evaluateN` and the MATLAB-function `evaluateN1`. The first one evaluates the Newtonian potential $\widetilde{N}$, defined in Equation (1.3) for piecewise constant functions and arbitrary evaluation points in $\mathbb{R}^2$. The latter one evaluates the normal derivative of the Newtonian potential $N_1 = \gamma_1 \widetilde{N}$ for piecewise affine, globally continuous functions and arbitrary evaluation points on the boundary Γ.

The respective signatures are given by

- `Nf_x = evaluateN(vertices, volumes, fh, x);`
- `N1f_x = evaluateN1(coordinates, elements, lamh, x, p2e [,eta]);`

First, we discuss `evaluateN`. As usual, `vertices` and `volumes` describe a triangulation $\mathcal{T}_\ell$ of the domain Ω. The $N \times 1$ vector `fh` describes a function $F_\ell \in \mathcal{P}^0(\Omega)$, i.e., there holds

$F_\ell = \sum_{i=k}^{N} \texttt{fh(}k\texttt{)}\chi_k$, where N denotes the number of volumes in $\mathcal{T}_\ell(\Omega)$ and χ_k denotes the characteristic function on the k-th element of $\mathcal{T}_\ell(\Omega)$. The $M \times 2$ matrix `x` contains a list of M evaluation points, one per row. `evaluateN` returns an $M \times 1$ vector, where the j-th entry contains the value of NF_ℓ, evaluated at $x_j = \texttt{x(j,:)}$.

In `evaluateN1`, `coordinates` and `elements` describe a boundary mesh $\mathcal{E}_\ell$ with nodes $\mathcal{K}_\ell$, as usual. Like for `evaluateN` the $M \times 2$ matrix `x` contains a list of M evaluation points. The $M \times 1$ vector `p2e` maps the evaluation points to the elements they are contained in, i.e., `p2e(`j`)` $= k$ if $x_j \in E_k \in \mathcal{E}_\ell$. The function `evaluateN1` is implemented by use of the well-known identity

$$(N_1 f)(x) = \left(\left(K' - \frac{1}{2} \right) V^{-1} (Nf) \right)(x),$$

c.f. [31, Lemma 6.20]. Arguing as in Section 8.7, $\lambda = V^{-1}(Nf)$ can be computed by solving an equivalent variational form, c.f. (8.27). The $|\mathcal{E}_\ell| \times 1$ vector `lamh` denotes the solution Λ_ℓ of the discretized variational form (8.29), i.e. `lamh` $= \mathbf{V}^{-1}$(`Nfh`), where $\mathbf{V}$ and $\mathbf{N}$ denote the integral operator matrices of the simple-layer and the Newtonian potential.

LISTING 35. Two-level estimator τ_ℓ for Symm's IE

```
function ind = computeEstSlpTau(father2son,V_fine,b_fine,x_coarse)
nE = size(x_coarse,1);

%*** build index vector son2father to link fine mesh with coarse mesh
son2father = zeros(2*nE,1);
son2father(father2son) = repmat((1:nE)',1,2);

%*** compute energy ||| psi_Ej |||^2 of two-level basis function
energy = 2*( V_fine(father2son(:,1) + 2*nE*(father2son(:,1) - 1)) ...
           - V_fine(father2son(:,1) + 2*nE*(father2son(:,2) - 1)) );

%*** compute residual of x_coarse w.r.t. fine mesh
residual = b_fine - V_fine*x_coarse(son2father);

%*** compute vector of (squared) indicators w.r.t. coarse mesh
ind = ( residual(father2son(:,1)) - residual(father2son(:,2)) ).^2./energy;
```

9.4. Computation of Error Estimator τ_ℓ for Symm's Integral Equation (Listing 35). This section deals with the implementation of the $(h-h/2)$-based two-level error estimator from [27], defined by

$$\tau_\ell := \left(\sum_{i=1}^{N} \tau_\ell(E_i)^2 \right)^{1/2}. \tag{9.3}$$

To define the local contributions $\tau_\ell(E_i)$, let $\widehat{\mathcal{E}}_\ell$ denote the uniform refinement of $\mathcal{E}_\ell = \{E_1, \dots, E_N\}$. Given $E_i \in \mathcal{E}_\ell$, let $e_j, e_k \in \widehat{\mathcal{E}}_\ell$ denote the sons, i.e., $E_i = e_j \cup e_k$. With the associated basis functions $\widehat{\chi}_j, \widehat{\chi}_k \in \mathcal{P}^0(\widehat{\mathcal{E}}_\ell)$, we define the two-level basis function

$$\psi_{E_i} = -\widehat{\chi}_j + \widehat{\chi}_k.$$

The contribution $\tau_\ell(E_i)$ now formally reads

$$\tau_\ell(E_i) = \| \mathbb{G}_{E_i} (\widehat{\Phi}_\ell - \Phi_\ell) \|_V,$$

where $\mathbb{G}_{E_i}$ denotes the Galerkin projection onto the one-dimensional space $\mathrm{span}\{\psi_{E_i}\}$. Here, $\Phi_\ell \in \mathcal{P}^0(\mathcal{E}_\ell)$ and $\widehat{\Phi}_\ell \in \mathcal{P}^0(\widehat{\mathcal{E}}_\ell)$ denote the solutions of (5.21) and (5.22), respectively. In [15], we prove that τ_ℓ is equivalent to the $(h - h/2)$-based error estimator η_ℓ, i.e., $\tau_\ell \simeq \eta_\ell$.

Recall that Linear Algebra predicts a representation of $\mathbb{G}_{E_i}$ in terms of the basis function ψ_{E_i}, namely

$$\mathbb{G}_{E_i}\psi = \frac{\langle\!\langle \psi\,,\psi_{E_i}\rangle\!\rangle_V}{\|\psi_{E_i}\|_V^2}\,\psi_{E_i} \quad \text{for all } \psi \in H^{-1/2}(\Gamma).$$

If we plug-in $\psi = \widehat{\Phi}_\ell - \Phi_\ell$ and use that $\psi_{E_i} \in \mathcal{P}^0(\widehat{\mathcal{E}}_\ell)$, the Galerkin formulation (5.22) for $\mathcal{P}^0(\widehat{\mathcal{E}}_\ell)$ yields

$$\mathbb{G}_{E_i}(\widehat{\Phi}_\ell - \Phi_\ell) = \frac{\langle (K+1/2)G_\ell\,,\,\psi_{E_i}\rangle_\Gamma - \langle\!\langle \Phi_\ell\,,\psi_{E_i}\rangle\!\rangle_V}{\|\psi_{E_i}\|_V^2}\,\psi_{E_i}.$$

To discuss the implementation of $\tau_\ell(E_i)$, let $\mathbf{x} \in \mathbb{R}^N$ and $\widehat{\mathbf{y}} \in \mathbb{R}^{2N}$ be the coefficient vectors of Φ_ℓ with respect to the canonical bases of $\mathcal{P}^0(\mathcal{E}_\ell)$ and $\mathcal{P}^0(\widehat{\mathcal{E}}_\ell)$, i.e.,

$$\Phi_\ell = \sum_{j=1}^{N} \mathbf{x}_j \chi_j = \sum_{j=1}^{2N} \widehat{\mathbf{y}}_j \widehat{\chi}_j.$$

We define the algebraic residual

$$\widehat{\mathbf{r}} := \widehat{\mathbf{b}} - \widehat{\mathbf{V}}\widehat{\mathbf{y}},$$

where $\widehat{\mathbf{V}} \in \mathbb{R}^{2N\times 2N}$ and $\widehat{\mathbf{b}} \in \mathbb{R}^{2N}$ are the Galerkin data with respect to $\widehat{\mathcal{E}}_\ell$. Together with $\psi_{E_i} = -\widehat{\chi}_j + \widehat{\chi}_k$, we consequently obtain

$$\begin{aligned}
\tau_\ell(E_i)^2 &= \frac{|\langle (K+1/2)G_\ell\,,\,\psi_{E_i}\rangle_\Gamma - \langle\!\langle \Phi_\ell\,,\psi_{E_i}\rangle\!\rangle_V|^2}{\|\psi_{E_i}\|_V^2} \\
&= \frac{\big|-\widehat{\mathbf{b}}_j + \widehat{\mathbf{b}}_k - \big(-(\widehat{\mathbf{V}}\widehat{\mathbf{y}})_j + (\widehat{\mathbf{V}}\widehat{\mathbf{y}})_k\big)\big|^2}{\|-\widehat{\chi}_j + \widehat{\chi}_k\|_V^2} \\
&= \frac{|-\widehat{\mathbf{r}}_j + \widehat{\mathbf{r}}_k|^2}{\|\widehat{\chi}_j\|_V^2 - 2\,\langle\!\langle \widehat{\chi}_j\,,\widehat{\chi}_k\rangle\!\rangle_V + \|\widehat{\chi}_k\|_V^2} \\
&= \frac{|-\widehat{\mathbf{r}}_j + \widehat{\mathbf{r}}_k|^2}{2(\|\widehat{\chi}_j\|_V^2 - \langle\!\langle \widehat{\chi}_j\,,\widehat{\chi}_k\rangle\!\rangle_V)}
\end{aligned}$$

Altogether, the documentation of Listing 35 now reads as follows:

- The function takes the coefficient vector $\mathbf{x} \in \mathbb{R}^N$ of the Galerkin solution Φ_ℓ with respect to $\mathcal{E}_\ell$ as well as the Galerkin data $\widehat{\mathbf{V}}$ and $\widehat{\mathbf{b}}$ with respect to the uniformly refined mesh $\widehat{\mathcal{E}}_\ell$. Besides this, the $(N \times 2)$-array `father2son` links the indices of elements $E_i \in \mathcal{E}_\ell$ with the indices of the sons $e_j, e_k \in \widehat{\mathcal{E}}_\ell$ in the sense that `father2son(`i`,:) = [`j`,`k`]` for $E_i = e_j \cup e_k$ and consequently $\widehat{\mathbf{y}}_j = \widehat{\mathbf{y}}_k = \mathbf{x}_i$.
- We construct the inverse relation to `father2son` by building a vector such that $i =$ `son2father`(j) returns the coarse-mesh father $E_i \in \mathcal{E}_\ell$ of element $e_j \in \widehat{\mathcal{E}}_\ell$ (Line 5–6).
- Then, we compute the vector of energies

  $$\|\psi_{E_i}\|_V^2 = 2(\|\widehat{\chi}_j\|_V^2 - \langle\!\langle \widehat{\chi}_j\,,\widehat{\chi}_k\rangle\!\rangle_V),$$

  where $e_j, e_k \in \widehat{\mathcal{E}}_\ell$ are the sons of $E_i \in \mathcal{E}_\ell$ (Line 9–10). For vectorization, we use linear indexing of the matrix $\widehat{\mathbf{V}}$ with the knowledge that MATLAB stores matrices in columnwise order like FORTRAN.
- Next, we compute the residual $\widehat{\mathbf{r}}$ (Line 13), using that the coefficient vector $\widehat{\mathbf{y}}$ is now given in terms of the coarse-mesh coefficient vector $\mathbf{x}$ and the index field `son2father`.
- Finally (Line 16), the function computes and returns the vector

  $$\mathbf{v} := (\tau_\ell(E_1)^2, \dots, \tau_\ell(E_N)^2) \in \mathbb{R}^N.$$

  In particular, there holds $\tau_\ell = \big(\sum_{i=1}^N \mathbf{v}_i\big)^{1/2}$.

Remark 9.1. *The two-level error estimator was one of the first estimators available to steer an h-adaptive mesh-refinement for boundary element methods* [27]. *Compared to the other* $(h - h/2)$*-based error estimators* $\widetilde{\eta}_\ell$ *and* $\widetilde{\mu}_\ell$*, the advantage of* τ_ℓ *is that only the coarse-mesh solution* Φ_ℓ *is needed. However, the implementation still needs the fine-mesh data which is the most time-consuming part of a boundary element implementation: Having assembled the Galerkin data with respect to* $\widehat{\mathcal{E}}_\ell$*, the computational time for the computation of* $\widehat{\Phi}_\ell$ *is empirically negligible. Moreover, our program package* `HILBERT` *restricts to the canonical bases and direct solution of the Galerkin system. Therefore, both Galerkin matrices* $\mathbf{V}$ *and* $\widehat{\mathbf{V}}$ *have to be built to compute* τ_ℓ*. Alternatively, one could either build* $\widehat{\mathbf{V}}$ *with respect to the hierarchical basis* $\{\chi_1, \dots, \chi_N, \psi_{E_1}, \dots, \psi_{E_N}\}$ *so that the* $\mathbf{V}$*-matrix is a subblock of* $\widehat{\mathbf{V}}$*, or one could use an iterative solver. In the latter case, the matrix-vector multiplication with* $\mathbf{V}$ *can be realized via prolongation, matrix-vector multiplication with* $\widehat{\mathbf{V}}$*, and restriction* [18]. *In both cases, one could thus avoid the explicit assembly of* $\mathbf{V}$*, but only build* $\widehat{\mathbf{V}}$*.* □

LISTING 36. Two-level estimator τ_ℓ for hypersingular IE

```
function ind = computeEstHypTau(elements_fine,elements_coarse,...
                                                father2son,W_fine,b_fine,x_coarse)
nC = length(x_coarse);

%*** build index field k = idx(j) such that j-th node of coarse mesh coincides
%*** with k-th node of fine mesh
idx = zeros(nC,1);
idx(elements_coarse) = [ elements_fine(father2son(:,1),1),...
                         elements_fine(father2son(:,2),2) ];

%*** build index field k = mid(j) such that midpoint of j-th element of coarse
%*** mesh is k-th node of fine mesh
mid = elements_fine(father2son(:,1),2);

%*** compute coefficient vector of u_coarse w.r.t. fine mesh
x = zeros(length(b_fine),1);
x(mid) = 0.5*sum(x_coarse(elements_coarse),2);
x(idx) = x_coarse;

%*** obtain energies ||| phi_j |||^2 from matrix W_fine where phi_j denotes the
%*** hatfunction corresponding to j-th node of the uniformly refined coarse mesh
energy = diag(W_fine);

%*** compute residual of x_coarse w.r.t. fine mesh
residual = b_fine - W_fine*x;

%*** compute vector of (squared) indicators w.r.t. coarse mesh
%*** as described above
ind = residual(mid).^2 ./ energy(mid);
```

9.5. Computation of Error Estimator τ_ℓ for the Hypersingular Integral Equation (Listing 36). This section deals with the implementation of the $(h - h/2)$-based two-level error estimator from [23, 26], defined by

$$\tau_\ell := \Big(\sum_{i=1}^{N} \tau_\ell(E_i)^2 \Big)^{1/2}. \tag{9.4}$$

The contribution $\tau_\ell(E_i)$ formally reads

$$\tau_\ell(E_i) = \|\mathbb{G}_{E_i}(\widehat{U}_\ell - U_\ell)\|_{W+S},$$

where $\mathbb{G}_{E_i}$ denotes the Galerkin projection onto the one-dimensional space $\operatorname{span}\{\zeta_{\mathrm{E_i}}\}$. Here, U_ℓ and $\widehat{U}_\ell$ denote the solutions of (6.21) and (6.22). The two-level basis function $\zeta_{E_i} \in \mathcal{S}^1(\widehat{\mathcal{E}}_\ell)$ is just the fine-mesh basis function associated with the midpoint of the coarse-mesh element E_i. In [16], we prove that τ_ℓ is equivalent to the $(h-h/2)$-based error estimator η_ℓ from Section 6.6, i.e., $\tau_\ell \simeq \eta_\ell$.

Recall that Linear Algebra predicts a representation of $\mathbb{G}_{E_i}$ in terms of the basis function ζ_{E_i}, namely

$$\mathbb{G}_{E_i}\zeta = \frac{\langle\!\langle \zeta\,, \zeta_{E_i}\rangle\!\rangle_{W+S}}{\|\zeta_{E_i}\|^2_{W+S}}\,\zeta_{E_i} \quad \text{for all } \zeta \in H^{1/2}(\Gamma).$$

If we plug-in $\zeta = \widehat{U}_\ell - U_\ell$ and use that $\zeta_{E_i} \in \mathcal{S}^1(\widehat{\mathcal{E}}_\ell)$, the Galerkin formulation (6.22) for $\mathcal{S}^1(\widehat{\mathcal{E}}_\ell)$ yields

$$\mathbb{G}_{E_i}(\widehat{U}_\ell - U_\ell) = \frac{\langle (1/2 - K')\Phi_\ell\,,\, \zeta_{E_i}\rangle_\Gamma - \langle\!\langle U_\ell\,, \zeta_{E_i}\rangle\!\rangle_{W+S}}{\|\zeta_{E_i}\|^2_{W+S}}\,\zeta_{E_i}.$$

To discuss the implementation of $\tau_\ell(E_i)$, let $\mathbf{x} \in \mathbb{R}^N$ and $\widehat{\mathbf{y}} \in \mathbb{R}^{2N}$ be the coefficient vectors of U_ℓ with respect to the basis of $\mathcal{S}^1(\mathcal{E}_\ell)$ and $\mathcal{S}^1(\widehat{\mathcal{E}}_\ell)$, i.e.,

$$U_\ell = \sum_{j=1}^{N} \mathbf{x}_j \zeta_j = \sum_{j=1}^{2N} \widehat{\mathbf{y}}_j \widehat{\zeta}_j.$$

We define the algebraic residual

$$\widehat{\mathbf{r}} := \widehat{\mathbf{b}} - (\widehat{\mathbf{W}} + \widehat{\mathbf{S}})\widehat{\mathbf{y}},$$

where $\widehat{\mathbf{W}} + \widehat{\mathbf{S}} \in \mathbb{R}^{2N\times 2N}$ and $\widehat{\mathbf{b}} \in \mathbb{R}^{2N}$ are the Galerkin data with respect to $\widehat{\mathcal{E}}_\ell$. We consequently obtain

$$\tau_\ell(E_i)^2 = \frac{|\langle (1/2 - K')\Phi_\ell\,,\, \zeta_{E_i}\rangle_\Gamma - \langle\!\langle U_\ell\,, \zeta_{E_i}\rangle\!\rangle_{W+S}|^2}{\|\zeta_{E_i}\|^2_{W+S}} = \frac{\big|\widehat{\mathbf{b}}_{\mathrm{mid}(i)} - \big((\widehat{\mathbf{W}} + \widehat{\mathbf{S}})\widehat{\mathbf{y}}\big)_{\mathrm{mid}(i)}\big|^2}{\|\zeta_{E_i}\|^2_{W+S}} \tag{9.5}$$

$$= \frac{|\widehat{\mathbf{r}}_{\mathrm{mid}(i)}|^2}{(\widehat{\mathbf{W}} + \widehat{\mathbf{S}})_{\mathrm{mid}(i),\mathrm{mid}(i)}}, \tag{9.6}$$

where $\mathrm{mid}(i)$ provides the index of the degree of freedom which corresponds to the midpoint node of the fine mesh $\widehat{\mathcal{E}}_\ell$ of E_i. Altogether, the documentation of Listing 36 now reads as follows:

- The function takes the coefficient vector $\mathbf{x} \in \mathbb{R}^N$ of the Galerkin solution U_ℓ with respect to $\mathcal{E}_\ell$, the Galerkin data $\widehat{\mathbf{b}}$ as well as the sum $\widehat{\mathbf{W}} + \widehat{\mathbf{S}}$ of the hypersingular matrix $\widehat{\mathbf{W}}$ and the stabilization term matrix $\widehat{\mathbf{S}}$ for the fine mesh $\widehat{\mathcal{E}}_\ell$ stored in `W_fine`. Besides this, the $(N \times 2)$-array `father2son` links the indices of elements $E_i \in \mathcal{E}_\ell$ with the indices of the sons $e_j, e_k \in \widehat{\mathcal{E}}_\ell$ in the sense that `father2son(`i`,:) = [`j`,`k`]` for $E_i = e_j \cup e_k$. Furthermore we need the array `elements_coarse` describing the coarse triangulation $\mathcal{E}_\ell$ and `elements_fine` describing the uniformly refined mesh $\widehat{\mathcal{E}}_\ell$.
- We construct an index field $k =$`idx(`j`)` such that the j-th node of the coarse mesh coincides with the k-th node of the fine mesh (Line 7–9).
- Similar to our first step, we construct an index field $k =$`mid(`j`)` such that the midpoint of the j-th element of the coarse mesh coincides with the k-th node of the fine mesh (Line 13).
- Next, we compute the coefficient vector of U_ℓ with respect to the fine mesh (Line 16–18).
- Then, we compute the vector of energies $\|\zeta_{E_i}\|^2_{W+S}$ from the matrix $\widehat{\mathbf{W}} + \widehat{\mathbf{S}}$ where we use the fact that ζ_{E_i} denotes the hatfunction corresponding to the i-th node of the uniformly refined coarse mesh $\widehat{\mathcal{E}}_\ell$ as seen in the last term of (9.5) (Line 22).
- Next, we compute the residual $\widehat{\mathbf{r}}$ (Line 25).

- Finally (Line 29), the function computes and returns the vector

$$\mathbf{v} := (\tau_\ell(E_1)^2, \dots, \tau_\ell(E_N)^2) \in \mathbb{R}^N.$$

  In particular, there holds $\tau_\ell = \big(\sum_{i=1}^N \mathbf{v}_i\big)^{1/2}$.

LISTING 37. Two-level estimator τ_ℓ for mixed BVP

```
function ind = computeEstMixTau(father2dirichlet, father2neumann,neumann_coarse,...
    neumann_fine, V_fine, K_fine, W_fine, bD_fine, bN_fine, x_coarse,...
    free_neumann,free_neumann_fine)
nD_coarse = size(father2dirichlet,1);
dof_fine = 2*nD_coarse+size(K_fine,2);
Ndof_fine = length(free_neumann_fine);
Ndof = length(free_neumann);

%*** build index vector son2father to link fine mesh with coarse mesh
dirichlet2father = zeros(2*nD_coarse,1);
dirichlet2father(father2dirichlet) = repmat((1:nD_coarse)',1,2);

%*** build index field k = idx(j) such that j-th node of coarse mesh coincides
%*** with k-th node of fine mesh
idx = zeros(2,1);
idx(neumann_coarse) = [ neumann_fine(father2neumann(:,1),1),...
                        neumann_fine(father2neumann(:,2),2) ];
idx=idx(free_neumann);

%*** build index field k = mid(j) such that midpoint of j-th element of coarse
%*** mesh is k-th node of fine mesh
mid = neumann_fine(father2neumann(:,1),2);

%*** compute coefficient vector of u_coarse w.r.t. fine mesh
x = zeros(dof_fine,1);
x_coarse_tmp=zeros(2,1);
x_coarse_tmp(free_neumann)=x_coarse(1:Ndof);
x(mid) = 0.5*sum(x_coarse_tmp(neumann_coarse),2);
x(idx) = x_coarse(1:Ndof);
x=x(free_neumann_fine);
x(Ndof_fine+(1:2*nD_coarse))=x_coarse(Ndof+dirichlet2father);

%*** map dirichlet coordinates onto degrees of freedom on the dirichlet
%*** boundary
idxFN=zeros(2,1);
idxFN(free_neumann_fine)=1:Ndof_fine;

%*** compute residual w.r.t. the coarse mesh solution
A=[W_fine,K_fine';-K_fine,V_fine];
res = A*x-[bN_fine;bD_fine];

%*** compute engergy
energy = [diag(W_fine);V_fine(father2dirichlet(:,1)...
    + 2*nD_coarse*(father2dirichlet(:,1) - 1))...
    - V_fine(father2dirichlet(:,1) ...
    + 2*nD_coarse*(father2dirichlet(:,2) - 1))];

%*** compute estimator
ind=[res(idxFN(mid)).^2./energy(idxFN(mid));...
    (res(father2dirichlet(:,1)+Ndof_fine)...
```

```
51      -res(father2dirichlet(:,2)+Ndof_fine)).^2 ...
52      ./(2*energy(Ndof_fine+1:end))];
```

9.6. Computation of Error Estimator τ_ℓ for the mixed problem (Listing 37). This section deals with the implementation of the $(h-h/2)$-based two-level error estimator for the mixed problem described in Section 7 resp. Section 8.11 in case of nonhomogeneous volume forces. The estimator is defined by

$$\tau_\ell := \Big(\sum_{i=1}^{N} \tau_\ell(E_i)^2\Big)^{1/2}. \tag{9.7}$$

As for the preceding cases, the contribution $\tau_\ell(E_i)$ formally reads

$$\tau_\ell(E_i) = \|\mathbb{G}_{E_i}(\widehat{\mathbf{U}}_\ell - \mathbf{U}_\ell)\|_A,$$

where $\mathbb{G}_{E_i}$ denotes the Galerkin projection onto the one-dimensional space $\operatorname{span}\{(\zeta_{E_i}, 0)\}$ for $E_i \subset \Gamma_N$ or $\operatorname{span}\{(0, \psi_{E_i})\}$ for $E_i \subset \Gamma_D$. For the definitions of the two-level functions ζ_{E_i} and ψ_{E_i}, we refer to Sections 9.4 and 9.5. Here, $\mathbf{U}_\ell$ and $\widehat{\mathbf{U}}_\ell$ denote the solutions of (7.26) and (7.27).

To compute $\mathbb{G}_{E_i}$ we distinguish two cases: First, let $E_i \subset \Gamma_D$. Then, it holds

$$\begin{aligned}\|\mathbb{G}_{E_i}(\widehat{\mathbf{U}}_\ell - \mathbf{U}_\ell)\|_A &= \frac{\big|\langle\!\langle \widehat{\mathbf{U}}_\ell - \mathbf{U}_\ell\,, (0,\psi_{E_i})\rangle\!\rangle_A\big|}{\|(0,\psi_{E_i})\|_A} \\ &= \frac{\big|\langle \mathcal{F}_\ell\,, (0,\psi_{E_i})\rangle_{\mathcal{H}^\star\times\mathcal{H}} - \langle -KU_\ell + V\Phi_\ell\,, \psi_{E_i}\rangle_{\Gamma_D}\big|}{\|\psi_{E_i}\|_V}.\end{aligned}$$

For $E_i \subset \Gamma_N$, it holds

$$\begin{aligned}\|\mathbb{G}_{E_i}(\widehat{\mathbf{U}}_\ell - \mathbf{U}_\ell)\|_A &= \frac{\big|\langle\!\langle \widehat{\mathbf{U}}_\ell - \mathbf{U}_\ell\,, (\zeta_{E_i},0)\rangle\!\rangle_A\big|}{\|(\zeta_{E_i},0)\|_A} \\ &= \frac{\big|\langle \mathcal{F}_\ell\,, (\zeta_{E_i},0)\rangle_{\mathcal{H}^\star\times\mathcal{H}} - \langle WU_\ell + K'\Phi_\ell\,, \zeta_{E_i}\rangle_{\Gamma_N}\big|}{\|\zeta_{E_i}\|_{W+S}}.\end{aligned}$$

In both cases, we exploited the Galerkin formulation (7.17) as well as the fact that $(\zeta_{E_i},0), (0,\phi_{E_i}) \in \widehat{X}_\ell$.

To discuss the implementation of $\tau_\ell(E_i)$, let $\mathbf{x} \in \mathbb{R}^N$ denote the solution vector of (7.19) or (8.48), i.e., the representation of $\mathbf{U}_\ell$ w.r.t. the basis of X_ℓ. Let additionally $\widehat{\mathbf{y}} \in \mathbb{R}^{2N}$ denote the representation of $\mathbf{U}_\ell$ w.r.t. the basis of $\widehat{X}_\ell$. The algebraic residual reads

$$\widehat{\mathbf{r}} := \widehat{\mathbf{b}} - \begin{pmatrix} \widehat{\mathbf{W}} & \widehat{\mathbf{K}}^T \\ -\widehat{\mathbf{K}} & \widehat{\mathbf{V}} \end{pmatrix} \widehat{\mathbf{y}},$$

where $\widehat{\mathbf{W}}, \widehat{\mathbf{K}}, \widehat{\mathbf{V}}$ and $\widehat{\mathbf{b}} \in \mathbb{R}^{2N}$ are the Galerkin data with respect to $\widehat{\mathcal{E}}_\ell$. We consequently obtain

$$\tau_\ell(E_i)^2 = \begin{cases} |\widehat{\mathbf{r}}_i|/\|\psi_{E_i}\|_V & \text{for } E_i \subset \Gamma_D, \\ |\widehat{\mathbf{r}}_i|/\|\zeta_{E_i}\|_{W+S} & \text{for } E_i \subset \Gamma_N. \end{cases}$$

The right way call the function **`computeEstMixTau`** is

```
ind = computeEstMixTau(father2dirichlet, father2neumann, neumann_coarse,...
      neumann_fine, V_fine, K_fine, W_fine, bD_fine, bN_fine, x_coarse,...
      free_neumann,free_neumann_fine);
```

The arrays `father2dirichlet` and `father2neumann` link the coarse mesh with the fine mesh. The Neumann part of the meshes is described via `neumann_coarse` and `neumann_fine`. The integral operators `V_fine`, `K_fine`, and `W_fine` must be provided in the shrinked form, i.e., the same form which is used to solve for the mixed solution vector $\mathbf{x}$ in (7.19) or (8.48). `x_coarse` represents the solution vector $\mathbf{x}$ on the coarse mesh $\mathcal{E}_\ell$ and `free_neumann` as well as

`free_neumann_fine` indicate the indices of the degrees of freedom on the Neumann boundary for the coarse mesh and the fine mesh, respectively.

For the documentation, we refer to Section 9.4 and Section 9.5. The only part which is new is the prolongation of the coarse mesh solution vector `x`.

- Lines 25–27 provide a vector which contains the nodal values of U_ℓ on Γ_N and is zero elsewhere.
- This vector is prolonged to $\mathcal{S}^1(\widehat{\mathcal{E}}_\ell)$ in Line 28–30 and restricted to the degrees of freedom on the Neumann boundary Γ_N.
- Finally, the element values of Φ_ℓ are prolongated and added to `x` in Line 31. The vector `x` represents the restriction of $\widehat{\mathbf{y}}$ onto the degrees of freedom.
- Lines 35–36 provide a link between the numbers of the degrees of freedom on the Neumann boundary Γ_N of the fine mesh and the corresponding nodes of $\widehat{\mathcal{E}}_\ell$, i.e., `j=idxFN(k)` means that the k-th node on the Neumann boundary corresponds to the j-th degree of freedom on the fine mesh.

LISTING 38. Weighted-residual error estimator ρ_ℓ for Symm's IE

```
function ind = computeEstSlpResidual(varargin)
if nargin == 6
  [vertices,volumes,elements,phih,gh,fh] = varargin{:};
  coordinates = vertices(unique(elements),:);
elseif nargin == 5
  [coordinates,elements,phih,gh_dir,gh] = varargin{:};
elseif nargin == 4
  [coordinates,elements,phih,gh] = varargin{:};
end

%*** Gaussian quadrature on [-1,1] with 2 nodes and exactness 3
quad_nodes = [-1 1]/sqrt(3);
quad_weights = [1;1];

%*** elementwise interpolation is done in (gauss_left,gauss_right,midpoint)
quad_nodes(3) = 0;
nE = size(elements,1);
nQ = length(quad_nodes);

%*** build vector of evaluations points as (nQ*nE x 2)-matrix
a = coordinates(elements(:,1),:);
b = coordinates(elements(:,2),:);
sx = reshape(a,2*nE,1)*(1-quad_nodes) + reshape(b,2*nE,1)*(1+quad_nodes);
sx = 0.5*reshape(sx',nQ*nE,2);

%*** evaluate gh elementwise at (left, right, midpoint)
if ~isempty(gh)
  gh_left = gh(elements(:,1));
  gh_right = gh(elements(:,2));
  gh_sx = gh_left*(1-quad_nodes) + gh_right*(1+quad_nodes);
  gh_sx = 0.5*reshape(gh_sx',nQ*nE,1);
end

%*** evaluate V*phih in all interpolation nodes sx
p = evaluateV(coordinates,elements,phih,sx);

%*** distinguish between different cases
if (nargin == 6)
  if ~isempty(gh)
    p = p - evaluateK(coordinates,elements,gh,sx) ...
        - 0.5*gh_sx ...
        + evaluateN(vertices,volumes,fh,sx);
  else
    p = p + evaluateN(vertices,volumes,fh,sx);
  end
elseif (nargin == 5) && isempty(gh_dir)
  p = p - gh_sx;
elseif nargin == 4
  p = p - evaluateK(coordinates,elements,gh,sx) ...
      - 0.5*gh_sx;
end

%*** evaluate arclength-derivative p' elementwise at (left,right)
p_prime = reshape(p,nQ,nE)' * [-3 1 ; -1 3 ; 4 -4]*sqrt(0.75);

%*** return ind(j) = diam(Ej) * || [ V*phi - (K+1/2)*gh ]' ||_{L2(Ej)}^2
```

```
57 ind = 2*p_prime.^2*quad_weights;
```

9.7. Computation of Weighted-Residual Error Estimator ρ_ℓ for Symm's Integral Equation (Listing 38). Due to the fact that the simple-layer potential $V : H^{-1/2}(\Gamma) \to H^{1/2}(\Gamma)$ is an isomorphism, we may consider the residual of Symm's integral equation (5.1) on the mesh $\mathcal{E}_\ell$:

$$\|V\Phi_\ell - (1/2 + K)G_\ell\|_{H^{1/2}(\Gamma)} \simeq |\!|\!| \phi_\ell - \Phi_\ell |\!|\!|_V \quad \text{for all } \ell \in \mathbb{N}, \tag{9.8}$$

where $\Phi_\ell \in \mathcal{P}^0(\mathcal{E}_\ell)$ denotes the discrete solution of (5.7). Thus, the left-hand side of the above equation provides an efficient and reliable error estimator for the right-hand side. Unfortunately, the $H^{1/2}$-norm doesn't provide local information on where to refine the mesh. Therefore, the following localization was firstly proposed in [8] for 2D and later extended to 3D in [9]:

$$\begin{aligned} &\|V\Phi_\ell - (1/2 + K)G_\ell\|^2_{H^{1/2}(\Gamma)} \\ &\qquad \lesssim \sum_{E_i \in \mathcal{E}_\ell} \operatorname{length}(E_i) \|(V\Phi_\ell - (1/2 + K)G_\ell)'\|^2_{L^2(E_i)} =: \sum_{E_i \in \mathcal{E}_\ell} \rho_\ell(E_i)^2. \end{aligned} \tag{9.9}$$

Now, the global estimator can be written as $\rho_\ell = \|h_\ell^{1/2}(V\Phi_\ell - (1/2+K)G_\ell)'\|_{L^2(\Gamma)}$. The hidden constant in estimate (9.9) depends only on Γ and an upper bound for the local mesh-ratio $\kappa(\mathcal{E}_\ell)$.
Inclusion of Volume Forces: In an analogous way, one may derive a weighted-residual error estimator for Symm's integral equation with volume forces. To that end, let $\Phi_\ell \in \mathcal{P}^0(\mathcal{E}_\ell)$ denote the solution of (8.6). The estimator now reads

$$\rho_\ell := \|h_\ell^{1/2}(V\Phi_\ell - (1/2 + K)G_\ell + N_0F_\ell)'\|_{L^2(\Gamma)}. \tag{9.10}$$

Error Estimator for Indirect Formulation: For a given function $g \in H^{1/2}(\Gamma)$, where $\Gamma \subseteq \partial\Omega$, the indirect formulation of Symm's integral equation is to find $\phi \in \widetilde{H}^{-1/2}(\Gamma)$ with

$$|\!|\!| \phi \,, \psi |\!|\!|_V = \langle g \,,\, \psi \rangle_\Gamma \quad \text{for all } \psi \in \widetilde{H}^{-1/2}(\Gamma). \tag{9.11}$$

The space $\widetilde{H}^{-1/2}(\Gamma)$ consists of all continuous linear functionals on $H^{1/2}(\Gamma)$. Hence, if $\Gamma \subsetneq \partial\Omega$, and $\phi \in \widetilde{H}^{-1/2}(\Gamma)$ has additonal regularity $\phi \in L^2(\Gamma)$, this corresponds to extending ϕ by 0 on $\partial\Omega$. As in the case of the Dirichlet problem, we approximate g by a discrete function $G_\ell \in \mathcal{S}^1(\mathcal{E}_\ell)$ and solve the corresponding Galerkin formulation

$$|\!|\!| \Phi_\ell \,, \Psi_\ell |\!|\!|_V = \langle G_\ell \,,\, \Psi_\ell \rangle_\Gamma \quad \text{for all } \Psi_\ell \in \mathcal{P}^0(\mathcal{E}_\ell). \tag{9.12}$$

In a similar way, we can derive the weighted-residual estimator

$$\rho_\ell := \|h_\ell^{1/2}(V\Phi_\ell - G_\ell)'\|_{L^2(\Gamma)}. \tag{9.13}$$

Error Estimator for an Auxiliary Problem: Solving the hypersingular integral equation with non-vanishing volume forces needs the Galerkin solution Λ_ℓ of the auxiliary problem (8.29). Instead of employing (8.35), we could estimate the introduced error by

$$|\!|\!| \lambda - \Lambda_\ell |\!|\!|_V \approx \rho_\ell + \operatorname{osc}_{\Omega,\ell},$$

with ρ_ℓ beeing the weighted-residual error estimator

$$\rho_\ell := \|h_\ell^{1/2} \, (V\Lambda_\ell - N_0F_\ell)' \, \|_{L^2(\Gamma)}. \tag{9.14}$$

Note that this is a special case of a weighted-residual error estimator for Symm's integral fomulation with volume force, where $G_\ell = 0$ and the sign of $N_0 f_\ell$ changed.
Implementation: For the numerical implementation, we use the same approach as for the computation of the Dirichlet data oscillations in Section 5.1. Let R_ℓ denote the residual in either of the cases introduced above. For $E_j = [a_j, b_j] \in \mathcal{E}_\ell$ and $h := |b_j - a_j|$, let $\gamma_j : [-1, 1] \to E_j$ denote the reference parametrization (2.1). Recall that $|\gamma_j'| = h/2$. With the definition

of a boundary integral from Section 2.2 and the definition of the arclength derivative from Section 2.3, we obtain

$$\|R'_\ell\|^2_{L^2(E_j)} = \int_{E_j} \big(R'_\ell\big)^2 \, d\Gamma = \frac{h}{2}\int_{-1}^{1} \big((R'_\ell \circ \gamma_j)(s)\big)^2 \, ds = \frac{2}{h}\int_{-1}^{1} \big((R_\ell \circ \gamma_j)'(s)\big)^2 \, ds. \tag{9.15}$$

We now approximate $R_\ell \circ \gamma_j : [-1,1] \to \mathbb{R}$ by a polynomial $p_j \in \mathcal{P}^2\,[-1,1]$. Using the quadrature points from a 2-point Gauss rule on $[-1,1]$,

$$x_1 = -\frac{\sqrt{3}}{3}, \quad \text{and} \quad x_2 = \frac{\sqrt{3}}{3},$$

as well as the midpoint $m_j = (a_j + b_j)/2$ of E_j, we define

$$p_j(x_1) = R_\ell \circ \gamma_j(x_1), \quad p_j(0) = R_\ell(m_j), \quad p_j(x_2) = R_\ell \circ \gamma_j(x_2).$$

Note that $p'_j \in \mathcal{P}^1[-1,1]$ and $\Big(p'_j\Big)^2 \in \mathcal{P}^2[-1,1]$ so that

$$\|R'_\ell\|^2_{L^2(E_j)} = \frac{2}{h}\int_{-1}^{1} \big((R_\ell \circ \gamma_j)'(s)\big) \, ds \approx \frac{2}{h}\int_{-1}^{2} \big(p'_j\big)^2 \, ds = \frac{2}{h}\,\mathrm{quad}_2\big(\big(p'_j\big)^2\big),$$

where quad_2 is a quadrature rule on $[-1,1]$ which is exact on $\mathcal{P}^2[-1,1]$. In fact, wee use the 2-point Gauss rule with quadrature points x_1 and x_2. To evaluate $p'_j(x_1)$ and $p'_j(x_2)$, we proceed as for the Dirichlet data oscillations in Section 5.1. With L_1, L_2, and L_3 denoting the Lagrange polynomials w.r.t. x_1, 0, and x_2 on $[-1,1]$,

$$\begin{aligned}
&L_1(s) = \frac{1}{2}s(3s - \sqrt{3}), && L_2(s) = 1 - 3s^2, && L_3(s) = \frac{1}{2}s(3s + \sqrt{3}),\\
&L'_1(s) = 3s - \frac{\sqrt{3}}{2}, && L'_2(s) = -6s, && L'_3(s) = 3s + \frac{\sqrt{3}}{2},
\end{aligned}$$

we obtain

$$\begin{aligned}
\begin{pmatrix} p'_j(x_1) \\ p'_j(x_2) \end{pmatrix} &= \begin{pmatrix} L'_1(x_1) & L'_2(x_1) & L'_3(x_1) \\ L'_1(x_2) & L'_2(x_2) & L'_3(x_2) \end{pmatrix} \begin{pmatrix} R_\ell \circ \gamma_j(x_1) \\ R_\ell(m_j) \\ R_\ell \circ \gamma_j(x_2) \end{pmatrix} \\
&= \sqrt{\frac{3}{4}} \begin{pmatrix} -3 & 4 & -1 \\ 1 & -4 & 3 \end{pmatrix} \begin{pmatrix} R_\ell \circ \gamma_j(x_1) \\ R_\ell(m_j) \\ R_\ell \circ \gamma_j(x_2) \end{pmatrix}.
\end{aligned} \tag{9.16}$$

Remark 9.2. *To obtain a reliable error estimator for $\|\!|\phi - \Phi_\ell|\!\|_V$, again one has to incorporate the Dirichlet data oscillations $\mathrm{osc}_{D,\ell}$ for the cases (9.9) and (9.13), i.e.,*

$$\|\!|\phi - \Phi_\ell|\!\|_V \lesssim \rho_\ell + \mathrm{osc}_{D,\ell} \quad \textit{for all } \ell \in \mathbb{N}$$

and additionally the volume oscillations $\mathrm{osc}_{\Omega,\ell}$ for the case (9.10), i.e.,

$$\|\!|\phi - \Phi_\ell|\!\|_V \lesssim \rho_\ell + \mathrm{osc}_{D,\ell} + \mathrm{osc}_{\Omega,\ell} \quad \textit{for all } \ell \in \mathbb{N}.$$

The hidden constants in the last two estimates depend only on Γ and an upper bound for $\kappa(\mathcal{T}_\ell)$.

The implementation **`computeEstSlpResidual`** of the estimator ρ_ℓ covers the four cases (9.9), (9.10), (9.13), and (9.14) and is found in Listing 38. We first describe the four different sets of parameters for the call of **`computeEstSlpResidual`**:

- To compute (9.9), the function must be called in the form

  ```
  ind = computeEstSlpResidual(coordinates,elements,phih,gh)
  ```

  The function takes the mesh $\mathcal{E}_\ell$ in terms of `coordinates` and `elements`. The parameter `phih` is the coefficient vector `x` of the Galerkin solution Φ_ℓ of (5.7), whereas the parameter `gh` is the coefficient vector `g` of the discrete Dirichlet data G_ℓ which can be computed either by nodal interpolation with **`computeOscDirichlet`** or by the L^2-projection with **`computeOscDirichletL2`**.

- To compute (9.10), the function must be called in the form

```
ind = computeEstSlpResidual(vertices,triangles,elements,phih,gh,fh)
```

  The input parameters `vertices` and `triangles` describe the triangulation $\mathcal{T}_\ell$ of Ω as explained in Section 3.3. The parameter `elements` describes the boundary mesh. The implementation **`computeEstSlpResidual`** assumes that the nodes on the boundary appear at the beginning of the vector `vertices`. This can be assured by reordering the mesh with the help of the function **`buildSortedMesh`**, see Section 4.2. The parameter `phih` represents the coefficient vector $\mathbf{x}$ of the Galerkin solution Φ_ℓ of (8.6). The parameters `gh` and `fh` describe the discrete Dirichlet data G_ℓ and discrete volume data F_ℓ of (8.6). Again, G_ℓ can be computed either by nodal interpolation with **`computeOscDirichlet`** or by the L^2-projection with **`computeOscDirichletL2`**, whereas F_ℓ can be computed by **`computeOscVolume`**. In case of `gh`$= 0$, the function can be called via

```
ind = computeEstSlpResidual(vertices,triangles,elements,phih,[],fh)
```

  where the call of `evaluateK` is avoided.
- To compute (9.13), the function must be called in the form

```
ind = computeEstSlpResidual(coordinates,elements,phih,[],gh)
```

  Here, `gh` is the coefficient vector $\mathbf{u}$ of the discrete data G_ℓ and can be computed as in the other two cases. We stress that the fourth parameter has to be an empty vector.
- To compute (9.14), the function must be called in the form

```
ind = computeEstSlpResidual(vertices,triangles,elements,lambdah,[],-fh)
```

  The input parameters `vertices` and `triangles` describe the triangulation $\mathcal{T}_\ell$ of Ω as explained in section 3.3. The parameter `elements` describes the boundary mesh. The implementation **`computeEstSlpResidual`** assumes that the nodes on the boundary appear at the beginning of the vector `vertices`. This can be assured by reordering the mesh with the help of the function **`buildSortedMesh`**, see Section 4.2. The parameter `lambdah` represents the coefficient vector $\mathbf{x}$ of the Galerkin solution Λ_ℓ of (8.29). Since we set `gh=[]` and use $-$`fh` as last input parameter, we ensure that (9.14) is computed.

In either case, the output `ind` is a vector of the squared local contributions of ρ_ℓ. The different cases are distinguished by the different number of arguments that are provided as input parameters, Line 2–21. We now describe the rest of the implementation of **`computeEstSlpResidual`**.

- To compute the arc-length derivative of the residual, we take the approach discussed above. At first, we prepare the quadrature nodes and weights of the gauss_2 rule in Lines 24 and 25. As interpolation nodes, we take the quadrature nodes and the midpoint in Lines 28–30.
- The transformation onto the elements is done in Line 33–36. The vector `sx` contains interpolation nodes for each element $E_i \in \mathcal{E}_\ell$.
- In Lines 39–44, we evaluate the discrete data G_ℓ, which is represented by `gh`, in all interpolation nodes. Note, however, that this is done only if `gh` is not empty, as is the case when we invoke **`computeEstSlpResidual`** to estimate the error of the auxiliary problem (9.14).
- Now, the vector `p` is used to add up all the residual contributions for the different cases in Lines 50–63.
- In Line 66, we use the evaluations of p_j to compute their derivatives in the quadrature points as stated in (9.16).
- Line 69 applies the quadrature rule and returns the squared elementwise error estimator in the vector `ind`.

LISTING 39. Weighted-residual error estimator ρ_ℓ for the hypersingular IE

```
1 function ind = computeEstHypResidual(varargin)
2 if nargin == 5
3   [coordinates,elements,gh,phih_dir,phih_ind] = varargin{:};
```

```
elseif nargin == 4
  [coordinates,elements,gh,phih_dir] = varargin{:};
end

%*** Gaussian quadrature on [-1,1] with 2 nodes and exactness 3
quad_nodes = [-1 1]/sqrt(3);
quad_weights = [1;1];

%*** define constants
nE = size(elements,1);
nQ = length(quad_nodes);

%*** build vector of all quadrature nodes as (nQ*nE x 2)-matrix
a = coordinates(elements(:,1),:);
b = coordinates(elements(:,2),:);
sx = reshape(a,2*nE,1)*(1-quad_nodes) ...
        + reshape(b,2*nE,1)*(1+quad_nodes);
sx = 0.5*reshape(sx',nQ*nE,2);

%*** sx2element(j) returns the element number k such that sx(j,:) lies on Ek
sx2element = reshape(repmat((1:nE),nQ,1),nQ*nE,1);

%*** compute vector of (squared) element-widths
h = sum((a-b).^2,2);

%*** compute outer normal vector
normal = b-a;
normal = [normal(:,2),-normal(:,1)]./repmat(sqrt(h),1,2);

%*** evaluate the the associated terms
if (nargin == 5) && isempty(phih_dir)
  p = evaluateW(coordinates,elements,gh,sx,normal(sx2element,:)) - phih_ind(sx2element);
else
  p = evaluateW(coordinates,elements,gh,sx,normal(sx2element,:)) ...
    - 0.5*phih_dir(sx2element) ...
    + evaluateKadj(coordinates,elements,phih_dir,sx,normal(sx2element,:));
  if nargin == 5
    p = p + evaluateKadj(coordinates,elements,phih_ind,sx,normal(sx2element,:)) ...
      - 0.5*phih_ind(sx2element);
  end
end

p = reshape(p,nQ,nE)';
ind = 0.5*h.*((p.^2)*quad_weights);
```

9.8. Computation of Weighted-Residual Error Estimator ρ_ℓ for Hypersingular Integral Equation (Listing 39). The hypersingular integral operator W is an isomorphism from $H^{1/2}_\star(\Gamma)$ to $H^{-1/2}_\star(\Gamma)$. Here, the lower index $(\cdot)_\star$ denotes, in case of $\Gamma = \partial\Omega$, that functions v of this space have vanishing integral mean, i.e., $\int_\Gamma v\,d\Gamma = 0$. We may thus consider the residual of the hypersingular integral equation (6.1) on the mesh $\mathcal{E}_\ell$:

$$\|WU_\ell - (1/2 - K')\Phi_\ell\|_{H^{-1/2}(\Gamma)} \simeq |\!|\!| u - U_\ell |\!|\!|_W \qquad \text{for all } \ell \in \mathbb{N},$$

where $U_\ell \in \mathcal{S}^1(\Gamma)$ denotes the discrete solution of (6.7). As in the case of Symm's integral equation, we deal with the non-local $H^{-1/2}(\Gamma)$-norm on the left hand side. The following

localization was firstly analyzed in [8]:

$$\begin{aligned}\|WU_\ell - (1/2 - K')\Phi_\ell\|^2_{H^{-1/2}(\Gamma)} \\ \lesssim \sum_{E_i\in\mathcal{E}_\ell} \operatorname{length}(E_i)\|WU_\ell - (1/2 - K')\Phi_\ell\|^2_{L_2(E_i)} =: \sum_{E_i\in\mathcal{E}_\ell} \rho_\ell(E_i)^2.\end{aligned} \tag{9.17}$$

Again, the constants in this estimate depend solely on Γ and an upper bound for the local mesh-ratio $\kappa(\mathcal{E}_\ell)$.

Inclusion of Volume Forces: In an analogous way, one may derive a weighted-residual error estimator for the hypersingular integral equation with non-vanishing volume force. To that end, let $U_\ell \in \mathcal{S}^1(\mathcal{E}_\ell)$ denote the Galerkin solution of the hypersingular integral equation with volume force (8.30) and define the estimator by

$$\rho_\ell := \|h_\ell^{1/2}\left(WU_\ell - (1/2 - K')(\Phi_\ell + \Lambda_\ell)\right)\|_{L_2(\Gamma)}, \tag{9.18}$$

where $\Lambda_\ell \in \mathcal{P}^0(\mathcal{E}_\ell)$ denotes the solution of (8.29).

Error Estimator for Indirect Formulation: For a given function $\phi \in H_\star^{-1/2}(\Gamma)$, where $\Gamma \subseteq \partial\Omega$, the indirect formulation of the hypersingular integral equation is to find $u \in \widetilde{H}_*^{1/2}(\Gamma)$ such that

$$\langle\!\langle u\,, v\rangle\!\rangle_W = \langle \phi\,,\, v\rangle_\Gamma \quad \text{for all } v \in H_*^{1/2}(\Gamma).$$

The space $\widetilde{H}_*^{1/2}(\Gamma)$ is either the space $H_*^{1/2}(\Gamma)$ in case of $\Gamma = \partial\Omega$, or the space $\widetilde{H}^{1/2}(\Gamma)$ if $\Gamma \subsetneq \partial\Omega$, which is the space of restrictions of all functions $v \in H^{1/2}(\partial\Omega)$ which have support in $\overline{\Gamma}$. As in the case of the Neumann problem, we approximate ϕ by a discrete function $\Phi_\ell \in \mathcal{P}^0(\mathcal{E}_\ell)$ and solve the corresponding Galerkin formulation

$$\langle\!\langle U_\ell\,, V_\ell\rangle\!\rangle_W = \langle \Phi_\ell\,,\, V_\ell\rangle_\Gamma \tag{9.19}$$

and can derive the weighted-residual error estimator in a similar way as before to obtain

$$\rho_\ell := \|h_\ell^{1/2}\,(WU_\ell - \Phi_\ell)\,\|_{L^2(\Gamma)} \tag{9.20}$$

Implementation: We will not comment on any implementational details here, as they are essentially the same as for the Neumann Data oscillations in Section 6.1. Nevertheless, we comment on the parameters needed to call the function **`computeEstHypResidual`** for the desired case.

- To compute (9.17), the function must be called in the form

  ```
  ind = computeEstHypResidual(coordinates,elements,gh,phih)
  ```

  Here, `coordinates` and `elements` describe the mesh $\mathcal{E}_\ell$. The parameter `gh` is the coefficient vector $\mathbf{x}$ of the Galerkin solution $U_\ell \in \mathcal{S}^1(\mathcal{E}_\ell)$ of (6.7), whereas the parameter `phih` is the coefficient vector $\mathbf{p}$ of the discrete Neumann data Φ_ℓ which can be computed by the function **`computeOscNeumann`**.
- To compute (9.18), the function must be called in the form

  ```
  ind = computeEstHypResidual(coordinates,elements,gh,phih,lambdah)
  ```

  The parameter `gh` represents the coefficient vector $\mathbf{x}$ of the Galerkin solution (8.30), whereas the parameter `phih` is the coefficient vector $\mathbf{p}$ of the discrete Neumann data Φ_ℓ which can be computed by the function **`computeOscNeumann`**. The last parameter, `lambdah`, is the coefficient vector of the Galerkin solution $\Lambda_\ell \in \mathcal{P}^0(\mathcal{E}_\ell)$ of (8.29).
- To compute (9.20), the function must be called in the form

  ```
  ind = computeEstHypResidual(coordinates,elements,gh,[],phih)
  ```

  The parameter `gh` represents the coefficient vector $\mathbf{x}$ of the Galerkin solution (9.19), whereas `phih` denotes the coefficient vector of the right hand side Φ_ℓ of (9.19).

LISTING 40. Weighted-residual error estimator ρ_ℓ for mixed BVP

```
function [indSLP,indHYP] = computeEstMixResidual(varargin)
if nargin == 11
  [vertices,coordinates,volumes,dirichlet,neumann,uNh,...
        phiDh,uDh,phiNh,fh,lambdah] = varargin{:};
  elements=[dirichlet;neumann];
elseif nargin == 7
  [coordinates,dirichlet,neumann,uNh,phiDh,uDh,phiNh] = varargin{:};
  elements=[dirichlet;neumann];
end
%***compute first equation estimate
%*** Gaussian quadrature on [-1,1] with 2 nodes and exactness 3
quad_nodes = [-1 1]/sqrt(3);
quad_weights = [1;1];

%*** elementwise interpolation is done in (gauss_left,gauss_right,midpoint)
quad_nodes(3) = 0;
nE = size(elements,1);
nQ = length(quad_nodes);

%*** build vector of evaluations points as (nQ*nE x 2)-matrix
a = coordinates(elements(:,1),:);
b = coordinates(elements(:,2),:);
sx = reshape(a,2*nE,1)*(1-quad_nodes) + reshape(b,2*nE,1)*(1+quad_nodes);
sx = 0.5*reshape(sx',nQ*nE,2);

%*** evaluate gh elementwise at (left, right, midpoint)

uDh_left = uDh(elements(:,1));
uDh_right = uDh(elements(:,2));
uDh_sx = uDh_left*(1-quad_nodes) + uDh_right*(1+quad_nodes);
uDh_sx = 0.5*reshape(uDh_sx',nQ*nE,1);
pSLP = evaluateV(coordinates,dirichlet,phiDh,sx) ...
          - evaluateK(coordinates,elements,uDh,sx) ...
          - 0.5*uDh_sx ...
          + evaluateV(coordinates, elements, phiNh,sx) ...
          - evaluateK(coordinates,neumann,uNh,sx);
if nargin == 11
    pSLP = pSLP + evaluateN(vertices, volumes,fh,sx);
end

%*** evaluate arclength-derivative p' elementwise at (left,right)
pSLP_prime = reshape(pSLP,nQ,nE)' * [-3 1 ; -1 3 ; 4 -4]*sqrt(0.75);

%*** return ind(j) = diam(Ej) * || [ V*phi - (K+1/2)*gh ]' ||_{L2(Ej)}^2
indSLP = 2*pSLP_prime.^2*quad_weights;

%***compute second equation estimate
quad_nodes(3)=[];
nQ=length(quad_nodes);
%*** build vector of all quadrature nodes as (nQ*nE x 2)-matrix
a = coordinates(elements(:,1),:);
b = coordinates(elements(:,2),:);
sx = reshape(a,2*nE,1)*(1-quad_nodes) ...
       + reshape(b,2*nE,1)*(1+quad_nodes);
sx = 0.5*reshape(sx',nQ*nE,2);

```

```
%*** sx2element(j) returns the element number k such that sx(j,:) lies on Ek
sx2element = reshape(repmat((1:nE),nQ,1),nQ*nE,1);
%*** compute vector of (squared) element-widths
h = sum((a-b).^2,2);

%*** compute outer normal vector
normal = b-a;
normal = [normal(:,2),-normal(:,1)]./repmat(sqrt(h),1,2);

%*** evaluate p1 = (1/2-Kadj)*phih at (left,right)
pHYP = 0.5*phiNh(sx2element) ...
   - evaluateKadj(coordinates,elements,phiNh,sx,normal(sx2element,:))...
   - evaluateKadj(coordinates,dirichlet,phiDh,sx,normal(sx2element,:));

if nargin == 11
   pHYP=pHYP - evaluateKadj(coordinates,elements,lambdah,sx,normal(sx2element,:)) ...
        +0.5*lambdah(sx2element);
end

pHYP=pHYP-evaluateW(coordinates,elements,uDh+uNh,sx,normal(sx2element,:));

%*** evaluate integrand p = (1/2-Kadj)*phih - W*gh = p1 + p2'
pHYP = reshape(pHYP,nQ,nE)';

%*** return ind(j) = diam(Ej) * ||(1/2-K*)phih+(Vgh')'||_{L2(Ej)}^2
indHYP = 0.5*h.*(pHYP.^2*quad_weights);
```

9.9. Computation of Weighted-Residual Error Estimator ρ_ℓ for the Mixed BVP (Listing 40). Similar to the two preceding sections, the residual of the mixed BVP (7.9) measured in the dual norm provides a reliable and efficient error estimator

$$\|\!|\mathbf{u}_\ell - \mathbf{U}_\ell|\!\|_A \simeq \|\!|(\mathrm{res}_{N,\ell}, \mathrm{res}_{D,\ell})|\!\|_{A^\star},$$

where

$$\begin{aligned}\mathrm{res}_{D,\ell} &:= V\Phi_\ell - (1/2+K)U_{D,\ell} + V\Phi_{N,\ell} - KU_\ell,\\ \mathrm{res}_{N,\ell} &:= (1/2-K')\Phi_{N,\ell} - K'\Phi_\ell - W(U_\ell + U_{D,\ell}),\end{aligned}$$

and where $\mathbf{u}_\ell \in \mathcal{H}_\ell$ denotes the exact solution of (7.9) with perturbed data $U_{N,\ell}$ and $\Phi_{D,\ell}$. The localization techniques from the hypersingular case as well as the case of Symm's integral equation apply analogously. Therefore, we consider the weighted-residual error estimator

$$\rho_\ell^2 := \|h_\ell^{1/2}\mathrm{res}'_{D,\ell}\|^2_{L^2(\Gamma_D)} + \|h_\ell^{1/2}\mathrm{res}_{N,\ell}\|^2_{L^2(\Gamma_N)}, \tag{9.21}$$

which provides an upper bound for the error, i.e., $\|\!|\mathbf{u}_\ell - \mathbf{U}_\ell|\!\|_A \lesssim \rho_\ell$ with a constant which depends only on Γ and an upper bound for the local mesh-ratio $\kappa(\mathcal{E}_\ell)$.

Inclusion of Volume Forces: As before, one may also include volume forces. To that end, let $\mathbf{U}_\ell$ denote the Galerkin solution of the mixed problem with volume force (8.46) and define the estimator by

$$\rho_\ell^2 := \|h_\ell^{1/2}(\mathrm{res}_{D,\ell} - N_0F_\ell)'\|^2_{L^2(\Gamma_D)} + \|h_\ell^{1/2}(\mathrm{res}_{N,\ell} - (1/2-K')\Lambda_\ell)\|^2_{L^2(\Gamma_N)}, \tag{9.22}$$

where $\Lambda_\ell \in \mathcal{P}^0(\mathcal{E}_\ell)$ denotes the solution of (8.29).

Remark 9.3. *Obviously, one has to include data oscillations to obtain a reliable error estimator for the perturbed problem, i.e.,*

$$\|\!|\mathbf{u} - \mathbf{U}_\ell|\!\|_A \lesssim \rho_\ell + \mathrm{osc}_{D,\ell} + \mathrm{osc}_{N,\ell}$$

for the case (9.21) *and*

$$\|\mathbf{u} - \mathbf{U}_\ell\|_A \lesssim \rho_\ell + \rho_{V,\ell} + \mathrm{osc}_{D,\ell} + \mathrm{osc}_{N,\ell} + \mathrm{osc}_{\Omega,\ell}$$

for (9.22). *Here,* $\rho_{V,\ell}$ *is the weighted-residual error estimator for Symm's integral equation* $V\Lambda_\ell = N_0 F_\ell$, *i.e.,* $\rho_{V,\ell} := \|h_\ell^{1/2}(V\Lambda_\ell - N_0F_\ell)'\|_\Gamma$. *This is used to control the error of the approximation of* N_1F_ℓ *by* $(1/2 - K')\Lambda_\ell$.

To call the function for the mixed problem without volume forces (9.21), use

```
[indSLP,indHYP] = computeEstMixResidual(coordinates,dirichlet,neumann,...
                  uNh,phiDh,uDh,phiNh);
```

For the case with volume forces (9.22), use

```
[indSLP,indHYP] = computeEstMixResidual(vertices,coordinates,volumes,...
                  dirichlet,neumann,uNh,phiDh,...
                  uDh,phiNh,fh,lambdah);
```

The arrays `vertices`, `volumes`, `coordinates`, `dirichlet`, `neumann` describe the volume mesh as well as the boundary mesh in case of (9.22). The vectors `uNh`, `phiDh` describe the solution $\mathbf{U}_\ell = (U_\ell, \Phi_\ell) \in X_\ell$. It is important, that `uNh` has one entry for every coordinate, where the entries which don't correspond to a degree of freedom on the Neumann boundary are zero. The data $(U_{D,\ell}, \Phi_{N,\ell})$ is given by the vectors `uDh` and `phiNh`. In case of non-homogeneous volume forces (9.22), one has to provide the discretized volume data `fh`, as well as the solution vector `lambdah` of (8.29) which represents Λ_ℓ and is used to compute N_1F_ℓ.

The implementation in Listing 40 combines the techniques from the two previous sections, and we therefore omit further details.

Acknowledgement. The majority of the authors, namely MA, MF, PG, MK, MM, and DP, are partially funded through the research project *Adaptive Boundary Element Method*, funded by the Austria Science Fund (FWF) under grant P21732. The author SFL acknowledges a grant of the graduate school *Differential Equations – Models in Science and Engineering*, funded by the Austrian Science Fund (FWF) under grant W800-N05.

References

[1] M. Aurada, M. Ebner, S. Ferraz-Leite, M. Mayr, P. Goldenits, M. Karkulik, D. Praetorius: *HILBERT — A Matlab implementation of adaptive BEM*, software download at `http://www.asc.tuwien.ac.at/abem/hilbert/`

[2] M. Aurada, M. Ebner, S. Ferraz-Leite, M. Mayr, P. Goldenits, M. Karkulik, D. Praetorius: *The Analytical Computation of the Newton Potential*, work in progress 2010.

[3] M. Aurada, M. Feischl, T. Führer, M. Karkulik, D. Praetorius: *Efficiency and optimality of some weighted-residual error estimator for adaptive 2D boundary element methods*, ASC Report **15/2012**, Institute for Analysis and Scientific Computing, Vienna University of Technology, 2012.

[4] M. Aurada, S. Ferraz-Leite, D. Praetorius: *Estimator reduction and convergence of adaptive BEM*, Appl. Numer. Math., **62** (2012), 787–801.

[5] M. Aurada, P. Goldenits, D. Praetorius: *Convergence of Data Perturbed Adaptive Boundary Element Methods*, ASC Report **40/2009**, Institute for Analysis and Scientific Computing, Vienna University of Technology, 2009.

[6] M. Aurada, S. Ferraz-Leite, P. Goldenits, M. Karkulik, M. Mayr, D. Praetorius: *Convergence of adaptive BEM for some mixed boundary value problem*, Appl. Numer. Math., **62** (2012), 226–245.

[7] S. Börm, M. Löhndorf, J. Melenk: *Approximation of Integral Operators by Variable-Order Interpolation*, Numer. Math. **99** (2005), 605-643.

[8] C. Carstensen, E.P. Stephan: *A posteriori error estimates for boundary element methods*, Math. Comp. **64** (1995), 483–500.

[9] C. Carstensen, M. Maischak, E.P. Stephan: *A posteriori error estimate and h-adaptive algorithm on surfaces for Symm's integral equation*, Numer. Math. **90** (2001), no. 2, 197–213.

[10] J. Cascon, C. Kreuzer, R. Nochetto, K. Siebert: *Quasi-Optimal Convergence Rate for an Adaptive Finite Element Method*, SIAM J. Numer. Anal. **46** (2008), 2524–2550.

[11] C. Carstensen, D. Praetorius: *Averaging Techniques for the Effective Numerical Solution of Symm's Integral Equation of the First Kind*, SIAM J. Sci. Comput., **27** (2006), 1226–1260.
[12] C. Carstensen, D. Praetorius: *Averaging Techniques for the A Posteriori BEM Error Control for a Hypersingular Integral Equation in Two Dimensions*, SIAM J. Sci. Comput., **29** (2007), 782–810.
[13] W. Doerfler: *A Convergent Adaptive Algorithm for Possion's Equation*, SIAM J. Numer. Anal. **33** (1996), 1106–1124.
[14] W. Doerfler, R. Nochetto: *Small Data Oscillation Implies the Saturation Assumption*, Numer. Math. 91 (2002), 1–12.
[15] C. Erath, S. Ferraz-Leite, S. Funken, D. Praetorius: *Energy norm based a posteriori error estimation for boundary element methods in two dimensions*, Appl. Numer. Math., **59** (2009), 2713–2734.
[16] C. Erath, S. Funken, P. Goldenits, D. Praetorius: *Simple error estimators for the Galerkin BEM for some hypersingular integral equation in 2D*, accepted for publication in Appl. Anal., 2012.
[17] S. Ferraz-Leite, C. Ortner, D. Praetorius: *Convergence of simple adaptive Galerkin schemes based on $h - h/2$ error estimators*, Numer. Math., **116** (2010), 291–316.
[18] S. Ferraz-Leite, D. Praetorius: *Simple A Posteriori Error Estimators for the h-Version of the Boundary Element Method*, Computing **83** (2008), 135–162.
[19] S. Funken, D. Praetorius, P. Wissgott: *Efficient implementation of adaptive P1-FEM in MATLAB*, Comput. Methods Appl. Math., **11** (2011), 460–490.
[20] W. Hackbusch: *Hierarchische Matrizen — Algorithmen und Analysis*, Springer, Berlin 2009.
[21] M. Karkulik, G. Of, D. Praetorius: *Convergence of adaptive 3D BEM for weakly singular integral equations based on isotropic mesh-refinement*, in press, Numer. Methods Partial Differential Equations (2013).
[22] M. Maischak: *The Analytical Computation of the Galerkin Elements for the Laplace, Lamé and Helmholtz Equation in 2D-BEM*, Preprint, Institut für Angewandte Mathematik, Universität Hannover, Hannover, 1999.
[23] M. Maischak, P. Mund, E. Stephan: *Adaptive multilevel BEM for acoustic scattering*, Symposium on Advances in Computational Mechanics, Vol. 2 (Austin, TX, 1997), Comput. Methods Appl. Mech. Engrg. **150** (1997), 351–367.
[24] M. Mayr: *Stabile Implementierung der Randelementmethode auf stark adaptierten Netzen*, Bachelor thesis (in German), Institute for Analysis and Scientific Computing, Vienna University of Technology, 2010.
[25] W. McLean: *Strongly Elliptic Systems and Boundary Integral Equations*, Cambridge University Press, Cambridge, 2000.
[26] P. Mund, E. Stephan: *An adaptive two-level method for hypersingular integral equations in $\mathbb{R}^3$*, Proceedings of the 1999 International Conference on Computational Techniques and Applications (Canberra), ANZIAM J: **42** (2000), C1019-C1033.
[27] P. Mund, E. Stephan, J. Weisse: *Two-Level Methods for the Single Layer Potential in $\mathbb{R}^3$*, Computing **60** (1998), 243–266.
[28] G. Of, O. Steinbach, P. Urthaler: *Fast Evaluation of Newton Potentials in the Boundary Element Method*, SIAM J. Sci. Comput. **32** (2010), 585–602.
[29] S. Rjasanov, O. Steinbach: *The Fast Solution of Boundary Integral Equations*, Springer, New York, 2007.
[30] S. Sauter, C. Schwab: *Randelementmethoden: Analysis, Numerik und Implementierung schneller Algorithmen*, Teubner, Wiesbaden, 2004.
[31] O. Steinbach: *Numerical Approximation Methods for Elliptic Boundary Value Problems: Finite and Boundary Elements*, Springer, New York, 2008.
[32] R. Stevenson: *The Completion of Locally Refined Simplicial Partitions Created by Bisection*, Math. Comp. **77** (2008), 227–241.
[33] S. Wandzura, H. Xiao: *Symmetric Quadrature Rules on a Triangle*, Comput. Math. Appl. **45** (2003), 1829–1840.

Institute for Analysis and Scientific Computing, Vienna University of Technology, Wiedner Hauptstrasse 8-10, A-1040 Wien, Austria
E-mail address: Dirk.Praetorius@tuwien.ac.at (corresponding author)