Solving Boundary Integral Problems with BEM++

WOJCIECH ŚMIGAJ, University College London and Adam Mickiewicz University in Poznań
TIMO BETCKE, SIMON ARRIDGE, JOEL PHILLIPS and MARTIN SCHWEIGER,
University College London

Many important partial differential equation problems in homogeneous media, such as those of acoustic or electromagnetic wave propagation, can be represented in the form of integral equations on the boundary of the domain of interest. In order to solve such problems, the boundary element method (BEM) can be applied. The advantage compared to domain-discretisation-based methods such as finite element methods is that only a discretisation of the boundary is necessary, which significantly reduces the number of unknowns. Yet, BEM formulations are much more difficult to implement than finite element methods. In this paper we present BEM++, a novel open-source library for the solution of boundary integral equations for Laplace, Helmholtz and Maxwell problems in three space dimensions. BEM++ is a C++ library with Python bindings for all important features, making it possible to integrate the library into other C++ projects or to use it directly via Python scripts. The internal structure and design decisions for BEM++ are discussed. Several examples are presented to demonstrate the performance of the library for larger problems.

Categories and Subject Descriptors: G.1.8 [**Partial Differential Equations**]: Elliptic equations; G.1.9 [**Integral Equations**]: Fredholm equations; G.4 [**Mathematical Software**]: Algorithm design and analysis

Additional Key Words and Phrases: boundary element methods, boundary integral equations, C++, Python interface

ACM Reference Format:
Śmigaj, W., Betcke, T., Arridge, S., Phillips, J., and Schweiger, M. 2012. Solving boundary integral problems with BEM++. ACM Trans. Math. Softw. V, N, Article A (January YYYY), 50 pages.
DOI = 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

1. INTRODUCTION

The efficient numerical solution of partial differential equations (PDEs) via boundary integral formulations plays an important role in diverse applications, such as acoustics, electrostatics, computational electromagnetics or elasticity [Colton and Kress 2013; Nédélec 2001; Harrington and Harrington 1996]. Consider as an example a Laplace problem of the form

$$-\Delta u(\mathbf{x}) = 0 \tag{1}$$

This work is supported by Engineering and Physical Sciences Research Council Grants EP/I030042/1 and EP/H004009/2.
Author's addresses: W. Śmigaj, Department of Mathematics, University College London, London, UK, and Faculty of Physics, Adam Mickiewicz University, Poznań, Poland; S. Arridge and M. Schweiger, Department of Computer Science, University College London, London, UK; T. Betcke and J. Phillips, Department of Mathematics, University College London, London, UK.

DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

in some domain $\Omega \subset \mathbb{R}^d$ with piecewise smooth Lipschitz boundary Γ, where $d = 2, 3$. Green's representation theorem allows us to write the solution u as

$$u(\mathbf{x}) = \int_\Gamma g(\mathbf{x},\mathbf{y}) \frac{\partial}{\partial n} u(\mathbf{y}) \,\mathrm{d}\Gamma(\mathbf{y}) - \int_\Gamma \frac{\partial}{\partial n(\mathbf{y})} g(\mathbf{x},\mathbf{y})\, u(\mathbf{y}) \,\mathrm{d}\Gamma(\mathbf{y}) \quad \text{for} \quad \mathbf{x} \in \Omega. \tag{2}$$

Here, $\mathbf{n}$ is the unit outward pointing normal at Γ and $g(\mathbf{x},\mathbf{y})$ is the Laplace Green's function defined as

$$g(\mathbf{x},\mathbf{y}) = \begin{cases} -\frac{1}{2\pi} \log|\mathbf{x}-\mathbf{y}|, & d = 2, \\ \frac{1}{4\pi|\mathbf{x}-\mathbf{y}|}, & d = 3. \end{cases} \tag{3}$$

Hence, in principle if either u or $\frac{\partial}{\partial n} u$ is known on Γ, we can recover the unknown quantity by restricting (2) to the boundary and solving for the unknown boundary data.

The advantage of boundary-integral formulations of PDE problems is that we require only $\mathcal{O}(N^{d-1})$ unknowns to discretise the boundary Γ, where N is the number of variables in each space dimension. In contrast, for domain-based methods we need $\mathcal{O}(N^d)$ variables. Moreover, solving exterior problems in $\Omega^+ := \mathbb{R}^d \setminus \overline{\Omega}$ is naturally possible with boundary integral equations. Yet there are also some fundamental disadvantages compared to domain formulations. Probably the most significant ones are:

— Calculation of matrix entries requires the evaluation of complicated singular integrals, which in the case of Galerkin formulations for problems in $d = 3$ are four-dimensional.
— The operators are non-local, leading to dense matrices. Hence, the cost of a matrix-vector product for problems in $d = 3$ space dimensions is $\mathcal{O}(N^4)$, while for standard finite elements with sparse discretisations the cost is $\mathcal{O}(N^3)$.

Traditionally, the cost of dense-matrix storage and evaluation has restricted the applicability of boundary element methods to problems of moderate size. However, advances in the evaluation of singular integrals appearing in boundary element methods and the development of fast formulations based on $\mathcal{H}$-matrices, wavelets or the fast multipole method (FMM) have made it possible to solve very large application problems with boundary elements. The fast formulations reduce the cost and storage of matrix-vector products to $\mathcal{O}(N^2 \log^\alpha N)$ ($\alpha \geq 0$ depends on the formulation) for problems in three space dimensions [Of et al. 2006; Cheng et al. 2006; Bebendorf 2008; Harbrecht and Schneider 2006; Sauter and Schwab 2011].

In this paper we describe a novel project to develop a modern open-source library for boundary-element calculations, BEM++. It combines many of the recent advances in the development of boundary element methods into one easy-to-use software package.

The library itself is written in C++. In addition, we provide Python bindings for almost all high-level features of the library. The Python wrappers also contain several simple visualisation functions that facilitate interactive use of the library.

To allow a natural strong formulation of boundary integral problems the library uses the concepts of spaces, dual spaces, operators, and grid functions. Weak-form discretisations are done automatically when they are needed. Automatic projections provide mappings between function spaces and their duals so that functions on grids are always represented in the correct spaces.

Extensibility is ensured by building heavily on C++ object orientation. The library uses templates for its low-level implementation, on which a standard object-oriented layer is built. The latter makes use of runtime polymorphism and inheritance to allow the user to extend the library, e.g. by providing new kernels, function spaces or grid types.

Reuse of existing high-quality software was an important principle from the start. The grid management is done with Dune-Grid [Bastian et al. 2008b; Bastian et al. 2008a; DUNE 2012], a high-performance parallel grid library, which supports features such as load balancing and adaptive refinement. Although this advanced functionality of Dune is not yet used in the current version of BEM++, it ensures the availability of the basic infrastructure for parallelisation on HPC clusters. For the solution of linear systems we provide interfaces to Trilinos [Heroux et al. 2005; Trilinos 2012], giving access to a range of high-quality iterative solvers, and allow to use BEM++ objects without conversion in more complex Trilinos-based applications. Fast solution of large boundary-element problems is made possible by an interface to AHMED [Bebendorf 2008; 2012], which implements the adaptive cross approximation (ACA) algorithm and a complete $\mathcal{H}$-matrix algebra.

The library is an ongoing project and its functionality is continuously extended. The most recent version of BEM++, 2.0, includes the following features:

— Galerkin discretisation of the single-, double- and adjoint double-layer potential boundary operators and hypersingular boundary operators associated with the Laplace and Helmholtz equations in three dimensions, as well as of the single- and double-layer potential boundary operators associated with the Maxwell equations in three dimensions.
— Off-surface evaluation of potentials.
— Piecewise polynomial scalar basis functions of order up to 10 (continuous or discontinuous) and Raviart-Thomas basis functions of the lowest order.
— Grids composed of planar triangular elements; import of grids in Gmsh format.
— Solution of discretised equations using iterative solvers from Trilinos (including GMRES, CG).
— Interfaces to AHMED for $\mathcal{H}$-matrix assembly, $\mathcal{H}$-matrix-vector product and $\mathcal{H}$-matrix-based preconditioners.
— Parallel matrix assembly and matrix-vector product on shared-memory machines.
— Python wrappers of the main library features.

These features permit the current version of the library to be used already in a wide range of contexts while development of advanced features, in particular MPI support, is ongoing.

The current version of the library relies on Galerkin discretisations of boundary integral equations. We do not claim that this is always the best approach. Collocation, and in particular Nyström methods have been used highly successfully in a range of application areas. The initial focus on Galerkin discretisations was motivated by their suitability for a natural description of FEM-BEM coupling and by the aim of incorporating at a later stage novel developments about a posteriori error estimation. (A future goal is to provide results together with good error estimates.) However, the software design of the library is open to other types of discretisation methods, and in future we may choose to add support for e.g. Nyström methods.

The library itself is available under a permissive MIT license, which allows unrestricted use in open-source and commercial applications. The licenses of the dependencies are compatible with this model in the sense that they do not restrict the license of the main library. The only exception is AHMED, which is only open for non-commercial applications. The use of AHMED in BEM++ is optional.

The source code of the library is available from its home page, `www.bempp.org`. The library comes with a dedicated Python-based installer that automatically downloads and installs all necessary dependencies before building and installing BEM++ itself. Full installation instructions can be found on the website of the library.

BEM++ is not the only recent open-source software project whose aim is to develop a versatile boundary element library. The Fortran package Maiprogs [Maischak 2013] makes it possible to use Galerkin BEM to solve the Laplace, Helmholtz, Lamé and Stokes equations. High-order basis functions, including the hp variant of BEM, are supported. BEMLAB [Wieleba and Sikora 2011] is a C++ library capable of solving the Laplace and Helmholtz equations; piecewise constant, linear and quadratic basis functions are supported. The Concepts C++ library [Schmidt 2013] currently contains an implementation of BEM for the Laplace equation. Both the Galerkin and the collocation variants are available, with piecewise constant or linear basis functions. All the preceding libraries handle both 2D and 3D geometries. However, only dense matrix assembly is possible (FMM and ACA are not supported), which limits the applicability of these libraries to problems of modest size.

Two general-purpose libraries implementing accelerated variants of BEM are HyENA (Hyperbolic and Elliptic Numerical Analysis [Messner et al. 2010])[1] and BETL (Boundary Element Template Library [Hiptmair and Kielhorn 2012; Kielhorn 2012]). HyENA uses ACA to speed up the solution of the Laplace, Helmholtz and Lamé equations in 2D and 3D using the Galerkin or collocation approaches. Up to quadratic basis functions are supported. BETL provides access to implementations of both ACA and FMM, and can be used to discretise boundary operators associated with the Laplace, Helmholtz and Maxwell equations in 3D using the Galerkin approach. Isoparametric elements of order up to 4 are available. An in-depth overview of BETL can be found in Hiptmair and Kielhorn [2012].

In addition, there exist several codes designed for solving a particular equation with BEM. For instance, Puma-EM [van den Bosch 2013] is a C++/Python package that calculates electromagnetic fields scattered by perfectly conducting 3D obstacles using FMM-accelerated BEM.

Like BETL and HyENA, BEM++ provides an accelerated BEM implementation based on ACA, thanks to its interface to the AHMED library. However, while both HyENA and BETL are designed as pure C++ template libraries. BEM++ has been developed for easy access from scripting languages, making very different design decisions necessary (see section 3.2). Another difference is the high-level operator interface in BEM++, which allows natural formulations of products of two operators and products of operators and functions. The correct mappings between the spaces are handled automatically by BEM++ (see section 3.5).

The plan of this article is as follows. In section 2 we review the foundations of boundary element methods. Section 3 is devoted to a presentation of the major features of BEM++. The practical use of BEM++ is demonstrated in section 4, where we develop example Python scripts that use the library to solve particular PDEs. Finally, in section 5 we discuss plans for further development of the library.

2. GALERKIN BOUNDARY ELEMENTS

2.1. Boundary integral operators and Calderón projection

Elliptic equations. In this section we will give a brief overview of some of the concepts of boundary integral equations for the solution of the Laplace problem (1). For a complete presentation of the theory of boundary integral equations see e.g. Steinbach [2008].

We denote by $v := \gamma_0^{\text{int}} u$ the Dirichlet trace of $u(\mathbf{x})$ onto the boundary Γ and by $t := \gamma_1^{\text{int}} u$ the conormal derivative, which in the case of the Laplace problem is just the normal derivative, or Neumann trace, of the solution $u(\mathbf{x})$ on Γ. By convention we

[1]We would like to thank the developers of the HyENA project for making available their implementation of the Sauter-Schwab quadrature rules [Sauter and Schwab 2011] to BEM++.

assume that normal directions point to the exterior of the domain Ω. We use the symbol $H^s(\Omega)$ for the standard Sobolev space of order $s \in \mathbb{R}$ on Ω, as defined in Steinbach [2008, p. 33]. Define the single-layer potential operator $\mathcal{V} : H^{-\frac{1}{2}}(\Gamma) \to H^1(\Omega)$ and the double-layer potential operator $\mathcal{K} : H^{\frac{1}{2}}(\Gamma) \to H^1(\Omega)$ by

$$[\mathcal{V}\psi](\mathbf{x}) = \int_\Gamma g(\mathbf{x},\mathbf{y})\,\psi(\mathbf{y})\,\mathrm{d}\Gamma(\mathbf{y}), \quad [\mathcal{K}\phi](\mathbf{x}) = \int_\Gamma \gamma^{\text{int}}_{1,\mathbf{y}} g(\mathbf{x},\mathbf{y})\,\phi(\mathbf{y})\,\mathrm{d}\Gamma(\mathbf{y}), \quad \mathbf{x} \in \Omega. \tag{4}$$

Using Green's representation theorem (2) the solution u now takes the form

$$u = \mathcal{V}t - \mathcal{K}v. \tag{5}$$

Then by taking traces on both sides of (5), we get

$$v = \left(\frac{1}{2}I - K\right)v + Vt. \tag{6}$$

where $V : H^{-\frac{1}{2}}(\Gamma) \to H^{\frac{1}{2}}(\Gamma)$ is the single-layer potential boundary operator and $K : H^{\frac{1}{2}}(\Gamma) \to H^{\frac{1}{2}}(\Gamma)$ the double-layer potential boundary operator, defined by

$$[V\psi](\mathbf{x}) := \int_\Gamma g(\mathbf{x},\mathbf{y})\,\psi(\mathbf{y})\,\mathrm{d}\Gamma(\mathbf{y}) \qquad \text{and} \qquad [K\phi](\mathbf{x}) := \int_\Gamma \gamma^{\text{int}}_{1,\mathbf{y}} g(\mathbf{x},\mathbf{y})\,\phi(\mathbf{y})\,\mathrm{d}\Gamma(\mathbf{y}) \tag{7}$$

for $(\psi,\phi) \in H^{-\frac{1}{2}}(\Gamma) \times H^{\frac{1}{2}}(\Gamma)$. The identity operator in (6) results from the jump relation of the double-layer potential. In a strict sense the pre-factor $\frac{1}{2}$ is only valid almost everywhere [Steinbach 2008, p. 123]. Taking the conormal derivative on both sides of (5) leads to

$$t = Dv + \left(\frac{1}{2}I + T\right)t. \tag{8}$$

Here, $D : H^{\frac{1}{2}}(\Gamma) \to H^{-\frac{1}{2}}(\Gamma)$ is the hypersingular operator and $T : H^{-\frac{1}{2}}(\Gamma) \to H^{-\frac{1}{2}}(\Gamma)$ is the adjoint double-layer potential boundary operator. They are defined by

$$[T\psi](\mathbf{x}) := \int_\Gamma \gamma^{\text{int}}_{1,\mathbf{x}} g(\mathbf{x},\mathbf{y})\,\psi(\mathbf{y})\,\mathrm{d}\Gamma(\mathbf{y}) \tag{9}$$

and

$$[D\phi](\mathbf{x}) := -\gamma^{\text{int}}_{1,\mathbf{x}} \left[\int_\Gamma \gamma^{\text{int}}_{1,\mathbf{y}} g(\mathbf{x},\mathbf{y})\,\phi(\mathbf{y})\,\mathrm{d}\Gamma(\mathbf{y})\right]. \tag{10}$$

Combining (6) and (8) we obtain the Calderón projection

$$\begin{bmatrix} v \\ t \end{bmatrix} = \begin{bmatrix} \frac{1}{2}I - K & V \\ D & \frac{1}{2}I + T \end{bmatrix} \begin{bmatrix} v \\ t \end{bmatrix}. \tag{11}$$

By prescribing either v or t we can derive from (11) an equation for the corresponding other unknown. A pair $(v,t) \in H^{\frac{1}{2}}(\Gamma) \times H^{-\frac{1}{2}}(\Gamma)$ describes the boundary trace and conormal trace of a solution of the Laplace problem (1) if and only if this pair satisfies (11). With a suitable kernel $g(x,y)$ the representation in (11) is also valid for other elliptic PDEs, e.g. the Helmholtz equation.

Maxwell equations. Consider a domain Ω with boundary Γ, filled with a material with permittivity ϵ and permittivity μ, and define the wavenumber $k := (\epsilon\mu)^{1/2}\omega$. The treatment of the time-harmonic Maxwell equations

$$\nabla \times \mathbf{E} = \mathrm{i}\omega\mu\mathbf{H}, \qquad \nabla \times \mathbf{H} = -\mathrm{i}\omega\epsilon\mathbf{E}, \tag{12}$$

in BEM++ closely follows that of Buffa and Hiptmair [2003]. Let $\mathbf{u}$ stand for either the electric field $\mathbf{E}$ or the magnetic field $\mathbf{H}$. The *interior Dirichlet trace* $\gamma_{\mathrm{D,int}}\mathbf{u}$ of $\mathbf{u}$ at a point $\mathbf{x} \in \Gamma$ is defined as

$$[\gamma_{\mathrm{D,int}}\mathbf{u}](\mathbf{x}) \equiv \mathbf{u}|_{\Gamma,\mathrm{int}}(\mathbf{x}) \times \mathbf{n}(\mathbf{x}), \tag{13}$$

where $\mathbf{n}$ is the outward unit vector normal to Γ at $\mathbf{x}$ and $\mathbf{u}|_{\Gamma,\mathrm{int}}(\mathbf{x})$ is the limit of $\mathbf{u}(\mathbf{y})$ as $\mathbf{y}$ approaches $\mathbf{x}$ from within Ω. The *interior Neumann trace* $\gamma_{\mathrm{N,int}}\mathbf{u}$ at $\mathbf{x} \in \Gamma$ is defined as

$$[\gamma_{\mathrm{N,int}}\mathbf{u}](\mathbf{x}) \equiv \frac{1}{\mathrm{i}k}(\boldsymbol{\nabla} \times \mathbf{u})|_{\Gamma,\mathrm{int}}(\mathbf{x}) \times \mathbf{n}(\mathbf{x}). \tag{14}$$

The *exterior* traces are defined analogously. Both the Dirichlet and Neumann trace belong to the Sobolev space $\mathbf{H}_{\times}^{-1/2}(\mathrm{div}_\Gamma, \Gamma)$ defined in Buffa and Hiptmair [2003].

Owing to the duality between the electric and magnetic field, only two integral operators are needed rather than four as in the Laplace case: the single-layer potential operator $\boldsymbol{\Psi}_{\mathrm{SL},k}$ and the double-layer potential operator $\boldsymbol{\Psi}_{\mathrm{DL},k}$. They are defined by

$$[\boldsymbol{\Psi}_{\mathrm{SL},k}\mathbf{v}](\mathbf{x}) := \mathrm{i}k \int_\Gamma g_k(\mathbf{x},\mathbf{y})\, \mathbf{v}(\mathbf{y})\, \mathrm{d}\Gamma(\mathbf{y}) - \frac{1}{\mathrm{i}k}\nabla_{\mathbf{x}} \int_\Gamma g_k(\mathbf{x},\mathbf{y})\, (\boldsymbol{\nabla}_\Gamma \cdot \mathbf{v})(\mathbf{y})\, \mathrm{d}\Gamma(\mathbf{y}), \tag{15a}$$

$$[\Psi_{\mathrm{DL},k}\mathbf{v}](\mathbf{x}) := \nabla_{\mathbf{x}} \times \int_\Gamma g_k(\mathbf{x},\mathbf{y})\, \mathbf{v}(\mathbf{y})\, \Gamma(\mathbf{y}). \tag{15b}$$

where

$$g_k(\mathbf{x},\mathbf{y}) \equiv \frac{\exp(\mathrm{i}k|\mathbf{x}-\mathbf{y}|)}{4\pi|\mathbf{x}-\mathbf{y}|} \tag{16}$$

is the Green's function of the Helmholtz equation with wave number k and $\mathbf{v}(\mathbf{x})$ is a vector-valued function defined on a surface Γ.

Taking the interior and exterior Dirichlet and Neumann traces of the Stratton-Chu representation formula [Buffa and Hiptmair 2003, theorem 6], one arrives at the boundary integral equations

$$\left(-\frac{1}{2}\mathbf{I} + \mathbf{C}_k\right)\gamma_{\mathrm{D,int}}\mathbf{u} + \mathbf{S}_k\gamma_{\mathrm{N,int}}\mathbf{u} = 0, \tag{17a}$$

$$-\mathbf{S}_k\gamma_{\mathrm{D,int}}\mathbf{u} + \left(-\frac{1}{2}\mathbf{I} + \mathbf{C}_k\right)\gamma_{\mathrm{N,int}}\mathbf{u} = 0, \tag{17b}$$

where the *single-layer boundary operator* $\mathbf{S}_k : \mathbf{H}_{\times}^{-1/2}(\mathrm{div}_\Gamma, \Gamma) \to \mathbf{H}_{\times}^{-1/2}(\mathrm{div}_\Gamma, \Gamma)$ and *double-layer boundary operator* $\mathbf{C}_k : \mathbf{H}_{\times}^{-1/2}(\mathrm{div}_\Gamma, \Gamma) \to \mathbf{H}_{\times}^{-1/2}(\mathrm{div}_\Gamma, \Gamma)$ denote the averages of the interior and exterior traces of the corresponding potential operators with wavenumber k, and $\mathbf{I}$ stands for the identity operator. Similarly, Maxwell equations in an exterior domain $\mathbb{R}^3 \setminus \Omega$ filled with a material corresponding to wave number k, with the Silver-Muller boundary conditions imposed at infinity, can be reduced to the boundary integral equations

$$\left(\frac{1}{2}\mathbf{I} + \mathbf{C}_k\right)\gamma_{\mathrm{D,ext}}\mathbf{u} + \mathbf{S}_k\gamma_{\mathrm{N,ext}}\mathbf{u} = 0, \tag{18a}$$

$$-\mathbf{S}_k\gamma_{\mathrm{D,ext}}\mathbf{u} + \left(\frac{1}{2}\mathbf{I} + \mathbf{C}_k\right)\gamma_{\mathrm{N,ext}}\mathbf{u} = 0. \tag{18b}$$

2.2. Boundary element spaces

To discretise the Sobolev spaces $H^{-\frac{1}{2}}(\Gamma)$ and $H^{\frac{1}{2}}(\Gamma)$ we introduce the triangulation $\mathcal{T}_h$ of Γ with triangular surface elements τ_ℓ and associated nodes $\mathbf{x}_i$ such that $\overline{\mathcal{T}_h} = \bigcup_\ell \overline{\tau_\ell}$. Here, h denotes the mesh size. We define two spaces of functions.

— The space of *piecewise constant functions* $S_h^0(\Gamma) := \mathrm{span}\{\psi_k^{(0)}\}$ with

$$\psi_k^{(0)}(\mathbf{x}) = \begin{cases} 1 & \text{for } \mathbf{x} \in \tau_k \\ 0 & \text{for } \mathbf{x} \notin \tau_k. \end{cases} \tag{19}$$

— The space of *continuous piecewise linear functions* $S_h^1(\Gamma) := \mathrm{span}\{\phi_j^{(1)}\}$ with

$$\phi_j^{(1)}(\mathbf{x}_i) = \begin{cases} 1 & \text{for } i = j \\ 0 & \text{for } i \neq j. \end{cases} \tag{20}$$

Approximation results for these spaces are given, for instance, in Steinbach [2008, section 10.2].

To discretise the Sobolev space $\mathbf{H}_\times^{-1/2}(\mathrm{div}_\Gamma, \Gamma)$ we use the space of lowest-order Raviart-Thomas functions [Raviart and Thomas 1977].

In the following we distinguish between *shape functions*, defined on a reference element (typically the unit triangle or the unit square), *element-level basis functions*, obtained by mapping the shape functions onto a (single) physical element, and *basis functions*, obtained by joining together one or more element-level basis functions defined on one or more adjacent physical elements. This usage is consistent with e.g. Szabo and Babuška [1991, p. 95] and Šolín [2005, p. 67]. We call the family of all shape functions associated with a particular reference element a *shapeset*. For example, a shapeset associated with the unit triangle with vertices $\mathbf{x}_1 = (0,0)$, $\mathbf{x}_2 = (0,1)$ and $\mathbf{x}_3 = (1,0)$ might be the set of the three linearly independent linear functions $\phi_j^{(1)}$ ($j = 1,2,3$) defined on that triangle and satisfying eq. (20).

2.3. Galerkin discretisation of a Dirichlet problem

Laplace equation. We now describe as an example the Galerkin discretisation of a Dirichlet problem. Here, the boundary data v are given and we need to compute t. Using the first row of (11) we obtain

$$Vt = \left(\frac{1}{2}I + K\right)v. \tag{21}$$

Denote by

$$\langle f, g\rangle := \int_\Gamma \overline{f}(\mathbf{x})\, g(\mathbf{x})\, \mathrm{d}\Gamma(\mathbf{x}) \tag{22}$$

the standard $L^2(\Gamma)$ inner product. The variational formulation of (21) is now given as follows. Find $t \in H^{\frac{1}{2}}(\Gamma)$ such that

$$\langle \psi, Vt\rangle = \left\langle \psi, \left(\frac{1}{2}I + K\right)v \right\rangle \tag{23}$$

for $\psi \in H^{-\frac{1}{2}}(\Gamma)$. By restricting $H^{\frac{1}{2}}(\Gamma)$ to $S_h^1(\Gamma)$ and restricting $H^{-\frac{1}{2}}(\Gamma)$ to $S_h^0(\Gamma)$ we obtain the corresponding discretised Galerkin formulation, which takes the matrix form

$$\mathsf{V}\mathbf{t} = \left(\frac{1}{2}\mathsf{M} + \mathsf{K}\right)\mathbf{v}.$$

The matrices are defined by

$$\mathsf{V}_{i,j} = \int_\Gamma \psi_i^{(0)}(\mathbf{x}) \int_\Gamma g(\mathbf{x},\mathbf{y})\, \phi_j^{(0)}(\mathbf{y})\, \mathrm{d}\Gamma(\mathbf{y})\, \mathrm{d}\Gamma(\mathbf{x}),$$
$$\mathsf{M}_{i,j} = \int_\Gamma \psi_i^{(0)}(\mathbf{x})\, \phi_j^{(1)}(\mathbf{x})\, \mathrm{d}\Gamma(\mathbf{x}),$$
$$\mathsf{K}_{i,j} = \int_\Gamma \psi_i^{(0)}(\mathbf{x}) \int_\Gamma g(\mathbf{x},\mathbf{y})\, \phi_j^{(1)}(\mathbf{y})\, \mathrm{d}\Gamma(\mathbf{y})\, \mathrm{d}\Gamma(\mathbf{x}).$$

If the solution t is piecewise sufficiently smooth then the following error estimate holds [Steinbach 2008, Chapter 12]:

$$\|t - t_h\|_{H^{-\frac{1}{2}}(\Gamma)} = \mathcal{O}(h^{\frac{3}{2}}),$$

where t_h is the solution of the discretised variational problem. This reflects the power of Galerkin boundary elements. We achieve superlinear convergence in the right space for the unknown t despite only using piecewise constant basis functions to approximate t.

Note the importance of the concept of dual spaces in the context of Galerkin boundary element methods. The functions on both sides of (21) are elements of $H^{\frac{1}{2}}(\Gamma)$. In order for (23) to be well defined we need that $\psi \in H^{-\frac{1}{2}}(\Gamma)$, the dual space of $H^{\frac{1}{2}}(\Gamma)$. BEM++ understands this notion of dual spaces of range spaces, and requires the user to define the domain, the range, and the dual-to-range space for linear operators.

A drawback of Galerkin boundary elements compared to collocation methods is that the corresponding matrix elements are expensive to evaluate. The computation of V and K requires the evaluation of four-dimensional integrals over singular kernels if the support elements of the basis functions interface or intersect each other. Fast numerical quadrature rules have been developed to deal with this problem (see e.g. Sauter and Schwab [2011], Chernov et al. [2011], Chernov and Schwab [2012] and Polimeridis et al. [2013]). In special cases semi-analytical [Rjasanow and Steinbach 2007; Polimeridis and Mosig 2010], or even fully analytical [Lenoir and Salles 2012], rules have also been developed.

Maxwell equations. Following Buffa and Hiptmair [2003], the Galerkin weak forms of the operators $\mathbf{S}_k$ and $\mathbf{C}_k$ are defined with respect to the antisymmetric pseudo-inner product

$$\langle \mathbf{u}, \mathbf{v} \rangle_{\tau,\Gamma} \equiv \int_\Gamma \overline{\mathbf{u}}(\mathbf{x}) \cdot [\mathbf{v}(\mathbf{x}) \times \mathbf{n}(\mathbf{x})]\, \mathrm{d}\Gamma(\mathbf{x}). \tag{24}$$

Explicit expressions for the weak forms of $\mathbf{S}_k$ and $\mathbf{C}_k$ are given in eqs. (32) and (33) from Buffa and Hiptmair [2003] (the former needs to be multiplied by i to adapt it to the convention used in BEM++).

3. AN OVERVIEW OF BEM++

3.1. General structure

The BEM++ library is composed of five major parts, schematically illustrated in fig. 1.

The *Grid* module is responsible for grid management. It is essentially a wrapper of the Dune-FoamGrid library [Gräser and Sander 2012], which provides an implementation of the abstract grid interface defined by the Dune-Grid package.

The *Fiber* (**F**ast **I**ntegration **B**oundary **E**lement **R**outines) module is a key component of the library, incorporating most of its low-level functionality. It is responsible for the *local assembly*, i.e. the evaluation of boundary-element integrals on single elements or pairs of elements, without taking into account their connectivity. In addition

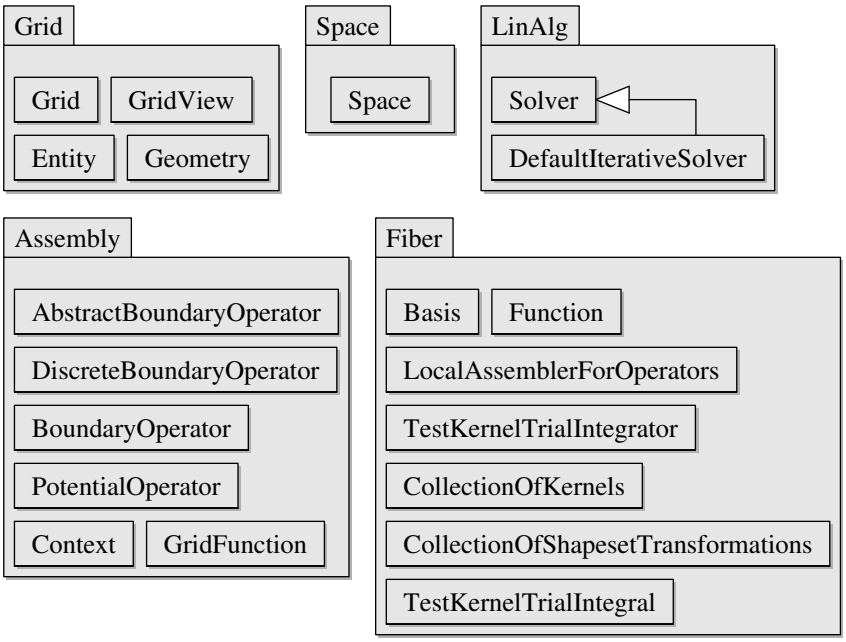

Fig. 1. The five modules of BEM++, together with their most important classes.

to classes performing the actual integration, *Fiber* defines a set of interfaces representing elements of weak forms of boundary-integral operators, such as kernels and shape function transformations, which will be discussed in section 3.9. This module is independent from the rest of BEM++ except for a small set of auxiliary header files used throughout the library. It could therefore be used in separate boundary-element codes, providing the most basic functionality common to all boundary-element libraries—evaluation of elementary integrals. With this in mind, the members of *Fiber* are defined in separate C++ namespace, `Fiber`, rather than the `Bempp` namespace used in the rest of BEM++.

The *Space* module consists of the `Space` class and its derivatives. A `Space` represents a space of functions defined on the elements of a grid. It provides a mapping between those elements and shapesets (`Fiber::Shapeset` objects) defined on the corresponding reference elements. It also acts as a degree-of-freedom manager, using its knowledge of element-to-element connectivity and the function space continuity properties to generate a mapping from local to global degrees of freedom and vice versa. Table I lists the main spaces which are currently available in BEM++. Some of these spaces have additional "discontinuous" and "barycentric" variants not included in the table. The basis functions of a "discontinuous" space are identical to the element-level basis functions of its non-"discontinuous" version. "Barycentric spaces", e.g. the space `PiecewiseLinearContinuousScalarSpaceBarycentric`, have the same basis functions as their non-barycentric counterparts, but are defined over a barycentric refinement of a grid. These additional spaces are not normally created explicitly by the user, but are needed internally. For example, "barycentric" spaces are used during the construction of opposite-order preconditioners (see section 4.1).

The *Assembly* module is the largest part of the library. It defines classes representing integral operators and functions defined on grids, which will be discussed in sections 3.3–3.6. In particular, it contains the code responsible for the *global assembly*, i.e. the

Table I. Main spaces available in BEM++

Name	Description
`PiecewiseConstantScalarSpace`	Space $S_h^{(0)}$ of piecewise constant functions.
`PiecewiseConstantDualGridScalarSpace`	Space of piecewise constant functions defined on the dual grid.
`PiecewiseLinearContinuousScalarSpace`	Space $S_h^{(1)}$ of continuous piecewise linear functions.
`PiecewiseLinearDiscontinuousScalarSpace`	Space of element-wise linear functions.
`PiecewisePolynomialContinuousScalarSpace`	Space of continuous piecewise polynomial functions.
`PiecewisePolynomialDiscontinuousScalarSpace`	Space of element-wise polynomial functions.
`RaviartThomas0VectorSpace`	Space of lowest-order Raviart-Thomas basis functions.
`UnitScalarSpace`	Space of globally constant functions.

formation of matrices of discretised operators from elementary integrals produced by the *Fiber* module.

Finally, the *LinAlg* module provides interfaces to a range of linear solvers. These will be briefly discussed in section 3.7.

3.2. Design for scriptability

A major goal in the development of BEM++ was to provide Python bindings in addition to the core C++ interface. This aim had a significant influence on the overall structure of the BEM++ code.

Numerous scientific libraries written in C++, such as the BEM codes HyENA and BETL and the grid-management library Dune, make heavy use of C++ templates to maximise performance. In such codes, quantities such as integration order, element shape or integral operator type tend to be parameters of class or function templates and are determined at compile time. The total number of possible variants of any template can be very large, but in a particular user program a template is actually instantiated only for a small number of parameter combinations.

This code flavour is perfectly reasonable for libraries intended to be used from C++ only; however, it becomes much less convenient when scripting-language interfaces are to be developed. A scripting-language wrapper of a C++ library typically requires access to a binary containing the compiled version of all the C++ code it may need to execute. For template-based libraries, this means that the templates need to be explicitly instantiated for all the permissible parameter combinations. Since the number of these grows exponentially with the number of template parameters, scriptable C++ codes need to use templates sparingly. For this reason, BEM++ relies mostly on dynamic polymorphism (technically implemented with virtual functions) and restricts the use of templates to two areas.

First, many classes in BEM++ are templates parametrised by the type used to represent values of (scalar components of) basis functions, and/or the type used to represent values of (scalar components of) functions produced by integral operators acting on these basis functions. These parameters are usually called `BasisFunctionType` and `ResultType`, respectively. Occasionally, other parameter names, such as `ValueType`, are also used. The parameters are allowed to take at most four values—`float`, `double`, `std::complex<float>` and `std::complex<double>`—corresponding to the single- and double-precision real and complex numbers, and the templates are explicitly instantiated for each sensible combination of these parameters. This gives at most eight different variants (mixing different precisions is not allowed).

Second, to improve performance and reduce code duplication, in some low-level code we combine coarse-grained dynamic polymorphism and fine-grained static polymor-

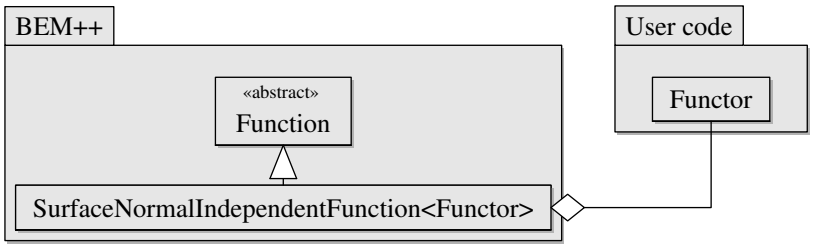

Fig. 2. Dependencies between some of the classes related to evaluation of user-define functions.

phism. This is most easily shown with an example. In the process of assembling the right-hand side of an integral equation, the user typically needs to expand a known function f, defined analytically or by interpolation of experimental data, in a boundary-element space. In BEM++, such functions are represented with classes derived from the abstract `Function` base class. The latter contains, in particular, the virtual function `Function::evaluate()`, which takes the geometrical data (e.g. global coordinates, surface normals etc.) associated with a list of points and is expected to produce the list of values of f at these points. The coarse-grained nature of this interface—with several evaluations of f per virtual-function call—helps to reduce the overhead due to dynamic polymorphism. However, it comes at the price of increased complexity of implementation: the body of the `evaluate()` method of every concrete subclass of `Function` now needs to contain a loop over the supplied points.

To remedy this, BEM++ provides a number of class templates, such as `SurfaceNormalIndependentFunction`, derived from `Function` and parameterised with the name of a user-defined functor class (see fig. 2). This class should provide an `evaluate()` method able to calculate f at a *single* point. The implementation of `SurfaceNormalIndependentFunction::evaluate()` loops over the supplied points and calls the `evaluate()` method of the functor object for each of these points separately, gathering the results and storing them in an array that is subsequently returned to the caller.

This mechanism has several advantages. The use of static polymorphism on the fine-grained level allows us to the reduce amount of code that needs to be written by user to the bare minimum (evaluation of f a single point), which simplifies development and limits the room for errors. It also improves performance, as the calls to the functor's `evaluate()` method can be inlined and potentially automatically vectorised by the compiler. On the other hand, the presence of the abstract `Function` class prevents the "spill-out" of the functor-type template parameter into other fragments of the library and the ensuing combinatorial explosion of the number of necessary template instantiations. It also permits us to provide a separate set of subclasses of `Function` that implement the virtual `evaluate()` method by calling a user-defined *Python* function. For the purposes of code external to the `Function` hierarchy, there is no difference between functions defined in C++ and in Python.

A similar approach (abstract base class + derived class templates parametrised with functors) is also used to represent terms occurring in boundary integrals, such as kernels or shape function transformations (standing for e.g. element-level basis functions or their curls). This will be discussed further in section 3.9.

3.3. Abstract and discrete boundary operators

BEM++ distinguishes between two types of boundary operators: *abstract* and *discrete* ones.

Abstract operators, subclasses of the `AbstractBoundaryOperator` class, represent boundary operators in their strong form. An abstract operator is a mapping $L : X_h \rightarrow$

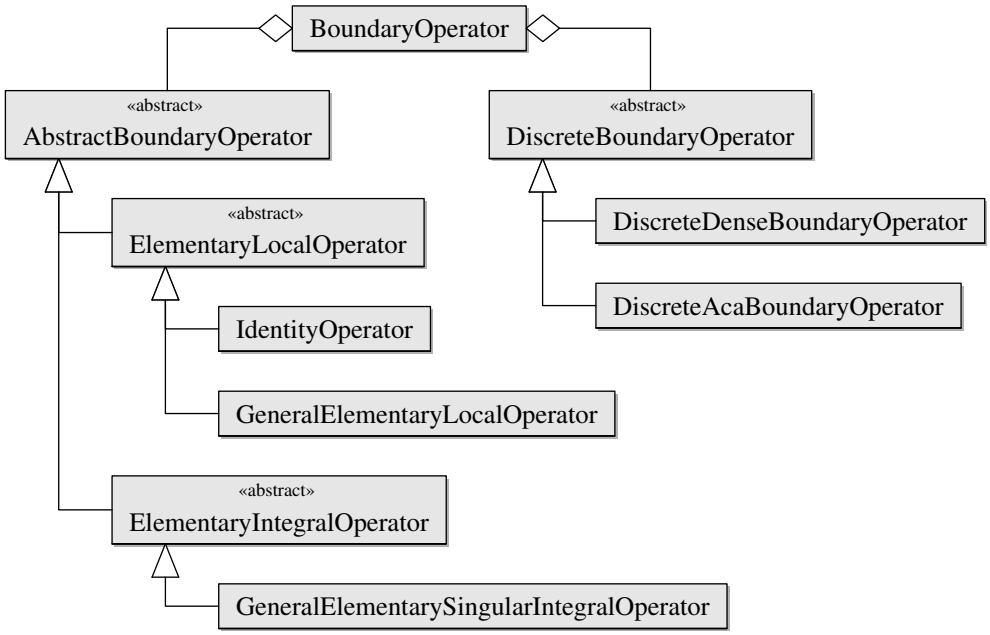

Fig. 3. Relationships between the main classes representing boundary operators.

Y_h, where the domain X_h and the range Y_h are two (finite-dimensional) spaces of *functions* defined on surfaces Γ and Σ (X_h and Y_h may be equal). Discrete operators, subclasses of the `DiscreteBoundaryOperator` class, represent boundary operators in their Galerkin weak form. The weak form of L is obtained by applying it to each basis function of the *trial space* X_h and projecting the result on the basis functions of the *test space* Y_h' dual to Y_h. This yields a matrix L (see section 2.3). Interpreted as an operator, this matrix $\mathsf{L} : \mathbb{C}^m \to \mathbb{C}^n$ acts on (algebraic) *vectors* of dimension $m = \dim X_h$ and produces vectors of dimension $n = \dim Y_h'$.

Abstract operators can be divided into two main categories: *local* and *non-local* operators. Let $f(\mathbf{x})$ be a function from X_h; we say that L is local if $(Lf)(\mathbf{x})$ depends only on the values of f in an infinitesimal neighbourhood of $\mathbf{x}$. The identity operator and differential operators, such as the Laplace-Beltrami operator, are local and their discretised weak forms are sparse matrices. Conversely, integral operators are in general non-local and their discretised weak forms are dense matrices.

Figure 3 depicts the relationships between the `BoundaryOperator`, `AbstractBoundaryOperator` and `DiscreteBoundaryOperator` classes, showing also some subclasses of the latter two.

It is obviously possible to use the discrete operators directly to solve a given boundary-element problem. However, BEM++ provides also a higher-level interface, in which direct access to discrete operators is not necessary. This allows programs to be written in a manner following more closely the simpler strong-form formulation of problems.

With this aim in mind, BEM++ defines the `BoundaryOperator` class, which acts as a wrapper of a pair of shared pointers referencing an `AbstractBoundaryOperator` and its discretised version, a `DiscreteBoundaryOperator`. The second pointer is at first null and is initialised only after the first call to `BoundaryOperator::weakForm()`. To create a `BoundaryOperator` object representing a standard integral operator, the user calls a non-member constructor function, e.g.

Table II. Functions that construct the elementary operators currently defined in BEM++.

Function	Weak form
`identityOperator()`	$\int_\Gamma \overline{\phi}(\mathbf{x})\, \psi(\mathbf{x})\, \mathrm{d}\Gamma(\mathbf{x})$
`maxwell3dIdentityOperator()`	$\int_\Gamma \overline{\boldsymbol{\phi}}(\mathbf{x}) \cdot [\boldsymbol{\psi}(\mathbf{x}) \times \boldsymbol{n}(\mathbf{x})]$
`laplaceBeltrami3dOperator()`	$\int_\Gamma \boldsymbol{\nabla}_\Gamma \overline{\phi}(\mathbf{x}) \cdot \boldsymbol{\nabla}_\Gamma \psi(\mathbf{x})\, \mathrm{d}\Gamma(\mathbf{x})$
`laplace3dSingleLayerBoundaryOperator()`	$\int_\Gamma \int_\Sigma \overline{\phi}(\mathbf{x})\, g(\mathbf{x},\mathbf{y})\, \psi(\mathbf{y})\, \mathrm{d}\Gamma(\mathbf{x})\, \mathrm{d}\Sigma(\mathbf{y})$
`laplace3dDoubleLayerBoundaryOperator()`	$\int_\Gamma \int_\Sigma \overline{\phi}(\mathbf{x})\, \partial_{\mathbf{n}(\mathbf{y})} g(\mathbf{x},\mathbf{y})\, \psi(\mathbf{y})\, \mathrm{d}\Gamma(\mathbf{x})\, \mathrm{d}\Sigma(\mathbf{y})$
`laplace3dAdjointDoubleLayerBoundaryOperator()`	$\int_\Gamma \int_\Sigma \overline{\phi}(\mathbf{x})\, \partial_{\mathbf{n}(\mathbf{x})} g(\mathbf{x},\mathbf{y})\, \psi(\mathbf{y})\, \mathrm{d}\Gamma(\mathbf{x})\, \mathrm{d}\Sigma(\mathbf{y})$
`laplace3dHypersingularBoundaryOperator()`	$\int_\Gamma \int_\Sigma \mathbf{curl}_\Gamma\, \overline{\phi}(\mathbf{x}) \cdot g(\mathbf{x},\mathbf{y})\ \mathbf{curl}_\Sigma\, \psi(\mathbf{y})\, \mathrm{d}\Gamma(\mathbf{x})\, \mathrm{d}\Sigma(\mathbf{y})$
`helmholtz3dSingleLayerBoundaryOperator()`	$\int_\Gamma \int_\Sigma \overline{\phi}(\mathbf{x})\, g_k(\mathbf{x},\mathbf{y})\, \psi(\mathbf{y})\, \mathrm{d}\Gamma(\mathbf{x})\, \mathrm{d}\Sigma(\mathbf{y})$
`helmholtz3dDoubleLayerBoundaryOperator()`	$\int_\Gamma \int_\Sigma \overline{\phi}(\mathbf{x})\, \partial_{\mathbf{n}(\mathbf{y})} g_k(\mathbf{x},\mathbf{y})\, \psi(\mathbf{y})\, \mathrm{d}\Gamma(\mathbf{x})\, \mathrm{d}\Sigma(\mathbf{y})$
`helmholtz3dAdjointDoubleLayerBoundaryOperator()`	$\int_\Gamma \int_\Sigma \overline{\phi}(\mathbf{x})\, \partial_{\mathbf{n}(\mathbf{x})} g_k(\mathbf{x},\mathbf{y})\, \psi(\mathbf{y})\, \mathrm{d}\Gamma(\mathbf{x})\, \mathrm{d}\Sigma(\mathbf{y})$
`helmholtz3dHypersingularBoundaryOperator()`	$\int_\Gamma \int_\Sigma g_k(\mathbf{x},\mathbf{y})[\mathbf{curl}_\Gamma\, \overline{\phi}(\mathbf{x}) \cdot \mathbf{curl}_\Sigma\, \psi(\mathbf{y})$ $-k^2\, \overline{\phi}(\mathbf{x})\, \mathbf{n}(\mathbf{x}) \cdot \psi(\mathbf{y})\, \mathbf{n}(\mathbf{y})]\, \mathrm{d}\Gamma(\mathbf{x})\, \mathrm{d}\Sigma(\mathbf{y})$
`maxwell3dSingleLayerBoundaryOperator()`	$\int_\Gamma \int_\Sigma g_k(\mathbf{x},\mathbf{y})[-\mathrm{i}k\, \overline{\boldsymbol{\phi}}(\mathbf{x}) \cdot \boldsymbol{\psi}(\mathbf{y})$ $-\frac{1}{\mathrm{i}k}\, \mathrm{div}_\Gamma\, \overline{\phi}(\mathbf{x})\ \mathrm{div}_\Sigma\, \psi(\mathbf{y})]\, \mathrm{d}\Gamma(\mathbf{x})\, \mathrm{d}\Sigma(\mathbf{y})$
`maxwell3dDoubleLayerBoundaryOperator()`	$\int_\Gamma \int_\Sigma \boldsymbol{\nabla}_{\mathbf{x}} g_k(\mathbf{x},\mathbf{y}) \cdot [\overline{\boldsymbol{\phi}}(\mathbf{x}) \times \boldsymbol{\psi}(\mathbf{y})]\, \mathrm{d}\Gamma(\mathbf{x})\, \mathrm{d}\Sigma(\mathbf{y})$

```
template <typename BasisFunctionType, typename ResultType>
BoundaryOperator<BasisFunctionType, ResultType>
laplace3dSingleLayerBoundaryOperator(
        const shared_ptr<const Context<
            BasisFunctionType, ResultType> >& context,
        const shared_ptr<const Space<BasisFunctionType> >& domain,
        const shared_ptr<const Space<BasisFunctionType> >& range,
        const shared_ptr<const Space<BasisFunctionType> >& dualToRange,
        const std::string& label = "",
        int symmetry = NO_SYMMETRY);
```

The key parameters are the first four ones. The parameter `context` controls the details of subsequent operator discretisation and will be described in section 3.4. The next three parameters are the spaces representing the domain of the abstract operator, its range and the space dual to the range. They might for example be chosen as the space of functions piecewise constant or piecewise linear on elements making up a specific grid. The domain and the space dual to the range are used during subsequent discretisation of the operator as the trial and test space, respectively. The range, in turn, is used in the creation of functions obtained by acting with the operator on already defined functions. This will be covered in more detail in section 3.5.

Most predefined abstract operators in BEM++ are represented with instances of the `GeneralSingularIntegralBoundaryOperator` or `GeneralLocalBoundaryOperator` class, which will be presented in detail in sections 3.9 and 3.10. For example, the `laplace3dSingleLayerBoundaryOperator()` function discussed above creates a new instance of `GeneralSingularIntegralBoundaryOperator` and wraps it in a `BoundaryOperator` object, which is then returned to the caller. The mechanism of construction of other operators is completely analogous, as will be evident from the examples presented in section 4. Table II lists the elementary boundary operators that are currently defined in BEM++.

Both in the C++ and Python interface to BEM++, the arithmetic operators (`+`, `-`, `*` and `/`) acting on `BoundaryOperators` are overloaded. Thus, the user can easily create composite operators representing linear superpositions of elementary ones, as in the code below:

```
typedef BoundaryOperator<double, double> BO;
BO I = identityOperator<double, double>(...);
BO K = laplace3dDoubleLayerBoundaryOperator<double, double>(...);
BO combined = 0.5 * I + K;
```

It is also possible to create operators consisting of several blocks in order to solve systems of boundary integral equations. Such operators are represented with the `BlockedBoundaryOperator` class, whose instances are treated analogously to those of `BoundaryOperator`—for example, the user can pass them to solver classes or extract their discrete weak forms. Example code creating a blocked boundary operator will be presented in section 4.2.

3.4. Construction of discrete weak forms

The discrete weak form of a given boundary operator is created on the first call to its `weakForm()` method, which returns a shared pointer to a `DiscreteBoundaryOperator` object. The details of the discretisation procedure are controlled by the `Context` object previously passed to the constructor of the `BoundaryOperator`. It is essentially a combination of two more specialised objects: `QuadratureStrategy` and `AssemblyOptions`. The former defines the strategy used to evaluate the element-by-element integrals occurring in the entries of the discretised weak form of the operator. Currently BEM++ offers a single concrete subclass of `QuadratureStrategy`: the `NumericalQuadratureStrategy`, which implements Sauter-Schwab quadrature rules [Sauter and Schwab 2011]. More information about the quadrature-related classes in BEM++ will be given in section 3.11. The `AssemblyOptions` object controls higher-level aspects of the weak-form assembly. Most importantly, it determines whether the ACA algorithm is used to accelerate the assembly and to reduce the memory consumption. `AssemblyOptions` can also be used to switch between serial and parallel assembly.

After the construction of a weak form, a shared pointer to it is stored in the `BoundaryOperator` object, so that any further calls to the `weakForm()` method do not trigger the costly recomputation of the discrete operator. Moreover, the implementation of `BoundaryOperator` ensures that all copies of a given `BoundaryOperator` (objects generated by calling its copy constructor) share a single `DiscreteBoundaryOperator` representing their common weak form, regardless of whether these copies are made before or after the discretisation. Thus if, for instance, a given elementary integral operator is reused in several blocks of a blocked boundary operator, or occurs on both the left- and the right-hand-side of an integral equation, it is discretised only once. Of course, this holds also if the operator is used as part of a more complex expression, e.g. a superposition of several operators. The possibility of this reuse of discrete weak forms relies crucially on the fact that most objects in BEM++, such as those representating grids, function spaces and abstract operators, are immutable.

3.5. Grid functions

Functions defined on surfaces are represented in BEM++ with `GridFunction` objects. As opposed to a `Function`, which can be defined in an arbitrary way (with an analytical formula, interpolation of experimental data etc.), a `GridFunction` is expressed as a superposition of the basis functions of a certain space defined on a boundary-element grid. One of the `GridFunction` constructors transforms a `Function` to a `GridFunction`.

The interaction of operators and functions is an area where the strong-form language proves particularly convenient. Consider an operator $A : X \to Y$. We approximate the spaces X, Y and the dual space Y' by the finite-dimensional spaces $X_h := \mathrm{span}\{x_i\}_{i=1}^m$, $Y_h := \mathrm{span}\{y_i\}_{i=1}^n$ and $Y_h' := \mathrm{span}\{y_i'\}_{i=1}^p$. The Galerkin weak-form approximation of A in these finite-dimensional spaces is the matrix $\mathsf{A}_{ij}^{(h)} := \langle y_i', Ax_j \rangle$,

$i = 1, \ldots, p$, $j = 1, \ldots, m$. Now consider a function $f := \sum_{j=1}^{m} f_j^{(h)} x_j \in X_h$ with associated coefficient vector $\mathbf{f}^{(h)}$. Then the result of the action of A on f is a function $g = Af \in Y$. To obtain an approximation $\tilde{g} = \sum_{j=1}^{n} \tilde{g}_j^{(h)} y_j \in Y_h$ of g we compute the vector of *projections*

$$\langle y_i', g \rangle = \langle y_i', Af \rangle = \left[\mathsf{A}^{(h)} \mathbf{f}^{(h)} \right]_i =: \mu_i, \quad i = 1, \ldots, p.$$

We then solve the least-squares problem

$$\langle y_i', \tilde{g} \rangle = \sum_{j=1}^{n} \langle y_i', y_j \rangle \tilde{g}_j^{(h)} = \mu_i, \; i = 1, \ldots, p$$

to obtain the coefficients $\tilde{g}_j^{(h)}$ of $\tilde{g} \in Y_h$, which can be done by multiplying the vector $\boldsymbol{\mu}$ with the pseudoinverse of the mass matrix M with elements $\mathsf{M}_{ij} = \langle y_i', y_j \rangle$. The pseudoinverse takes the form $\mathsf{M}^\dagger = (\mathsf{M}^H \mathsf{M})^{-1} \mathsf{M}^H$ for $p \geq n$ and $\mathsf{M}^H (\mathsf{M}\mathsf{M}^H)^{-1}$ otherwise. In BEM++ the multiplication by pseudoinverse is implemented using sparse direct solves with the product $\mathsf{M}^H \mathsf{M}$ or $\mathsf{M}\mathsf{M}^H$, respectively.

The calculation of the vector of coefficients of a function g generated by an integral operator is necessary whenever g needs to be evaluated or acted upon with another operator. For example, if one wants to evaluate the function $h = ABf$, two such conversions from projections to coefficients are needed. As we have seen, at least when the spaces X, Y and Y' are nonequal, the process involves a fair number of algebraic operations. With its interface modelled after the strong formulation, BEM++ completely encapsulates these manipulations, letting the user obtain the function h simply by writing

```
GridFunction<BasisFunctionType, ResultType> h = A * (B * f);
```

To this end, the `GridFunction` class offers a dual interface. A `GridFunction` can be constructed either from a list of coefficients in a primal space Y_h or projections on the basis of a dual space Y_h' (in the latter case, the primal space also needs to be provided). Similarly, in addition to the `coefficients()` method that returns the vector of coefficients of a given function in its primal space, `GridFunction` provides the `projections(const Space<BasisFunctionType>& dualSpace)` method that calculates on the fly the vector of scalar products of the `GridFunction` with the basis functions of `dualSpace`. Thus, a `GridFunction` can convert freely between its primal and dual representation.

3.6. Potential operators

In sections 3.3 and 3.4 we discussed the classes representing boundary operators—integral operators defined on $(d-1)$-dimensional surfaces embedded in a d-dimensional space. In order to evaluate the solution of a boundary integral equation problem away from the surface, we need to use the representation formula from eq. (5), containing the *potential operators* $\mathcal{V}$ and $\mathcal{K}$ defined in eq. (4). These potential operators map from the boundary Γ into the domain Ω and are therefore treated differently in BEM++ than the boundary operators V and K, which map from Γ into Γ. Specifically, they are represented with a hierarchy of classes implementing the interface defined by the `PotentialOperator` abstract base class. Its most important member is the `evaluateAtPoints()` function, which applies the operator to a supplied `GridFunction` and evaluates the resulting potential at specified points. The use of potential operators will be illustrated in section 4.2.

3.7. Solution of equations

The discrete operator objects in BEM++ are meant to be easily usable from the Trilinos library. For this reason, the `DiscreteBoundaryOperator` class is derived from `Thyra::LinearOpBase`, which defines the fundamental operator interface in Trilinos. This interface includes, in particular, the `apply()` method implementing the matrix-vector product (more precisely, the $\mathrm{y} := \alpha\mathrm{Ax}+\beta\mathrm{y}$ operation). As a result, discretisations of integral operators assembled by BEM++ can be directly passed to a wide range of solvers provided by various components of Trilinos, such as the iterative linear solvers from the Stratimikos-Belos module or the eigensolvers from the Anasazi module.

To facilitate the common task of solving linear equations involving integral operators, BEM++ provides a common high-level interface to the solvers from Stratimikos-Belos, including a GMRES and a CG solver, in the form of the `DefaultIterativeSolver` class. In particular, the `DefaultIterativeSolver` constructor accepts a `BoundaryOperator` (or a `BlockedBoundaryOperator`), rather than its discretisation, while the right-hand side and the solution are passed as `GridFunctions` rather than algebraic vectors. Thus, it is possible to use consistently the strong-form description of an integral-equation problem both during its formulation (construction of constituent boundary operators) and its solution. The only part where a transition to the description in terms of discrete weak forms is necessary is the construction of a preconditioner. This is by design, to allow greater flexibility in adding a preconditioner. An example demonstrating the construction of a preconditioner will be discussed in section 4.2.

3.8. Python interface

The Python bindings to BEM++ are generated using SWIG, a well-known and mature package for connecting programs written in C/C++ with a wide range of high-level programming languages [Beazley 2003; SWIG 2012]. The C++ and Python interfaces to BEM++ are in general very similar; here we will briefly discuss the few aspects handled differently.

The most important difference concerns the construction of objects. As was mentioned in section 3.2, most C++ classes and non-member functions in BEM++ depend on the `BasisFunctionType` and/or `ResultType` template parameters. Since Python has no notion of templates, SWIG generates a separate Python function or class for each allowed combination of these parameters. Thus, for example, the Python function `laplace3dSingleLayerBoundaryOperator_float64_complex128()` acts as a proxy for the C++ function template `laplace3dSingleLayerBoundaryOperator<BasisFunctionType, ResultType>()` instantiated with `BasisFunctionType=double` and `ResultType=std::complex<double> >`. (The declaration of this function template was shown in section 3.3.) It would be cumbersome, though, and go against the weak-typed nature of Python to have to specify these types explicitly each time an object is constructed.

To remedy this, we first of all extend the Python wrappers of all C++ template classes with the additional methods `basisFunctionType()` and/or `resultType()`, which return the (Pythonic) name of the respective type used in the class template instantiation. For example, the method `Laplace3dSingleLayerBoundaryOperator_float64_complex128.basisFunctionType()` returns the string `"float64"`. Second, in the `bempp.lib` module we define a family of factory functions that deduce the exact types of the objects to be constructed from the types of their arguments. For instance, the standard way of constructing a single-layer potential boundary operator in Python is to call the function

```
createLaplace3dSingleLayerBoundaryOperator(
    context, domain, range, dualToRange, label=None)
```

from `bempp.lib`. This function retrieves the basis function type and result type from the `context` object, verifies that the three spaces have the same basis function type, and finally calls the appropriate `laplace3dSingleLayerBoundaryOperator_*_*()` wrapper.

In some cases, the constructor's parameters are not enough to determine some or all of the types. For example, the constructor of the `PiecewiseConstantScalarSpace<BasisFunctionType>` C++ class template takes a single parameter: a shared pointer to a constant `Grid` object. Since `Grid` is not a class template, it does not constrain the value of `BasisFunctionType`. In such cases, the Python factory function takes a `Context` object as an additional parameter; this object is then used to determine the values of all the necessary types. Thus, the signature of the `createPiecewiseConstantScalarSpace()` function is

```
createPiecewiseConstantScalarSpace(context, grid)
```

and the type used to represent the values of the basis function of the newly constructed space is determined by calling `context.basisFunctionType()`.

In the end, therefore, the basis function type and return type must be specified explicitly only once: during the construction of the `NumericalQuadratureStrategy`, which is normally the first BEM++ object to be created. Thus, the factory function

```
createNumericalQuadratureStrategy(
    basisFunctionType, resultType, accuracyOptions)
```

takes strings (`"float64"`, `"complex128"` etc.) as its first two arguments. The use of the factory functions from the `bempp.lib` module will be illustrated by the examples from section 4.

A second area where the two interfaces of BEM++ differ is the construction of `GridFunctions`. The Python interface contains special implementations of the abstract `Function` interface, described in section 3.2, with the `evaluate()` method invoking a Python callable object. Thus, the user can construct a `GridFunction` representing e.g. input Dirichlet or Neumann data simply by writing a Python function evaluating these data at a prescribed point and passing this function to the `createGridFunction()` factory. The actual discretisation process is naturally slower than it would be in pure C++, since it involves repeated callbacks from C++ to Python, but this overhead is normally insignificant in comparison to the total time taken by a boundary-element calculation.

Finally, the third difference concerns the array classes used in both interfaces. In the C++ version, we use mainly the 1-, 2- and 3-dimensional array classes `Col`, `Mat` and `Cube` provided by the Armadillo library [Sanderson 2012]. For technical reasons, low-level code also employs simpler multidimensional array classes `_Array2d`, `_Array3d` and `_Array4d` defined in the *Fiber* module. In the Python bindings, Armadillo arrays are transparently converted into "native" NumPy arrays.

3.9. Low-level representation of integral operators

Basic concepts. BEM++ supports boundary integral operators with almost completely general weak forms. The two basic ingredients of weak forms are *collections of kernels* and *collections of shape function transformations*.

A kernel is a function mapping a pair of points (x, y) located on two, possibly identical, elements of a grid to a scalar, vector or tensor of a fixed dimension, with real or complex elements. It can depend on any geometrical data related to x and y—not only their global coordinates, but also the unit vectors normal to the grid at these points or the Jacobian matrices. A collection of kernels is a set of one or more such kernel functions, which are evaluated together and hence may reuse results of any intermediate calculations.

A shape function transformation is a function mapping a point x located at an element of a grid to a scalar or vector of a fixed dimension, with real or complex elements. In addition to any geometrical data related to x, it can depend on the value and/or the first derivatives of a shape function defined on the reference element (e.g. the unit triangle or the unit square). A shape function transformation can, for example, map shape functions to element-level basis functions or to their surface curls. A collection of shape function transformations is a set of one or more such transformation, again evaluated together.

A boundary integral operator is defined by its characteristic collection of kernels, collection of transformations of test functions, collection of transformations of trial functions, and the overall structure of its weak form, i.e. the way in which the kernels and transformed shape functions are linked together. For example, the weak form of the single-layer potential boundary operator for the Helmholtz equation,

$$\langle \phi, V\psi \rangle = \int_\Gamma \int_\Sigma \underbrace{\frac{\mathrm{e}^{\mathrm{i}k|\mathbf{x}-\mathbf{y}|}}{4\pi|\mathbf{x}-\mathbf{y}|}}_{\mathrm{Kernel}_1} \underbrace{\overline{\phi}(\mathbf{x})}_{\mathrm{TestBFT}_1} \underbrace{\psi(\mathbf{y})}_{\mathrm{TrialBFT}_1} \,\mathrm{d}\Gamma(\mathbf{x})\,\mathrm{d}\Sigma(\mathbf{y}), \tag{25}$$

involves only a single kernel, test and trial function transformation, which are combined by simple scalar multiplication. In the above formula, Γ and Σ are surfaces in $\mathbb{R}^3$, which may, but need not, be equal. In contrast, the weak form of the hypersingular operator

$$\begin{aligned}\langle \phi, D\psi \rangle = \int_\Gamma \int_\Sigma \Bigg[& \underbrace{\frac{\mathrm{e}^{\mathrm{i}k|\mathbf{x}-\mathbf{y}|}}{4\pi|\mathbf{x}-\mathbf{y}|}}_{\mathrm{Kernel}_1} \underbrace{\mathbf{curl}_S\, \overline{\phi}(\mathbf{x})}_{\mathbf{TestBFT}_1} \cdot \underbrace{\mathbf{curl}_T\, \psi(\mathbf{y})}_{\mathbf{TrialBFT}_1} \\ & - k^2 \underbrace{\frac{\mathrm{e}^{\mathrm{i}k|\mathbf{x}-\mathbf{y}|}}{4\pi|\mathbf{x}-\mathbf{y}|}}_{\mathrm{Kernel}_2} \underbrace{\overline{\phi}(\mathbf{x})\,\mathbf{n}(\mathbf{x})}_{\mathbf{TestBFT}_2} \cdot \underbrace{\psi(\mathbf{y})\,\mathbf{n}(\mathbf{y})}_{\mathbf{TrialBFT}_2} \Bigg] \mathrm{d}\Gamma(\mathbf{x})\,\mathrm{d}\Sigma(\mathbf{y})\end{aligned} \tag{26}$$

can be decomposed into two kernels and two test and trial function transformations, combined with an integrand of the form

$$\sum_{i=1}^{2} \mathrm{Kernel}_i \mathbf{TestBFT}_i \cdot \mathbf{TrialBFT}_i.$$

In an alternative decomposition, we could use just one kernel (Kernel_1) and an integrand of the form

$$\mathrm{Kernel}_1(\mathbf{TestBFT}_1 \cdot \mathbf{TrialBFT}_1 - k^2\, \mathbf{TestBFT}_2 \cdot \mathbf{TrialBFT}_2).$$

Abstract interfaces. BEM++ defines abstract interfaces representing the concepts defined above. For instance, it introduces the `CollectionOfKernels` abstract base class, whose declaration (slightly abridged) looks as follows:

```
template <typename ValueType_>
class CollectionOfKernels
{
public:
    typedef ValueType_ ValueType;
    typedef typename ScalarTraits<ValueType>::RealType CoordinateType;

    virtual void addGeometricalDependencies(
```

```
            size_t& testGeomDeps, size_t& trialGeomDeps) const = 0;

    virtual void evaluateAtPointPairs(
            const GeometricalData<CoordinateType>& testGeomData,
            const GeometricalData<CoordinateType>& trialGeomData,
            CollectionOf3dArrays<ValueType>& result) const = 0;

    virtual void evaluateOnGrid(
            const GeometricalData<CoordinateType>& testGeomData,
            const GeometricalData<CoordinateType>& trialGeomData,
            CollectionOf4dArrays<ValueType>& result) const = 0;
};
```

The class is parametrised with the type used to represent the values of kernel function components (`float`, `std::complex<double>` etc.). An implementation of `CollectionOfKernels::addGeometricalDependencies()` is expected to set those bits in the `testGeomDeps` and `trialGeomDeps` bitfields that correspond to the types of geometrical data required by the kernels. This function and analogous functions defined by other elements of the weak form are invoked by the code responsible for the weak-form assembly to determine the full list of geometrical data needed by a particular weak form. The function `evaluateAtPointPairs()` is provided with the geometrical data requested by `addGeometricalDependencies()` corresponding to a list of test points $\{\mathbf{x}_i\}_{i=1}^n$ and a list of trial points $\{\mathbf{y}_i\}_{i=1}^n$ of equal length, and it is expected to store the values of the kernel functions at the point pairs $\{(\mathbf{x}_i, \mathbf{y}_i)\}_{i=1}^n$ in the output argument `result`. In turn, the function `evaluateOnGrid()` obtains geometrical data corresponding to test points $\{\mathbf{x}_i\}_{i=1}^m$ and trial points $\{\mathbf{y}_j\}_{j=1}^n$ and is expected to evaluate the kernel functions at all the point pairs $\{(\mathbf{x}_i, \mathbf{y}_j)\}_{i=1,j=1}^{m\ \ n}$. It is therefore used in the evaluation of integrals with tensor-product quadrature rules, while the former function is used with non-tensor-product rules. The `CollectionOf`n`dArrays` classes are wrappers of lists of n-dimensional arrays, supporting some special operations, such as slicing. The exact format of these arrays is explained in the documentation of `CollectionOfKernels`.

The `CollectionOfShapesetTransformations` interface is of a similar nature. The main difference is that the `evaluate()` function is provided not only with `GeometricalData`, but also with a reference to a `BasisData` object, which encapsulates the values of shape functions and/or their derivatives at evaluation points.

The weak-form integrals over pairs of elements are evaluated by implementations of the `TestKernelTrialIntegral` interface, which defines, in particular, the

```
virtual void evaluateWithNontensorQuadratureRule(
        const GeometricalData<CoordinateType>& testGeomData,
        const GeometricalData<CoordinateType>& trialGeomData,
        const CollectionOf3dArrays<BasisFunctionType>& testTransforms,
        const CollectionOf3dArrays<BasisFunctionType>& trialTransforms,
        const CollectionOf3dArrays<KernelType>& kernels,
        const std::vector<CoordinateType>& quadWeights,
        arma::Mat<ResultType>& result) const = 0;
```

function and a similar `evaluateWithTensorQuadratureRule()` function. These are provided with the data produced by the collections of kernels and test and trial function transformations of a given operator, the list of quadrature weights and, if need be, additional geometrical data, and are expected to fill the matrix `result` with the values of the weak form integral for all pairs of test and trial functions defined on the pair of elements under consideration.

Finally, the `ElementaryIntegralOperator` interface defines the necessary attributes of an integral operator. It is a subclass of `AbstractBoundaryOperator` and introduces five new pure virtual functions:

```
virtual const CollectionOfKernels& kernels() const = 0;
virtual const CollectionOfShapesetTransformations&
    testTransformations() const = 0;
virtual const CollectionOfShapesetTransformations&
    trialTransformations() const = 0;
virtual const TestKernelTrialIntegral& integral() const = 0;
virtual bool isRegular() const = 0;
```

The first four of these should return references to the relevant elements of the weak form of the operator; the fifth informs whether all elements of the collection of kernels of the operator are regular. Currently all integral operators implemented in BEM++ are singular. There exists a dedicated `ElementarySingularIntegralOperator` class that overrides `isRegular()` to return `false`.

Default implementations. As in the case of functions used for the construction of right-hand sides (see section 3.2), a user implementing a new operator does not usually need to implement from scratch a new subclass of any of the abstract base classes discussed above. BEM++ provides default implementations of the classes representing the weak forms and their elements—`DefaultCollectionOfKernels` and so on—declared as class templates parametrised with the name of a functor class. This functor should evaluate the kernel or weak form at a single pair of points, or the shape function transformation at a single point. Several commonly used functors are already defined. For example, `ScalarFunctionValueFunctor` transforms scalar shape functions into element-level basis functions (effectively simply copying the contents of an array into another one), while `SurfaceCurl3dFunctor` calculates the surface curl of an element-level basis function. `SimpleTestScalarKernelTrialIntegrandFunctor` calculates an integrand of the form

$$\mathrm{Kernel}_1 \mathbf{TestBFT}_1 \cdot \mathbf{TrialBFT}_1$$

with a scalar kernel and scalar or vector test and trial function transformations. Finally, `Laplace3dSingleLayerPotentialKernelFunctor`, `Helmholtz3dDoubleLayerPotentialKernelFunctor` etc. evaluate the kernels of particular integral operators.

The implementations of `DefaultCollectionOfKernels::evaluateOnGrid()`, `DefaultCollectionOfKernels::evaluateAtPointPairs()` etc. repeatedly call the provided functor to evaluate the kernel at each quadrature point pair and store the result in the appropriate element of the `result` array. The implementations of analogous `evaluate...()` functions of `DefaultCollectionOfShapesetTransformations` and `DefaultTestKernelTrialIntegral` are done in the same way.

A general-purpose concrete subclass of `ElementarySingularIntegralOperator`, `GeneralElementarySingularIntegralOperator`, is also defined. Its constructor

```
template <typename KernelFunctor, typename TestTransformationsFunctor,
          typename TrialTransformationsFunctor,
          typename IntegrandFunctor>
GeneralElementarySingularIntegralOperator(
        const shared_ptr<const Space<BasisFunctionType> >& domain,
        const shared_ptr<const Space<BasisFunctionType> >& range,
        const shared_ptr<const Space<BasisFunctionType> >& dualToRange,
        const std::string& label,
```

```
            int symmetry,
            const KernelFunctor& kernelFunctor,
            const TestTransformationsFunctor& testTransformationsFunctor,
            const TrialTransformationsFunctor& trialTransformationsFunctor,
            const IntegrandFunctor& integrandFunctor) :
    Base(domain, range, dualToRange, label, symmetry), // call c'tor of base
    m_kernels(
        new Fiber::DefaultCollectionOfKernels<KernelFunctor>(kernelFunctor)),
    ...
```

is a function template taking, apart from the first five standard parameters passed ultimately to the constructor of `AbstractBoundaryOperator`, four functor objects. These are used to create instances of the default implementations of the four elements of the operator's weak form, parametrised with the type of the relevant functor. The instances are stored in internal member variables and the implementations of `kernels()` etc. simply return references to these objects.

To extend the library with a new integral operator, therefore, it is normally enough to define functors representing any elements of the weak form not yet provided by BEM++ (often only a kernel functor) and to write a wrapper for a call to the constructor of `GeneralElementarySingularIntegralOperator`. Indeed, most BEM++ functions returning `BoundaryOperator` objects encapsulating elementary integral operators, e.g. `laplace3dSingleLayerBoundaryOperator()`, `helmholtz3dAdjointDoubleLayerBoundaryOperator()` etc., are implemented in this way.

3.10. Low-level representation of local operators

In contrast to integral operators, whose weak forms are given by integrals over pairs of elements, the weak forms of local operators are integrals over single elements. These operators are represented with subclasses of `ElementaryLocalOperator`. The interface of this class is very similar to `ElementaryIntegralOperator`, and its most important components are the virtual functions

```
virtual const CollectionOfShapesetTransformations&
    testTransformations() const = 0;
virtual const CollectionOfShapesetTransformations&
    trialTransformations() const = 0;
virtual const TestTrialIntegral& integral() const = 0;
```

As in the case of integral operators, there exists a generic implementation of this interface, the `GeneralElementaryLocalOperator` class, whose constructor takes functors used to create instances of `CollectionOfShapesetTransformations` and `TestTrialIntegral`, subsequently returned by the implementation of the above methods. A `GeneralElementaryLocalOperator` object is internally constructed, in particular, by the `laplaceBeltrami3dOperator()` function.

3.11. Quadrature

BEM++ makes it possible to customise the quadrature rules under use at several levels of generality and complexity.

As was mentioned in section 3.1, the *Fiber* module is responsible for the local assembly, i.e. the evaluation of boundary-element integrals on single elements or pairs of elements, without taking into account their connectivity. In particular, in the process of integral operator discretisation, the `QuadratureStrategy` implementation in use creates an object derived from `LocalAssemblerForIntegralOperators`. The `evaluateLocalWeakForms()` method of the latter is then repeatedly called to evaluate the element-by-element integrals contributing to each matrix entry that needs to be calculated.

Thus, one can change completely the method used to evaluate integrals by deriving a new class from `LocalAssemblerForIntegralOperators`, and implementing its `evaluateLocalWeakForms()` method. This class should be accompanied by a new subclass of `QuadratureStrategy`, whose implementation of `makeAssemblerForIntegralOperators()` will return an instance of the custom local assembler class. In this way one could, for example, envisage implementation of semianalytic quadrature rules for a particular family of operators.

`NumericalQuadratureStrategy`, the builtin implementation of `QuadratureStrategy`, is strongly customisable; as long as one is happy with *numerical quadrature*, therefore, implementation of a custom local assembler is probably unnecessary. The quadrature rule used to approximate a particular integral is built in two steps. First, an implementation of the `QuadratureDescriptorSelectorForIntegralOperators` interface constructs a `DoubleQuadratureDescriptor` object that specifies (a) whether the test and trial elements integrated upon share any (and if so, which) vertices or edges and (b) the desired order of accuracy of the quadrature rule. The `QuadratureDescriptorSelectorForIntegralOperators` has access to the geometrical data of the grid (positions of element vertices), so that it can, for example, vary the quadrature order with distance between elements. Second, once a quadrature descriptor has been created, an implementation of the `DoubleQuadratureRuleFamily` interface builds a list of quadrature points and weights making up a quadrature rule of the required order and, if the elements are not disjoint, adapted to the singularity expected from the integrand.

By default, `NumericalQuadratureStrategy` uses an instance of `DefaultQuadratureDescriptorSelectorForIntegralOperators` to construct quadrature descriptors. This class by default makes regular integrals be evaluated with the lowest-order quadrature rule ensuring exact integration of a product of any test and trial functions defined on the given elements. The order of singular quadrature rules is by default chosen to be greater by 5 from that of regular ones. These choices can be modified by passing an `AccuracyOptionsEx` object to a constructor of `NumericalQuadratureStrategy`; in this way it is possible to set quadrature orders to fixed values, increase them by a fixed amount with respect to defaults, or even make the regular quadrature order dependent on the interelement distance. An example of how this can be done will be given in section 4.1. Note that since integral operator weak forms contain kernel functions in addition to test and trial functions, it is a good idea to increase slightly at least the regular quadrature orders even for coarse grids.

The default implementation of `DoubleQuadratureRuleFamily`, `DefaultDoubleQuadratureRuleFamily`, uses tensor products of Dunavant's [1985] rules for triangles to approximate integrals on regular pairs of elements and Sauter-Schwab transformations of tensor (four-dimensional) Gauss-Legendre rules to approximate integrals on singular element pairs.

The user can pass custom instances of `QuadratureDescriptorSelectorFactory` (a factory class used to create `QuadratureDescriptorSelectorForIntegralOperators` objects supplied with geometric data specific to individual operators) or `DoubleQuadratureRuleFamily` to constructors of `NumericalQuadratureStrategy`. These instances will then be used instead of the default implementations. Quadrature rules used in the discretisation of local operators, construction of grid functions and evaluation of potentials can be customised in a similar way, using a parallel hierarchy of classes such as `QuadratureDescriptorSelectorForPotentialOperators` or `SingleQuadratureRuleFamily`.

The number of classes responsible for the selection of quadrature rules may at first seem overwhelming. However, they make it possible to modify selected aspects of the process without reimplementing everything from scratch. For example, here is how specific modifications of the BEM++ integration mechanism might be implemented:

— To set quadrature orders to a fixed amount, change them by a fixed amount or make the regular quadrature order depend on interelement distance, construct an appropriate `AccuracyOptionsEx` object and pass it to a `NumericalQuadratureStrategy` constructor.
— To replace the Sauter-Schwab coordinate transformations [Sauter and Schwab 2011] by the ones proposed by Polimeridis et al. [2013], derive a new class from `DoubleQuadratureRuleFamily` and pass an instance of it to a `NumericalQuadratureStrategy` constructor.
— To make the quadrature order selection dependent on element shape (e.g. raising it for strongly deformed elements), derive new classes from `QuadratureDescriptorSelectorForIntegralOperators` and `QuadratureDescriptorSelectorFactory` and pass an instance of the latter to a constructor of `NumericalQuadratureStrategy`.
— To use semianalytic quadrature rules or to evaluate integrals adaptively, derive a new class from `LocalAssemblerForIntegralOperators` and make it used by a new subclass of `QuadratureStrategy`.

4. EXAMPLES

In this section we will present a number of examples demonstrating the use and capabilities of BEM++. In the first part, we will introduce the Python interface to BEM++ by creating a script solving possibly the simplest problem of all—the Laplace equation in a bounded domain with Dirichlet boundary conditions. We will also discuss the solution of other classes of Laplace problems. In the second part, we will turn our attention to the Helmholtz equation, considering in particular acoustic wave scattering on permeable and non-permeable obstacles. This will allow us to demonstrate the creation of blocked operators and preconditioners and evaluation of solutions away from a discretised surface. surface. In the third part, we will show how to handle mixed (part Dirichlet, part Neumann) boundary conditions. Finally, in the fourth part, we will discuss the solution of Maxwell equations with BEM++.

4.1. Laplace equation with Dirichlet and Neumann boundary conditions

Introduction. In this section we will first develop a Python script using BEM++ to solve eq. (21), the Dirichlet problem for the Laplace equation in a bounded domain $\Omega \in \mathbb{R}^3$ with boundary Γ.

Initialisation. We start by importing the symbols from the `lib` module of the `bempp` package. We also load NumPy, the de-facto standard Python module providing a powerful multidimensional-array data type:

```
from bempp.lib import *
import numpy as np
```

Before creating the operators, we need to specify certain options controlling the manner in which their weak form will be assembled.

The first of them is `QuadratureStrategy`, which determines how individual integrals occurring in the weak forms are calculated. Currently BEM++ only supports numerical quadrature, and thus we construct a `NumericalQuadratureStrategy` object:

```
accuracyOptions = createAccuracyOptions()
accuracyOptions.doubleRegular.setRelativeQuadratureOrder(4)
accuracyOptions.singleRegular.setRelativeQuadratureOrder(2)
quadStrategy = createNumericalQuadratureStrategy(
    "float64", "float64", accuracyOptions)
```

The `createNumericalQuadratureStrategy()` function takes three parameters. The first two are used to determine the `BasisFunctionType` and `ResultType` parameters

described in section 3.2. They can be set to `"float32"`, `"float64"`, `"complex64"` or `"complex128"`, which are the standard NumPy names of single- and double-precision real and complex types. In our case, we want both the basis function values and the values of functions produced by boundary integral operators to be represented with double-precision real numbers, so we set the above parameters to `"float64"`. The last parameter controls the numerical quadrature accuracy, and should be set to an instance of `AccuracyOptions` or `AccuracyOptionsEx`. The numerical quadrature strategy uses three separate families of quadrature rules to perform single integrals of regular functions, double integrals of regular functions and double integrals of singular functions. By default, regular quadrature is done using the least expensive rule ensuring exact integration of a product of a test and a trial function. Since boundary-element integrals contain, in addition, kernel functions, it is often a good idea to increase the regular quadrature orders slightly, as we did in the above code snippet. (A call to `setRelativeQuadratureOrder(`*`delta`*`)` increases the quadrature order by *`delta`* with respect to the default; alternatively, `setAbsoluteQuadratureOrder(`*`order`*`)` may be used to set the order to the fixed value *`order`*.) The default singular quadrature order is usually adequate, so in the above snippet we left it unchanged. If necessary, it can be increased by calling `set...QuadratureOrder()` on the `doubleSingular` member of `AccuracyOptions`.

Higher-level aspects of the weak-form assembly are controlled by `AssemblyOptions` objects. In particular, these determine whether the ACA algorithm is used to accelerate the assembly and to reduce the memory consumption. They can also be used to switch between serial and parallel assembly. To turn on ACA (which is off by default), it suffices to write

```
assemblyOptions = createAssemblyOptions()
acaOptions = createAcaOptions()
assemblyOptions.switchToAca(acaOptions)
```

One can also fine-tune the ACA parameters by editing the `AcaOptions` object—for example, the ACA tolerance can be set to 10^{-5} by inserting

```
acaOptions.eps = 1e-5
```

before the last line of the previous snippet.

The quadrature strategy and assembly options must now be merged into a so-called assembly context:

```
context = createContext(quadStrategy, assemblyOptions)
```

This object encompasses all the parameters influencing operator assembly, including the chosen basis-function and result types. It will be passed to the constructors of most objects created in the sequel.

Grid and spaces. We proceed by loading a triangular grid approximating the surface Γ from a file in the Gmsh [Geuzaine and Remacle 2009; 2012] format:

```
grid = createGridFactory().importGmshGrid("triangular",
                                           "sphere-ico-3.msh")
```

The file `sphere-ico-3.msh` is included in the Supplementary Material and contains a 1280-element triangulation of a sphere with unit radius and centred at the origin.

Now we can define the approximation spaces. As described in section 2.2, we will use the space $S_h^1(\Gamma)$ of continuous, piecewise linear scalar functions to approximate v and the space $S_h^0(\Gamma)$ of piecewice constant scalar functions to approximate t:

```
pconsts = createPiecewiseConstantScalarSpace(context, grid)
```

```
plins = createPiecewiseLinearContinuousScalarSpace(context, grid)
```

Operators. At this point we are ready to create the individual operators. Looking at eq. (21), we see that we need the single- and double-layer potential boundary operators for the 3D Laplace equation and the identity operator:

```
slpOp = createLaplace3dSingleLayerBoundaryOperator(
    context, pconsts, plins, pconsts)
dlpOp = createLaplace3dDoubleLayerBoundaryOperator(
    context, plins, plins, pconsts)
idOp = createIdentityOperator(
    context, plins, plins, pconsts)
```

These three calls produce `BoundaryOperator` objects (combining a reference to an abstract boundary operator with one to its, initially null, discretisation). The meaning of the parameters is the same as for the C++ `laplace3dSingleLayerBoundaryOperator()` function described in section 3.3.

The composite operator $\frac{1}{2}I + K$ occurring on the right-hand side of eq. (21) can be constructed simply by writing

```
rhsOp = 0.5 * idOp + dlpOp
```

since BEM++ provides appropriate overloads of the typical arithmetic operators for `BoundaryOperator` objects. It is important to stress that the result, `rhsOp`, will not store its discrete weak form as a single $\mathcal{H}$-matrix. Instead, invocation of `rhsOp.weakForm()` will trigger discretisation of `idOp` and `dlpOp`, and `rhsOp` operator will only store references (technically, shared pointers) to the resulting weak forms. The matrix-vector product for `rhsOp` will then be realised by processing the results generated by the matrix-vector products of the elementary operators `idOp` and `dlpOp`.

This *implicit* treatment of composite operators mirrors the design of the Thyra module of Trilinos [Bartlett 2007] and makes it easy to construct even very complicated operators. Occasionally the need arises, however, to "compress" a composite operator to a single $\mathcal{H}$-matrix, for example to calculate its H-LU decomposition or to eliminate the memory overhead incurred by storing individual terms of a superposition of operators as separate $\mathcal{H}$-matrices. A way to do it, the `asDiscreteAcaBoundaryOperator()` method, will be presented in section 4.2.

Right-hand side. We now need an object representing the expansion of the known Dirichlet trace v in the space of piecewise linears. We will take v to correspond to the exact solution of the Laplace equation

$$u_{\text{exact}}(\mathbf{x}) = \frac{1}{4\pi|\mathbf{x} - \mathbf{x}_0|} \quad \text{with} \quad \mathbf{x}_0 = (2, 2, 2). \tag{27}$$

We define, therefore, a native Python function

```
def evalDirichletTrace(point):
    x, y, z = point
    dist = np.sqrt((x - 2)**2 + (y - 2)**2 + (z - 2)**2)
    return 1 / (4 * np.pi * dist)
```

receiving an array of coordinates of a single point and returning the value of v at this point. Subsequently, we pass it as the last argument of `createGridFunction()`:

```
dirichletTrace = createGridFunction(
    context, plins, pconsts, evalDirichletTrace)
```

whose declaration looks as follows:

```
def createGridFunction(context, space, dualSpace, callable,
                       surfaceNormalDependent=False)
```

Internally, this routine calculates the vector of projections of the function f defined by the Python callable object `callable` on the basis functions of the space represented by the object `dualSpace`, and then converts this vector into the vector of coefficients of f in `space` in the manner described in section 3.5. In our example, we choose `space` to be a space of piecewise linears and `dualSpace` a space of piecewise constants. The assembly context `context` provides the quadrature strategy used to evaluate the necessary scalar products, while the `surfaceNormalDependent` argument determines whether `callable` needs information about the orientation of the vector normal to the grid at the evaluation point. If so, the components of this vector are passed as the second parameter to `callable`.

To construct the function standing on the right-hand side of eq. (21), we act with the operator `rhsOp` on the Dirichlet trace, using an appropriate overload of the multiplication operator:

```
rhs = rhsOp * dirichletTrace
```

This produces a `GridFunction` expanded in the range space of `rhsOp`, i.e. the space of piecewise linears.

Solution. We have now assembled all the elements of the equation and we are ready to solve it. We will use the wrapper of the GMRES solver from Trilinos provided by BEM++:

```
solver = createDefaultIterativeSolver(slpOp)
solver.initializeSolver(defaultGmresParameterList(1e-8))
solution = solver.solve(rhs)
print solution.solverMessage()
solFun = solution.gridFunction()
```

As can be seen, the solver takes the operator as a `BoundaryOperator` object and the right-hand side as a `GridFunction`; the discretisation is done automatically and there is no need to access the ensuing matrices and vectors explicitly. The solution, the Neumann trace t, is also extracted in the form of a `GridFunction`.

To see a plot of the Neumann trace on Γ, we can use the `bempp.visualization` module, which provides a number of functions for rapid visualization of solutions calculated with BEM++. Internally, it is based on TVTK [Ramachandran 2005], a set of Python bindings of the VTK toolkit [Kitware 2012b]. To plot a single grid function, it suffices to write

```
import bempp.visualization as vis
vis.plotGridFunction(solFun, "cell_data")
```

The string `"cell_data"` indicates that the function should be treated as constant on each element of the grid, which is consistent with the space we have chosen to expand the Neumann trace. The default mode is `"vertex_data"`, which causes the function to be linearly interpolated from its values at vertices of the grid. Figure 4 shows the plot generated by the above code snippet.

For serious data analysis it may be preferable to use a dedicated VTK viewer, such as Paraview [Kitware 2012a]. By calling

```
solFun.exportToVtk("cell_data", "neumann_trace", "solution")
```

one can export the solution to a VTK file `solution.vtu` as a cell-data series labelled `neumann_trace`.

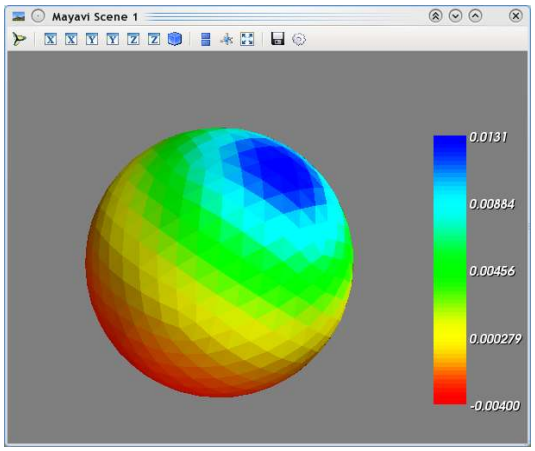

Fig. 4. Neumann trace of the solution calculated with the code presented in section 4.1.

The full code of the above program can be found in the `interior_laplace_dirichlet.py` script included in the Supplementary Material.

Benchmarks. To give an idea about the speed of calculations, table III lists the time used to assemble the discrete weak forms of the two boundary operators and to solve the resulting algebraic equation for a series of triangular grids generated by repeatedly refining an icosahedron and projecting the resulting nodes on the unit sphere. The table also lists the relative error of the obtained solution, defined as $\|v - v_0\|/\|v_0\|$, where v_0 is the exact Neumann trace, v its approximation computed numerically, and $\|\ldots\|$ stands for either the L^2 or the $H^{-\frac{1}{2}}$-norm.[2] The calculations were done on a 12-core, 2.80-GHz Intel workstation. The BEM++ library, built with the GCC 4.4.5 compiler using the `-O3 -m64 -march=native` compilation flags, was linked against the Intel MKL library. Regular Galerkin integrals over element pairs were approximated using quadrature rules of order greater by 4 than default[3] and singular integrals with quadrature rules of the default order of accuracy.[4] Regular integrals over single elements, evaluated during the discretisation of input Dirichlet data, were approximated using quadrature rules of order greater by 2 than default, comprising 4 quadrature points. Iterative solver tolerance was set to 1E$-$8. We have verified that increasing the quadrature rule orders or decreasing the solver tolerance had only a negligible effect on the accuracy of the solution. In contrast, the accuracy was somewhat affected by the ACA tolerance parameter (ϵ), thus for each grid we present results for several values of ϵ. Provided that this parameter is chosen sufficiently small, we recover the theoretically predicted convergence rates of $O(h)$ in the L^2 norm and $O(h^{3/2})$ in the $H^{-1/2}$ norm, with h the element size [Steinbach 2008, pp. 264-265]. In this and all other examples, the η parameter occurring in the admissibility condition used in ACA [Bebendorf 2008, section 1.3] was left at the default value of 1.2.

It can be seen that the weak-form assembly of the double-layer potential boundary operator K takes approximately three times longer than that of the single-layer poten-

[2]The method used to calculate Sobolev-space norms is discussed later in this section.
[3]Specifically, a 6×6-point rule was used to approximate entries of the matrix of operator V, discretised with piecewise constant test and trial functions, and 6×7-point rule was used to approximate entries of the matrix of operator K, discretised with piecewise constant test and piecewise linear trial functions.
[4]Specifically, Sauter-Schwab quadrature rules based on the 3- and 4-point Gauss-Legendre rules were used in the discretisation of operators V and K, respectively.

Table III. Benchmarks for the solution of the Dirichlet problem for the Laplace equation inside a unit sphere. Abbreviations: #Elem. = Number of elements; tol. = tolerance; op. = operator; Mem. = Memory; t = Time; #It. = Number of iterations. The memory use of ACA-compressed operators is listed in megabytes and in percent of the memory use of equivalent dense operators. The number of unknowns is equal to the number of elements.

	ACA	DLP op.		SLP op.		Solver		Relative	Relative
#Elem.	tol.	Mem. (MB / %)	t (s)	Mem. (MB / %)	t (s)	#It.	t (s)	L^2 error	$H^{-\frac{1}{2}}$ error
80	1E−2	0.0 / 100	0.1	0.0 / 100	0.0	15	0.0	8.56E−2	2.95E−2
80	1E−3	0.0 / 100	0.1	0.0 / 100	0.0	15	0.0	8.56E−2	2.95E−2
80	1E−4	0.0 / 100	0.1	0.0 / 100	0.0	15	0.0	8.56E−2	2.95E−2
320	1E−3	0.3 / 81	0.2	0.7 / 96	0.0	24	0.1	4.11E−2	9.34E−3
320	1E−4	0.3 / 88	0.1	0.8 / 97	0.0	23	0.1	4.11E−2	9.33E−3
320	1E−5	0.4 / 99	0.1	0.8 / 99	0.0	23	0.2	4.11E−2	9.33E−3
1280	1E−3	2.6 / 41	0.4	4.2 / 34	0.1	35	0.1	1.98E−2	3.17E−3
1280	1E−4	3.0 / 47	0.4	5.1 / 41	0.1	33	0.1	1.97E−2	3.12E−3
1280	1E−5	3.5 / 55	0.5	6.1 / 49	0.1	33	0.0	1.97E−2	3.12E−3
5120	1E−4	16.0 / 16	2.0	29.4 / 15	0.6	45	0.2	9.72E−3	1.09E−3
5120	1E−5	20.9 / 21	2.3	36.6 / 18	0.7	44	0.3	9.71E−3	1.08E−3
5120	1E−6	26.0 / 26	2.7	45.3 / 23	0.8	43	0.3	9.71E−3	1.08E−3
20480	1E−4	92.1 / 6	10.4	150.3 / 5	3.3	55	1.8	4.92E−3	4.04E−4
20480	1E−5	121.8 / 8	12.6	192.6 / 6	3.9	53	1.7	4.83E−3	3.81E−4
20480	1E−6	155.4 / 10	15.1	244.8 / 8	4.7	53	1.7	4.83E−3	3.81E−4
81920	1E−5	669.5 / 3	69.7	958.6 / 2	20.6	62	10.2	2.41E−3	1.35E−4
81920	1E−6	866.9 / 3	84.7	1238.7 / 2	24.9	61	10.3	2.41E−3	1.34E−4
81920	1E−7	1065.6 / 4	99.5	1541.3 / 3	29.8	61	11.2	2.41E−3	1.34E−4
327680	1E−6	4755.7 / 1	469.5	6004.8 / 1	125.5	68	41.6	1.21E−3	4.75E−5
327680	1E−7	5892.2 / 1	555.1	7515.7 / 1	150.7	68	46.2	1.21E−3	4.75E−5
327680	1E−8	7083.8 / 2	643.5	9124.5 / 1	178.0	68	46.6	1.21E−3	4.75E−5

tial boundary operator V. This is because the trial space of K consists of continuous piecewise linear functions whose supports extend over several elements, so the weak-form assembly requires the evaluation of, on average, three times more elementary integrals.

To complement these data, table IV presents the results obtained for the Neumann problem, formulated as

$$Dv = \left(\frac{1}{2}I - T\right)t; \tag{28}$$

the operators D and T have been defined in section 2.1. As before, v and t are expanded in the spaces of piecewise linears and piecewise constants, respectively. The solution of the Neumann problem for the Laplace equation is determined only up to a constant, so we introduced the constraint

$$\langle 1, v\rangle = \langle 1, \gamma_0^{\text{int}} u_{\text{exact}}\rangle, \tag{29}$$

where 1 denotes the unit function, and solved the problem with the method of Lagrange multipliers. The script `interior_laplace_neumann.py` used to obtain these results is included in the Supplementary Material.[5]

For the Neumann problem, the theoretically predicted convergence rate in the L^2 norm is quadratic in the element size h [Steinbach 2008, p. 280]. At the accuracy levels

[5]The same quadrature orders were used as in the previous example. Regular and singular integrals occurring in the matrix of operator D were evaluated using 4×4-point tensor rules and Sauter-Schwab rules based on the 4-point Gauss-Legendre rule, respectively. In the case of operator T 4×3- and 3-point rules were used.

Table IV. Benchmarks for the solution of the Neumann problem for the Laplace equation inside a unit sphere.
Abbreviations: #Elem. = Number of elements; tol. = tolerance; op. = operator; Mem. = Memory; t = Time; #It. = Number of iterations.
The memory use of ACA-compressed operators is listed in megabytes and in percent of the memory use of equivalent dense operators. The number of unknowns is roughly equal to half of the number of elements.

	ACA	Adjoint DLP op.		Hypersingular op.		Solver			Relative	Relative
#Elem.	tol.	Mem. (MB / %)	t (s)	Mem. (MB / %)	t (s)	Tol.	#It.	t (s)	L^2 error	$H^{\frac{1}{2}}$ error
80	1E−2	0.0 / 100	0.1	0.0 / 101	0.1	1E−8	8	0.0	2.20E−3	3.67E−2
80	1E−3	0.0 / 100	0.1	0.0 / 101	0.1	1E−8	8	0.0	2.20E−3	3.67E−2
320	1E−3	0.3 / 81	0.2	0.2 / 93	0.2	1E−8	10	0.0	4.96E−4	1.24E−2
320	1E−4	0.3 / 88	0.2	0.2 / 100	0.2	1E−8	10	0.0	4.95E−4	1.24E−2
320	1E−5	0.4 / 98	0.2	0.2 / 100	0.2	1E−8	10	0.0	4.95E−4	1.24E−2
1280	1E−4	3.0 / 47	0.4	1.9 / 60	1.0	1E−8	15	0.1	1.20E−4	4.33E−3
1280	1E−5	3.5 / 55	0.5	2.2 / 69	1.1	1E−8	15	0.1	1.20E−4	4.33E−3
1280	1E−6	3.9 / 63	0.5	2.5 / 78	1.3	1E−8	15	0.1	1.20E−4	4.33E−3
5120	1E−5	20.9 / 21	2.4	15.3 / 30	7.3	1E−8	20	0.1	2.97E−5	1.52E−3
5120	1E−6	26.0 / 26	2.9	18.3 / 37	8.3	1E−8	20	0.1	2.97E−5	1.52E−3
5120	1E−7	31.1 / 31	3.3	21.9 / 44	10.0	1E−8	20	0.1	2.97E−5	1.52E−3
20480	1E−6	155.3 / 10	16.6	116.2 / 15	49.4	1E−8	26	0.3	7.42E−6	5.38E−4
20480	1E−7	189.2 / 12	19.2	139.9 / 17	57.8	1E−8	26	0.3	7.40E−6	5.38E−4
20480	1E−8	226.1 / 14	22.3	164.5 / 21	66.1	1E−8	26	0.4	7.40E−6	5.38E−4
81920	1E−6	867.1 / 3	100.8	659.5 / 5	281.5	1E−10	46	3.3	2.73E−6	1.91E−4
81920	1E−7	1065.5 / 4	116.9	799.1 / 6	325.9	1E−10	46	3.4	1.85E−6	1.90E−4
81920	1E−8	1277.5 / 5	135.9	947.9 / 7	374.0	1E−8	35	2.4	1.99E−6	1.90E−4
						1E−10	46	3.7	1.85E−6	1.90E−4
327680	1E−8	7084.3 / 2	1046.7	5296.9 / 3	2443.5	1E−10	62	32.5	4.63E−7	6.72E−5
327680	1E−9	8363.6 / 2	1051.3	6193.5 / 3	2583.9	1E−10	62	31.3	4.62E−7	6.72E−5

reached for larger grids, parameters that had a negligible influence on the solution of the Dirichlet problem start to play a role. In particular, for the grids with 81920 and more elements, fully converged results are only obtained for a tightened solver tolerance. With these adjustments, the simulations reproduce the convergence rates predicted theoretically.

Let us now consider an example with a more complex geometry. Calculation of the distribution of charge on the surface of an electric conductor held at a fixed potential $v(\mathbf{x})$ requires the solution of the *exterior* Dirichlet problem for the Laplace equation. The latter can be transformed to the integral equation

$$V\gamma_1^{\text{ext}}u = \left(-\frac{1}{2}I + K\right)\gamma_0^{\text{ext}}u, \tag{30}$$

which involves the exterior Dirichlet and Neumann traces $\gamma_0^{\text{ext}}u = g$ and $\gamma_1^{\text{ext}}u$. As shown in elementary texts (see e.g. Griffiths [1998]), the surface charge density σ is proportional to the Neumann trace of the electrostatic potential on the conductor surface, $\gamma_1^{\text{ext}}u$. Figure 5 shows the map of σ on the surface of a bolt held at a constant potential. Clearly visible is the concentration of surface charge in the vicinity of edges and vertices. The calculation was done on a 154,076-element mesh derived from a CAD model obtained from [OpenCASCADE 2012]. The unknown Neumann data were expanded in piecewise constant basis functions. Regular Galerkin integrals were approximated using quadrature rules of order greater by 4 than default, and singular integrals with quadrature rules of the default order of accuracy. In view of the elongated shape of the object, to improve load balancing between processor cores, we limited the maximum $\mathcal{H}$-matrix block size to $\frac{1}{8}$ of the number of unknowns. With ACA tolerance set to 1E−6 and GMRES tolerance set to 1E−8, the solution of this problem

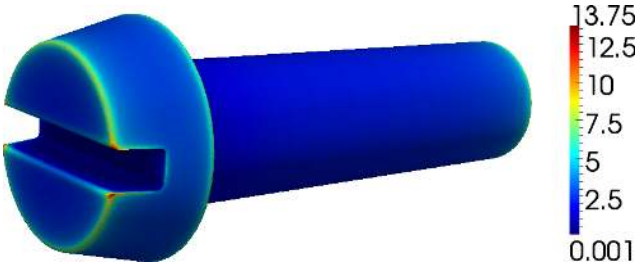

Fig. 5. Charge density (in nC/m^2) on the surface of a 13-mm-long bolt held at the potential $u = 1\,\mathrm{V}$.

took 402 s. The script `bolt.py`, together with the mesh `bolt.msh`, is included in the Supplementary Material.

Sobolev space norms. It is sometimes useful to calculate the $H^{-\frac{1}{2}}(\Gamma)$- or $H^{\frac{1}{2}}(\Gamma)$-norm of the numerical solution or its error with respect to an analytical solution. One definition of the $H^{-\frac{1}{2}}(\Gamma)$-norm, particularly convenient for computations, reads [Langer and Steinbach 2007, p. 74]

$$\|u\|_{H^{-1/2}(\Gamma)} = \sqrt{\langle u, Vu\rangle}, \tag{31}$$

where V is the single-layer potential boundary operator and $\langle\cdot,\cdot\rangle$ denotes the standard $L^2(\Gamma)$ inner product as defined in eq. (22). Using this definition, we can easily calculate the $H^{-\frac{1}{2}}(\Gamma)$-norm of a function u expressed as a `GridFunction` in BEM++. Let $u = \sum_{n=1}^{N} \phi_n u_n$, where u_n are the expansion coefficients of u in the basis $\{\phi_n\}_{n=1}^{N}$ of a finite-dimensional subspace W_h of $H^{-\frac{1}{2}}(\Gamma)$. Then

$$\|u\|_{H^{-1/2}(\Gamma)} = \sqrt{\sum_n \overline{u}_n \langle \phi_n, Vu\rangle}. \tag{32}$$

The numbers $\sqrt{\langle \phi_n, Vu\rangle}$ are the projections of Vu on the basis of W_h. Thus, we can calculate, for example, the $H^{-\frac{1}{2}}(\Gamma)$-norm of the solution `solFun` of the Dirichlet problem discussed earlier in this section using

```
H_norm = np.sqrt(np.dot(solFun.coefficients(),
                        (slpOp * solFun).projections(solFun.space())))
```

Here, the `dot` function from NumPy is called to calculate the inner product of two algebraic vectors.

Calculation of the $H^{-\frac{1}{2}}(\Gamma)$-norm of the error $u - u_{\text{exact}}$, where u is a numerical and u_{exact} an analytical solution, is slightly more involved. To obtain an accurate result, it is necessary to expand the solutions in a space Y_h larger than W_h; otherwise one obtains only the norm of the difference between u and the best approximation of u_{exact} in W_h. When W_h is a space of piecewise constants, a reasonable choice for V_h is the space of discontinuous piecewise linears defined on the same grid. An example implementation of this procedure looks as follows:

```
pdlins = createPiecewiseLinearDiscontinuousScalarSpace(context, grid)
subsamplingOp = createIdentityOperator(
    context, pconsts, pdlins, pdlins)
subsampledSolFun = subsamplingOp * solFun
```

(in the above fragment we expand `solFun`, originally defined in `pconsts`, in the larger space `pdlins`)

```
subsampledExactSolFun = createGridFunction(
    context, pdlins, pdlins, evalExactNeumannTrace,
    surfaceNormalDependent=True)
subsampledErrorFun = subsampledSolFun - subsampledExactSolFun
pquads = createPiecewisePolynomialContinuousScalarSpace(
    context, grid, 2)
subsampledSlpOp = createLaplace3dSingleLayerBoundaryOperator(
    context, pdlins, pquads, pdlins, "sSLP")
H_norm = np.sqrt(np.dot(
    subsampledErrorFun.coefficients(),
    (subsampledSlpOp * subsampledErrorFun).projections(
        subsampledErrorFun.space())))
```

The last column of table III lists the $H^{-\frac{1}{2}}$-norms of the error obtained when solving the Dirichlet problem with various discretisation parameters.

The $H^{\frac{1}{2}}$-norm can be calculated in a very similar way, using the hypersingular operator instead of the single-layer potential operator. The file `interior_laplace_neumann.py` contains an implementation of this procedure, and table IV lists the $H^{\frac{1}{2}}$-norms of the solution errors obtained with various discretisation parameters.

Higher-order basis functions. Starting from version 1.9 BEM++ supports higher-order polynomial basis functions (up to order 10). Although curvilinear elements are not implemented yet, high-order basis functions can be used to speed up calculations on piecewise flat surfaces. A `Space` object representing the function space spanned by piecewise polynomial, continuous or discontinuous, basis function of order d can be constructed by calling

```
createPiecewisePolynomialContinuousScalarSpace(context, grid, d)
```

or

```
createPiecewisePolynomialDiscontinuousScalarSpace(context, grid, d)
```

High-order basis functions are often used when results accurate to many significant digits are desired. Such results can only be obtained with sufficiently accurate quadrature rules. Use of a uniform high-order rule for all interelement distances is wasteful; to reduce time consumption, it is advisable to employ lower-order quadrature rules for distant elements. In the calculations whose results are presented in this section we varied the order q of the quadrature rule used to approximate double integrals on regular pairs of elements lying in the relative distance d in the following way:[6]

$$q = q_0 + \begin{cases} 10 & \text{if } d \leq 2.1 \\ 9 & \text{if } 2.1 \leq d < 4 \\ 6 & \text{if } 4 \leq d < 9 \\ 5 & \text{if } 9 \leq d < 13 \\ 4 & \text{otherwise}, \end{cases} \tag{33}$$

where q_0 is the default quadrature order ensuring exact integration of products of any test and trial functions defined on the elements. This rule was constructed so as to give relative integration error of less than ca. $1\mathrm{E}{-9}$ for the single-layer boundary potential operator associated with the Laplace equation discretised with constant test and trial functions defined on elements having the shape of isosceles triangles (but

[6]The *relative distance* of two elements is defined as the distance between their centres divided by their longest edge.

arbitrary relative orientation) – at least for elements lying in the relative distance $d > \sqrt{3}$ from each other. The singular quadrature order was increased by 6 with respect to the default value and the order of single-element quadrature, by 4. This choice of quadrature orders was done with the following code:

```
accuracyOptions = createAccuracyOptionsEx()
accuracyOptions.setDoubleRegular(
    [2.1, 4., 9., 13.], [10, 9, 6, 5, 4], True)
accuracyOptions.setDoubleSingular(6, True)
accuracyOptions.setSingleRegular(4, True)
quadStrategy = createNumericalQuadratureStrategy(
    "float64", "float64", accuracyOptions)
```

Table V presents the results obtained by solving the Laplace equation outside a unit cube with Dirichlet boundary conditions derived from the exact solution

$$u_{\text{exact}}(\mathbf{x}) = \frac{1}{4\pi|\mathbf{x}-\mathbf{x}_0|} \quad \text{with} \quad \mathbf{x}_0 = (0.4, 0.4, 0.4). \tag{34}$$

The (known) Dirichlet data were expanded in the space of globally continuous, piecewise polynomial functions of orders 1, 2 and 3, whereas the (unknown) Neumann data were expanded in the space of piecewise polynomial functions of orders 0, 1 and 2 without imposition of global continuity. The ACA tolerance ϵ was set to 1E$-$7 and the GMRES tolerance to 1E$-$10. The computational meshes were obtained by refining repeatedly the simplest 12-element cubic triangular mesh. The complete script used in the calculations, `interior_laplace_dirichlet_high_order.py`, can be found in the Supplementary Material. The table indicates that as the mesh is refined, the convergence rate of the $H^{-\frac{1}{2}}$ error approaches the theoretically predicted value of of $O(h^{p+\frac{3}{2}})$, where p is the order of the polynomials used to expand the Neumann data [Sauter and Schwab 2011, p. 201]. The convergence rate of the L^2 error tends to $O(h^{p+1})$. For the most dense mesh and $p = 2$, however, it would be necessary to increase further the singular quadrature rule order in order to get acceptably converged results. When comparing the calculation times it must be borne in mind that for lower-order bases it would be sufficient to use much less accurate quadrature rules, which would considerably shorten the computations.

Opposite-order and mass-matrix preconditioning. An effective way to precondition the Laplace single-layer potential boundary operator is via operator preconditioning [Hiptmair 2006]. We consider again the Dirichlet problem (21) to find the Neumann data t from

$$Vt = \left(\frac{1}{2}I + K\right) v, \tag{35}$$

where v is the given Dirichlet boundary data. The operator V is a pseudodifferential operator of order -1. The idea is now to multiply the above equation with a pseudodifferential operator of order 1, such that the product is a bounded operator with bounded inverse. A suitable operator is the hypersingular operator D. We obtain the new problem

$$DVt = D\left(\frac{1}{2}I + K\right) v. \tag{36}$$

Here, the operator DV maps functions from $H^{-\frac{1}{2}}(\Gamma)$ into $H^{-\frac{1}{2}}(\Gamma)$. To implement this product operator we could be tempted to proceed as previously, namely by defining the spaces as

Table V. Benchmarks of the solution of the Laplace equation with Dirichlet boundary conditions outside a unit cube for polynomial basis functions of varying order.
Abbreviations: p = polynomial order of basis functions used to expand the unknown Neumann data; #Elem. = Number of elements; #Unkn. = Number of unknowns; op. = operator; M. = Memory; t = Time; #It. = Number of iterations; eoc = experimental order of convergence. The memory use of ACA-compressed operators is listed in megabytes and in percent of the memory use of equivalent dense operators.

		DLP op.		SLP op.		Solver		Relative	Relative
#Elem.	#Unkn.	M. (MB / %)	t (s)	M. (MB / %)	t (s)	#It.	t (s)	L^2 error / eoc	$H^{1/2}$ error / eoc
				$p = 0$					
48	48	0.0 / 100	0.2	0.0 / 101	0.0	12	0.0	3.44E−1	9.47E−2
192	192	0.1 / 100	0.6	0.3 / 100	0.1	25	0.0	1.86E−1 / 1.84	3.08E−2 / 3.07
768	768	2.3 / 100	2.3	4.3 / 96	0.5	34	0.1	8.66E−2 / 2.15	9.37E−3 / 3.29
3072	3072	20.6 / 57	10.2	37.6 / 52	2.5	44	0.3	4.19E−2 / 2.07	3.16E−3 / 2.97
12288	12288	125.9 / 22	37.4	222.2 / 19	10.6	56	1.2	2.07E−2 / 2.02	1.10E−3 / 2.88
				$p = 1$					
48	144	0.1 / 100	0.3	0.2 / 100	0.1	33	0.0	1.10E−1	2.02E−2
192	576	1.7 / 100	1.1	2.5 / 100	0.6	53	0.0	3.66E−2 / 3.01	4.81E−3 / 4.21
768	2304	16.4 / 61	4.3	21.7 / 53	2.5	68	0.2	1.09E−2 / 3.35	1.10E−3 / 4.37
3072	9216	93.8 / 22	17.2	134.7 / 21	11.3	85	1.2	3.11E−3 / 3.51	2.30E−4 / 4.79
12288	36864	440.2 / 6	64.2	720.9 / 7	47.5	105	7.8	8.21E−4 / 3.79	4.34E−5 / 5.29
				$p = 2$					
48	288	0.5 / 100	0.8	0.6 / 100	0.2	70	0.0	3.79E−2	5.84E−3
192	1152	6.5 / 86	2.8	9.6 / 94	1.2	106	0.3	7.37E−3 / 5.14	8.55E−4 / 6.83
768	4608	44.9 / 37	10.7	57.0 / 35	5.0	139	0.7	9.16E−4 / 8.04	6.98E−5 / 12.25
3072	18432	222.1 / 11	38.6	318.0 / 12	21.2	174	5.1	1.14E−4 / 8.04	5.98E−6 / 11.67
12288	73728	997.2 / 3	148.5	1617.5 / 4	86.3	216	35.3	2.01E−5 / 5.66	8.69E−7 / 6.88

```
pconsts = createPiecewiseConstantScalarSpace(context, grid)
plins = createPiecewiseLinearContinuousScalarSpace(context, grid)
```

and then the operators by

```
slpOp = createLaplace3dSingleLayerBoundaryOperator(
    context, pconsts, plins, pconsts)
hypOp = createLaplace3dDoubleLayerBoundaryOperator(
    context, plins, plins, pconsts)

lhsOp = hypOp * slpOp
```

BEM++ will automatically map the range space of the operator `slpOp` to the domain space of the operator `hypOp`. However, this pairing between the space of piecewise constant functions $S_h^{(0)}(\Gamma)$ and the space of continuous piecewise linear functions $S_h^{(1)}$ is not stable [Hiptmair 2006; Steinbach 2002]. In order to achieve a stable pairing of spaces we need to define the single-layer operator using piecewise constant functions on the dual grid. These are implemented as a separate space class in BEM++. Hence, we only need to redefine `pconsts` as

```
pconsts = createPiecewiseConstantDualGridScalarSpace(context, grid)
```

The right-hand side of (36) is written in Python as

```
rhs =  hypOp * (0.5 * idOp + dlpOp) * dirichletTrace
```

The operators `idOp` and `dlpOp` are defined as in Section 4.1, but with the piecewise constant functions now defined on the dual grid. We could now solve in the same way as previously presented using the `DefaultIterativeSolver` class. However, in the fol-

Table VI. Comparison of mass-matrix and opposite-order preconditioning for a sphere mesh with 5120 elements.

Preconditioner	#It.	t (s)	Rel. L^2 error	Rel. $H^{-\frac{1}{2}}$ error
None	44	3.0	9.7E−3	1.1E−3
Mass matrix	19	24.8	9.8E−3	1.6E−3
Opposite order	13	28.0	9.8E−3	1.7E−3
Opposite order + mass matrix	6	27.5	9.8E−3	1.6E−3

lowing we demonstrate how to further accelerate convergence by switching on mass matrix preconditioning. The operator `slpOp` maps from the space $S^{(0)}_{h,\mathrm{d}}$ of piecewise constant functions on the dual grid into the space $[S^{(0)}_{h,\mathrm{d}}]'$. For the convergence of iterative solvers it is often of advantage to map back from the dual space into the original space. Hence, if we have a Galerkin discretized system $Ax = b$, $A : X_h \to Y_h'$, we wish to multiply both sides of the equation with the (pseudo)inverse $M^\dagger$ of a mass matrix M which maps Y_h into Y_h'. The new left-hand side operator $M^\dagger A$ now maps from X_h into Y_h (see also Kirby [2010]). In order to enable automatic mass-matrix preconditioning in BEM++ we can initialize the iterative solver as follows.

```
solver = createDefaultIterativeSolver(slpOp,"test_convergence_in_range")
solver.initializeSolver(defaultGmresParameterList(1e-8))
```

In Table VI we compare the mass-matrix and opposite-order preconditioners for a sphere mesh with 5120 elements. As ACA tolerance we have chosen a value of 1E−5. Furthermore, we have enabled the option

```
acaOps.mode = 'local_assembly'
```

Details and performance measures for the various assembly modes of BEM++ will be reported elsewhere. Otherwise, the parameters are the same as in Table III. The first row shows the case of no preconditioning. The errors have been computed on the primal grid by reprojecting the grid functions from the space of piecewise constant functions on the dual grid to the space of piecewise constant functions on the primal grid. The timing results include the assembly and the solve phase.

The table indicates that calculations with mass-matrix or opposite-order preconditioners are significantly slower than without preconditioning. The reason is that we need to define the space of piecewise constant functions on the dual grid, which is realized by a barycentric refinement of the original mesh, leading to about six times as many elements. This is an unavoidable cost if we need to work with stable dual pairings. Hence, opposite order preconditioning is more useful for fast multipole methods, where the setup time is small, but each matrix-vector product is costly, or in situations where we have no convergence in reasonable time at all without preconditioning.

Here we have shown opposite-order preconditioning for Laplace. But the same ideas also carry over to Helmholtz and are supported by BEM++. Dual-grid-based preconditioners for Maxwell [Andriulli et al. 2008] are in development and will be added in a later release.

4.2. Acoustic wave scattering

Mathematical background. Let Ω with surface Γ be a bounded acoustical obstacle made of material with density ρ_{int} and speed of sound c_{int} embedded in an infinite homogeneous medium with density ρ_{ext} and speed of sound c_{ext}. The propagation of

time-harmonic acoustic waves in this system is described by the equations

$$\Delta p(\mathbf{x}) + k_{\text{int}}^2 p(\mathbf{x}) = 0 \qquad \text{for} \quad \mathbf{x} \in \Omega, \tag{37a}$$

$$\Delta p(\mathbf{x}) + k_{\text{ext}}^2 p(\mathbf{x}) = 0 \qquad \text{for} \quad \mathbf{x} \in \mathbb{R}^3 \setminus \Omega, \tag{37b}$$

$$\gamma_0^{\text{int}} p(\mathbf{x}) = \gamma_0^{\text{ext}} p(\mathbf{x}) \qquad \text{for} \quad \mathbf{x} \in \Gamma, \tag{37c}$$

$$\rho_{\text{int}}^{-1} \gamma_1^{\text{int}} p(\mathbf{x}) = \rho_{\text{ext}}^{-1} \gamma_1^{\text{ext}} p(\mathbf{x}) \qquad \text{for} \quad \mathbf{x} \in \Gamma, \tag{37d}$$

where the wave numbers k_{int} and k_{ext} are defined as ω/c_{int} and ω/c_{ext}, respectively, with ω being the angular frequency, and $p(\mathbf{x})$ denotes the pressure perturbation at point $\mathbf{x}$. We want to calculate the pressure generated by the incident wave $p_{\text{inc}}(\mathbf{x})$ fulfilling the exterior Helmholtz equation (37b) in the whole three-dimensional space. Thus, we decompose the total field in the exterior medium into a sum of the incident and scattered field,

$$p(\mathbf{x}) = p_{\text{inc}}(\mathbf{x}) + p_{\text{sc}}(\mathbf{x}) \quad \text{for} \quad \mathbf{x} \in \mathbb{R}^3 \setminus \Omega, \tag{38}$$

with the scattered field fulfilling Sommerfeld's radiation conditions at infinity,

$$\lim_{|\mathbf{x}| \to \infty} |\mathbf{x}| \left[\frac{\partial}{\partial |\mathbf{x}|} p_{\text{sc}}(\mathbf{x}) - \mathrm{i} k_{\text{ext}} p_{\text{sc}}(\mathbf{x}) \right] = 0. \tag{39}$$

There exist several integral-equation formulations of the scattering problem presented above. One that will be suitable for our presentation was given by Kleinman and Martin [1988] and reads

$$\begin{bmatrix} D_{\text{ext}} + \alpha^{-1} D_{\text{int}} & T_{\text{ext}} + T_{\text{int}} \\ K_{\text{ext}} + K_{\text{int}} & -V_{\text{ext}} - \alpha V_{\text{int}} \end{bmatrix} \begin{bmatrix} \gamma_0^{\text{ext}} p \\ \gamma_1^{\text{ext}} p \end{bmatrix} = \begin{bmatrix} \gamma_1^{\text{ext}} p_{\text{inc}} \\ -\gamma_0^{\text{ext}} p_{\text{inc}} \end{bmatrix}, \tag{40}$$

where $\alpha := \rho_{\text{int}}/\rho_{\text{ext}}$ and V, K, T and D are the single-layer potential, double-layer potential, adjoint double-layer potential and hypersingular boundary operators for the interior or exterior Helmholtz equation (depending on the subscript). This formulation has the virtue of being free from the irregular frequency problem [Kleinman and Martin 1988].

The overall structure of the code required to solve this system of equations is similar to that presented in the previous section, so we will only discuss newly introduced features. The full code is available in `scattering.py` script included in the Supplementary Material.

Creation of operators. The first difference with respect to the code used to solve the Laplace equation is that the operators we are dealing with are now complex-valued. This needs to be indicated in the call to `createNumericalQuadratureStrategy()` by setting its second argument to `"complex128"`:

```
quadStrategy = createNumericalQuadratureStrategy(
    "float64", "complex128", accuracyOptions)
```

The construction of the elementary operators proceeds very similarly to the Laplace case, except that it becomes necessary to specify the wave number:

```
rhoExt = 1.; rhoInt = 2.; kExt = 4.; kInt = kExt / 5.

slpOpInt = createHelmholtz3dSingleLayerBoundaryOperator(
    context, pconsts, plins, pconsts, kInt, "SLP_int")
slpOpExt = createHelmholtz3dSingleLayerBoundaryOperator(
    context, pconsts, plins, pconsts, kExt, "SLP_ext")

dlpOpInt = createHelmholtz3dDoubleLayerBoundaryOperator(
```

```
    context, plins, plins, pconsts, kInt, "DLP_int")
dlpOpExt = createHelmholtz3dDoubleLayerBoundaryOperator(
    context, plins, plins, pconsts, kExt, "DLP_ext")

hypOpInt = createHelmholtz3dHypersingularBoundaryOperator(
    context, plins, pconsts, plins, kInt, "Hyp_int")
hypOpExt = createHelmholtz3dHypersingularBoundaryOperator(
    context, plins, pconsts, plins, kExt, "Hyp_ext")
```

As before, we use the space $S_h^1(\Gamma)$ of piecewise linears to expand $\gamma_0^{\text{ext}}p$ and the space $S_h^0(\Gamma)$ of piecewise constants to expand $\gamma_1^{\text{ext}}p$. These two spaces will also used to test the first and second equation, respectively.

The composite operators to be placed in individual blocks are defined with

```
alpha = rhoInt / rhoExt
lhsOp00 = hypOpExt + (1./alpha) * hypOpInt
lhsOp10 = dlpOpExt + dlpOpInt
lhsOp01 = adjoint(lhsOp10)
lhsOp11 = -slpOpExt - alpha * slpOpInt
```

Note that it is not necessary to create elementary adjoint double-layer potential boundary operators: the operator $T_{\text{ext}} + T_{\text{int}}$ can be constructed by passing the object representing the sum $K_{\text{ext}} + K_{\text{int}}$ to the `adjoint()` function. The resulting adjoint operator will reuse the discrete weak forms of the double-layer potential boundary operators, as is the case for other composite operators, such as sums.

Since we are dealing with a system of two integral equations, we need to construct a `BlockedBoundaryOperator` object, as indicated in section 3.3. This is very easy in Python—the individual blocks are passed simply as members of a nested list:

```
lhsOp = createBlockedBoundaryOperator(
    context, [[lhsOp00, lhsOp01], [lhsOp10, lhsOp11]])
```

Incident field. We choose the incident field to be a plane wave propagating in the x direction. The grid functions representing its Dirichlet and Neumann traces on Γ are defined in the familiar way:

```
def uIncDirichletTrace(point):
    x, y, z = point
    return np.exp(1j * kExt * x)
uInc = createGridFunction(context, plins, pconsts, uIncDirichletTrace)

def uIncNeumannTrace(point, normal):
    x, y, z = point
    nx, ny, nz = normal
    return 1j * kExt * nx * np.exp(1j * kExt * x)
uIncDeriv = createGridFunction(context, pconsts, plins, uIncNeumannTrace,
                               surfaceNormalDependent=True)
```

Solution and preconditioning. The equations can now be solved by constructing the solver in the way described in section 4.1—which we do not repeat here—and passing the list of grid functions occupying the individual blocks of the right-hand side to the `solve()` method:

```
solution = solver.solve([uIncDeriv, -uInc])
```

However, the convergence of the GMRES solver is slow: already for a 1280-element spherical grid 270 iterations are required to reduce the residual norm below 1E$-$8,

and the number of iterations grows rapidly with the size of the grid. We will therefore apply a preconditioner to speed up the calculations. Owing to the structure of eq. (40), a natural preconditioner is the approximate inverse of

$$\begin{bmatrix} \mathsf{D}_{\mathrm{ext}} + \alpha^{-1}\mathsf{D}_{\mathrm{int}} & 0 \\ 0 & -\mathsf{V}_{\mathrm{ext}} - \alpha\mathsf{V}_{\mathrm{int}} \end{bmatrix}, \tag{41}$$

where the sans-serif symbols denote discrete weak forms of the operators labelled with the corresponding serif letters.

Remark 4.1. We note that the block diagonal preconditioner given in (41) is not stable with respect to all wavenumbers. We have nevertheless chosen to present this preconditioner as it is very simple to apply in BEM++, gives good performance for low to moderate wavenumbers unless very close to a resonance, and demonstrates the effectiveness of purely algebraic preconditioning based on $\mathcal{H}$-matrix LU decompositions.

Approximate inverses of $\mathcal{H}$-matrices can be obtained readily by means of the approximate $\mathcal{H}$-matrix LU decomposition algorithm [Bebendorf 2008, pp. 180–183]. To construct the above preconditioner, we can use the following code:

```
precTol = 1e-2
invLhsOp00 = acaOperatorApproximateLuInverse(
    lhsOp00.weakForm().asDiscreteAcaBoundaryOperator(), precTol)
invLhsOp11 = acaOperatorApproximateLuInverse(
    lhsOp11.weakForm().asDiscreteAcaBoundaryOperator(), precTol)
prec = discreteBlockDiagonalPreconditioner([invLhsOp00, invLhsOp11])
```

This `acaOperatorApproximateLuInverse()` function takes two arguments: a `DiscreteAcaBoundaryOperator` object, i.e. a discrete operator stored as an $\mathcal{H}$-matrix, and a number δ controlling the approximation accuracy. It returns a new discrete operator storing the approximate LU decomposition of the original operator and acting as its approximate inverse. To obtain the required `DiscreteAcaBoundaryOperators`, we proceed in two steps. First, we call `weakForm()` to retrieve references to the discrete weak forms of the boundary operators that we want to invert. These operators are in fact superpositions of pairs of elementary boundary operators, and hence, as mentioned in section 4.1, their weak forms are little more than thin wrappers over pointers to the weak forms of the individual operands. Therefore we need to call the `asDiscreteAcaBoundaryOperator()` method, which uses the $\mathcal{H}$-matrix arithmetics to "compress" a `DiscreteBoundaryOperator` to a single $\mathcal{H}$-matrix and returns the resulting `DiscreteAcaBoundaryOperator` object. In the above snippet, we call the method without any arguments; however, it accepts two optional parameters than can be used to control the accuracy of arithmetic operations used to generate the resulting $\mathcal{H}$-matrix ($\mathcal{H}$-matrix arithmetic being intrinsically non-exact).

Currently, `asDiscreteAcaBoundaryOperator()` is not yet supported for all discrete boundary operators; in particular, it cannot be used to convert to $\mathcal{H}$-matrices products of operators or operators stored in the dense form. However, the most common types of composite operators, such as linear superpositions or (even) blocked operators, can be transformed into single $\mathcal{H}$-matrices.

The `discreteBlockDiagonalPreconditioner()` function produces a `Preconditioner` object wrapping a block-diagonal operator, with the argument being a list of discrete operators to be placed in the diagonal blocks. This function is provided for convenience, block-diagonal operators being an important class of preconditioners. It is possible to make a preconditioner out of any discrete operator; it suffices to pass it to the `discreteOperatorToPreconditioner()` function.

The newly created preconditioner needs now to be made available to the linear solver. This can be done with the second (optional) parameter to the `initializeSolver()` method:

```
solver.initializeSolver(params, prec)
```

The application of this preconditioner reduces the number of iterations to 32 for the 1280-element grid, and this number stays roughly constant as the grid is refined (with the wave number kept constant).

Off-surface field evaluation. The two components of the solution, the exterior Dirichlet and Neumann traces of the total pressure field on Γ, can be retrieved with

```
uExt = solution.gridFunction(0)
uExtDeriv = solution.gridFunction(1)
```

The traces may be plotted in the same way as in the Laplace case, i.e. by calling `plotGridFunction()` from the `visualization` module. In many cases one is interested, however, not only in the values of the field on the computational surface, but also away from it. These can be calculated using the Green's representation formula, which reads

$$p(\mathbf{x}) = \begin{cases} p_{\mathrm{inc}}(\mathbf{x}) - (\mathcal{V}_{\mathrm{ext}}\gamma_1^{\mathrm{ext}} p_{\mathrm{sc}})(\mathbf{x}) + (\mathcal{K}_{\mathrm{ext}}\gamma_0^{\mathrm{ext}} p_{\mathrm{sc}})(\mathbf{x}) & \text{for} \quad \mathbf{x} \in \mathbb{R}^3 \setminus \Omega, \\ (\mathcal{V}_{\mathrm{ext}}\gamma_1^{\mathrm{int}} p)(\mathbf{x}) - (\mathcal{K}_{\mathrm{ext}}\gamma_0^{\mathrm{int}} p)(\mathbf{x}) & \text{for} \quad \mathbf{x} \in \Omega \setminus \Gamma, \end{cases} \tag{42}$$

where the calligraphic letters denote the single- and double-layer *potential operators* (cf. section 3.6) associated with the fundamental solution of the Helmholtz equation in the interior and exterior domain. Their representations in BEM++, instances of subclasses of `PotentialOperator`, are created as follows:

```
slPotInt = createHelmholtz3dSingleLayerPotentialOperator(context, kInt)
dlPotInt = createHelmholtz3dDoubleLayerPotentialOperator(context, kInt)
```

etc. To evaluate a potential $\mathcal{A}f$, where $\mathcal{A}$ is a potential operator and f a grid function, at points $\{\mathbf{x}_i\}_{i=1}^n$, one should call the `evaluateAtPoints()` method of `PotentialOperator`. It takes a `GridFunction` object representing the function f acted upon by the operator, a two-dimensional NumPy array whose columns should contain the coordinates of the points $\{\mathbf{x}_i\}_{i=1}^n$, and an `EvaluationOptions` object that controls some aspects of the evaluation procedure, such as the level of parallelisation. For instance, assuming that the array `points` contains the coordinates of some points lying inside Ω, the snippet

```
evalOptions = createEvaluationOptions()
vals = (  slPotInt.evaluateAtPoints(uIntDeriv, points, evalOptions)
        - dlPotInt.evaluateAtPoints(uInt, points, evalOptions))
```

will produce an array of values of the pressure field at these points. The file `scattering.py` included in the Supplementary Material contains the complete code needed to calculate and plot the pressure field sampled at a regular grid of 201×201 points lying in the xy plane, together with the surface of the scatterer represented as a wireframe grid. The graph generated in this way is shown in figure 6.

Benchmarks. Table VII shows the results obtained by running the code presented in this section on a series of spherical grids, with the ratio of exterior wavelength $\lambda := 2\pi/k_{\mathrm{ext}}$ to element size h kept constant and approximately equal to 10. The ratio of k_{ext} to k_{int} was also fixed and equal to 5. In all calculations, the ACA tolerance ϵ was set to 10E$-$4 and the solver tolerance to 10E$-$8. The maximum rank of $\mathcal{H}$-matrix blocks to be considered low-rank during ACA was set to 1500 to reduce peak memory consumption in AHMED. Regular Galerkin integrals were approximated using quadra-

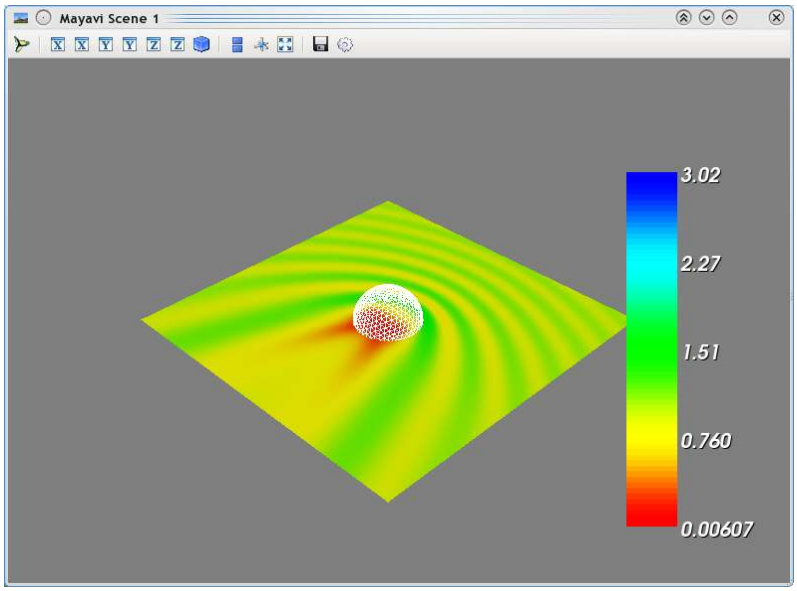

Fig. 6. Cross-section of the pressure distribution generated by plane wave impinging on a permeable sphere, as calculated with the code presented in section 4.2.

Table VII. $\mathcal{H}$-matrix assembly and solution times of the problem of plane-wave scattering on a permeable unit sphere. All times are given in seconds.

			Compr. rate (%)			No prec.		Prec. ($\delta = 0.1$)			Prec. ($\delta = 0.01$)		
k_{ext}	#Elem.	t_{ass}	V_{ext}	K_{ext}	D_{ext}	#It.	t_{sol}	t_{HLU}	#It.	t_{sol}	t_{HLU}	#It.	t_{sol}
1	80	0.4	100	100	100	47	0.03	0.0	17	0.0	0.0	15	0.0
2	320	0.9	92	93	100	100	0.19	0.0	23	0.0	0.1	19	0.0
4	1280	4.2	48	56	67	286	1.85	0.7	54	0.4	1.0	32	0.3
8	5120	24.9	20	22	31	786	35.2	3.4	50	2.7	7.1	40	2.4
16	20480	147.9	8	10	14	—	—	23.2	231	62.1	49.7	190	60.3
32	81920	904.8	3	4	6	—	—	169.5	1068	1667.3	445.6	250	495.6

ture rules of order greater by 2 than default[7] and singular integrals with quadrature rules of the default order of accuracy. The table shows the assembly times t_{ass} for the block operator system, compression rates achieved in the storage of the three exterior operators V_{ext}, K_{ext} and D_{ext}[8], the times t_{HLU} taken by the $\mathcal{H}$-LU decompositions, and the GMRES solver iteration counts #It. and solver times t_{sol}. The table compares the values obtained for no preconditioning with those for the block-diagonal preconditioner defined in eq. (41), for two values of the $\mathcal{H}$-LU decomposition accuracy δ. In table VIII we present the associated errors measured against an accurate solution obtained by series expansion.

Extraordinary acoustical transmission and screening. The laws of scalar acoustics apply essentially in gaseous and liquid media; the propagation of sound in solids is governed by the more complex theory of elasticity. However, the problem of scatter-

[7]This choice yields 3 quadrature points on elements supporting piecewise constant basis functions and 4 quadrature points on those supporting piecewise linear basis functions.

[8]The compression rate is defined as the ratio between the memory consumption of the $\mathcal{H}$-matrix and the equivalent dense matrix. The compression rates of the interior operators—not shown—are slightly better because $k_{\text{int}} < k_{\text{ext}}$.

Table VIII. Errors for the problem of plane-wave scattering on a permeable unit sphere.

		Dirichlet trace		Neumann trace	
k_{ext}	#Elem.	Rel. L^2 error	Rel. $H^{\frac{1}{2}}$ error	Rel. L^2 error	Rel. $H^{-\frac{1}{2}}$ error
21	80	2.70E−2	4.34E−2	3.05E−1	1.10E−1
12	320	1.81E−2	3.73E−2	1.86E−1	5.46E−2
14	1280	1.08E−2	3.16E−2	1.37E−1	3.54E−2
88	5120	8.11E−3	2.88E−2	1.17E−1	2.73E−2
616	20480	6.79E−3	2.75E−2	1.08E−1	1.96E−2
632	81920	6.73E−3	2.71E−2	1.05E−1	2.15E−2

ing of sound on solid objects can often be simplified by treating them as sound-hard obstacles, i.e. by imposing on their surface homogeneous Neumann boundary conditions instead of the more rigorous transmission conditions. This leads to the exterior Neumann problem for the Helmholtz equation. Its most popular stable (free from the irregular frequency problem) integral formulation is due to Burton and Miller [1971] and reads

$$\left(\frac{1}{2}I - K + \alpha D\right)\gamma_0^{\text{ext}} p = \gamma_0^{\text{ext}} p_{\text{inc}} + \alpha\gamma_1^{\text{ext}} p_{\text{inc}}, \tag{43}$$

where I is the identity operator, K and D the double-layer potential and hypersingular boundary operators for the exterior domain, and α is an imaginary coupling coefficient, usually chosen, after Kress [1985], as $\mathrm{i}/k_{\text{ext}}$.

After small modifications, the code developed in this section can be used to handle this class of problems. As an illustraction, we will consider the application of BEM++ to the modelling of sound scattering by perforated metallic plates. In 1998, Ebbesen et al. reported that metallic screens pierced by an array of subwavelength holes became almost transparent to light at certain frequencies—letting through much more energy than one would expect judging on the air filling fraction. This effect, dubbed the extraordinary optical transmission, excited a lot of interest, and its physical mechanism, based on the interaction of long-range propagating surface waves and localised cavity modes, was intensely studied in the last decade [Liu and Lalanne 2008; García de Abajo 2007]. More recently, an analogous effect was observed for acoustic waves [Estrada et al. 2008], together with the phenomenon of extraordinary screening—very low transmittance at specific frequencies. Here we present results of calculations performed for one of the geometries studied experimentally and theoretically by Estrada et al. [2008].

We consider a plane wave of wavelength λ propagating along the $+z$ direction and impinging perpendicularly on a finite aluminum plate of thickness $t = 3\,\text{mm}$, perforated with an array of $m \times n$ circular holes with diameter $d = 3\,\text{mm}$, arranged on a square lattice with period $a = 5\,\text{mm}$. The plate is treated as a sound-hard obstacle. Its lateral dimensions are such that the centres of the outermost holes lie at the distance $a/2$ from the plate's edges.

Figure 7 shows the pressure distributions generated by plane waves of wavelengths 5.4 mm (top) and 8 mm (bottom). In each case, the field on the surface of the plate is juxtaposed with the map of the field in the $y = 0$ plane. The figure obtained for the shorter wavelength illustrates the effect of extraordinary screening—the amplitude of the transmitted wave is low and strong interference fringes behind the plate testify its high reflectivity—while at the longer wavelength most of the incoming energy is transmitted through the obstacle. These results are in agreement with the findings of Estrada et al. [2008].

The simulations were done by discretising eq. (43) on a 31,966-element mesh representing the plate with 9×7 holes. The incident field and the sought total field were

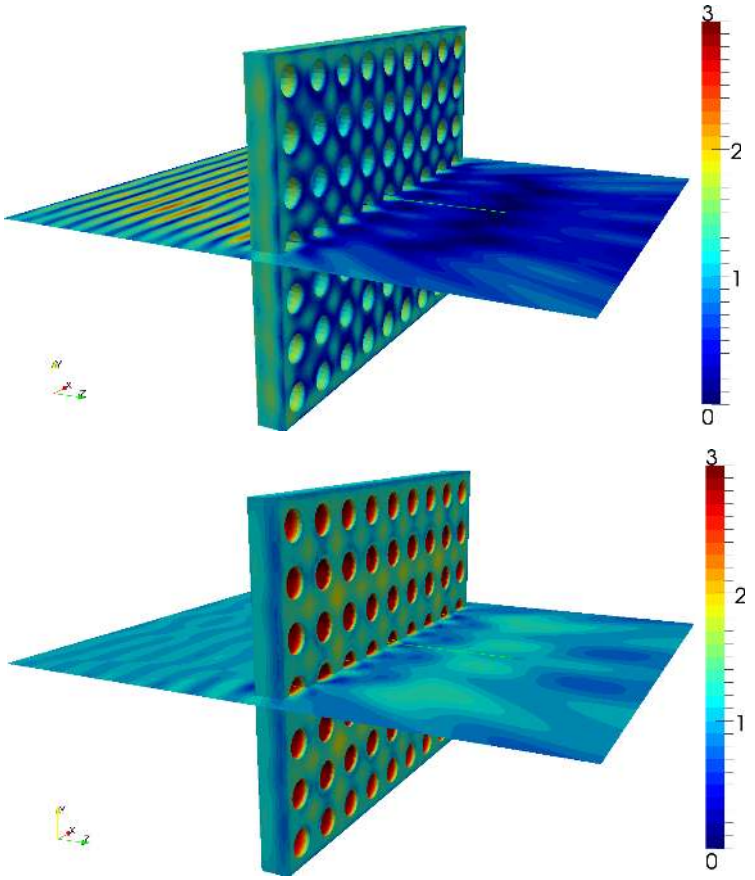

Fig. 7. Magnitude of the acoustic pressure generated by a plane wave with wavelength 5.4 mm (top) and 8 mm (bottom) impinging on a perforated sound-hard plate with dimensions specified in the text.

expanded in the space of continuous piecewise linear functions. This led to an algebraic system of equations with 15,859 unknowns. The quadrature orders were chosen as in the section "Benchmarks" above. The GMRES tolerance was set to 1E$-$8. At an ACA tolerance of 1E$-$5, the discrete weak forms of K and D were compressed to 20% and 15% of the corresponding dense-matrix memory usage, respectively, at the shorter wavelength of $\lambda = 5.4\,\mathrm{mm}$; at $\lambda = 8\,\mathrm{mm}$ these figures were 18% and 14%. The total solution time (excluding off-surface evaluation of the calculated field) at $\lambda = 5.4\,\mathrm{mm}$ was 403 s and at $\lambda = 8\,\mathrm{mm}$, 361 s. The script used in the simulations, `holey_plate.py`, together with the mesh `holey_plate.msh`, is included in the Supplementary Material.

4.3. Laplace equation with mixed boundary conditions

We will discuss now how BEM++ can be used to solve problems with mixed boundary conditions. Consider the Laplace equation imposed inside a domain Ω with boundary Γ, with Dirichlet boundary conditions prescribed on a subset of the boundary, $\Gamma_\mathrm{D} \subset \Gamma$, and

Neumann boundary conditions prescribed on the rest of the boundary, $\Gamma_\mathrm{N} = \Gamma \setminus \Gamma_\mathrm{D}$:

$$(\Delta u)(\mathbf{x}) = 0 \quad \text{for} \quad \mathbf{x} \in \Omega, \tag{44a}$$

$$\gamma_0^{\mathrm{ext}} u(\mathbf{x}) = g_\mathrm{D}(\mathbf{x}) \quad \text{for} \quad \mathbf{x} \in \Gamma_\mathrm{D}, \tag{44b}$$

$$\gamma_1^{\mathrm{ext}} u(\mathbf{x}) = g_\mathrm{N}(\mathbf{x}) \quad \text{for} \quad \mathbf{x} \in \Gamma_\mathrm{N}. \tag{44c}$$

A boundary integral formulation of this problem reads [Steinbach 2008, p. 179]

$$(V\tilde{t})(\mathbf{x}) - (K\tilde{u})(\mathbf{x}) = \left(\frac{1}{2}I + K\right)\tilde{g}_\mathrm{D}(\mathbf{x}) - (V\tilde{g}_\mathrm{N})(\mathbf{x}) \quad \text{for} \quad \mathbf{x} \in \Gamma_\mathrm{D}, \tag{45a}$$

$$(D\tilde{u})(\mathbf{x}) - (K'\tilde{t})(\mathbf{x}) = \left(\frac{1}{2}I - K'\right)\tilde{g}_\mathrm{N}(\mathbf{x}) - (D\tilde{g}_\mathrm{D})(\mathbf{x}) \quad \text{for} \quad \mathbf{x} \in \Gamma_\mathrm{N}, \tag{45b}$$

where $\tilde{g}_\mathrm{D} \in H^{\frac{1}{2}}(\Gamma)$ and $\tilde{g}_\mathrm{N} \in H^{-\frac{1}{2}}(\Gamma)$ are suitable extensions of the prescribed boundary data $g_\mathrm{D} \in H^{\frac{1}{2}}(\Gamma_\mathrm{D})$ and $g_\mathrm{N} \in H^{-\frac{1}{2}}(\Gamma_\mathrm{N})$ to the whole surface of the domain, whereas the new unknowns $\tilde{u}$ and $\tilde{t}$ are defined as

$$\tilde{u} := \gamma_0^{\mathrm{int}} u - \tilde{g}_\mathrm{D}, \qquad \tilde{t} := \gamma_1^{\mathrm{int}} u - \tilde{g}_\mathrm{N}. \tag{46}$$

These functions ara nonzero only on Γ_N and Γ_D, respectively.

The special difficulty of handling problems with mixed boundary conditions lies in the need of introducing function spaces defined only on parts (segments) of grids. Since version 2.0, BEM++ offers this possibility: each function constructing a `Space` object (e.g. `createPiecewiseConstantScalarSpace()`) takes an optional argument of type `GridSegment`, which describes the part of a grid to which the support of basis functions of the newly constructed `Space` should be restricted. A `GridSegment` is effectively a list of the indices of the elements, edges and vertices belonging to the chosen part of the grid.

Suppose that a grid has been imported from the Gmsh file `grid.msh` by calling

```
grid = createGridFactory().importGmshGrid("triangular", "grid.msh")
```

A Gmsh file can contain indices of so-called *physical entities* associated with elements. (In the BEM++ documentation we prefer to use the term *domain indices* to avoid confusion with entities understood as elements of grids—faces, edges and vertices.) If the elements belonging to Γ_D have been assigned the physical entity index (domain index) 1 and those belonging to Γ_N the index 2, then `GridSegment` objects representing the two parts of the grid can be constructed as follows:

```
segmentD = GridSegment.closedDomain(grid, 1)
segmentN = segmentD.complement()
```

The `closedDomain(`*`grid`*`,` *`index`*`)` static function creates a segment consisting of the elements of the grid *`grid`* whose domain index is equal to *`index`*, together with all their edges and vertices. The four standard set operations (union, difference, intersection and complement) are defined for `GridSegment` objects, which makes it possible to construct more complicated segments. Above, `segmentN` is set to the complement of `segmentD`; thus, it will contain all the elements belonging to Γ_N and the edges and vertices lying inside Γ_N, excluding the edges and vertices lying on the boundary separating Γ_N from Γ_D.

With the `segmentD` and `segmentN` objects available and after creating a `Context` object, which is done as in the previous section, the spaces of piecewise constants and (continuous) piecewise linears on Γ_D and Γ_N can be constructed as follows:

```
pwiseConstantsD = createPiecewiseConstantScalarSpace(
    context, grid, segmentD)
```

```
pwiseConstantsN = createPiecewiseConstantScalarSpace(
    context, grid, segmentN)
pwiseLinearsD = createPiecewiseLinearContinuousScalarSpace(
    context, grid, segmentD)
pwiseLinearsN = createPiecewiseLinearContinuousScalarSpace(
    context, grid, segmentN)
```

The way a `Space` constructor uses the lists of indices from a `GridSegment` object is space-dependent. For example, `pwiseConstantsD` will include only the piecewise constant basis functions defined on the elements belonging to `segmentD`, whereas `pwiseLinearsD` will include only the continuous, piecewise linear ("hat") basis functions associated with the vertices belonging to `segmentD`. Note that, as a result, the support of some basis functions of `pwiseConstantsD`, namely those associated with the vertices lying on the boundary of Γ_{D}, will extend outside Γ_{D}. It is possible to clip all basis functions to the elements belonging to Γ_{D} by setting the optional parameter `strictlyOnSegment` to `True`:

```
clippedPwiseLinearsD = createPiecewiseLinearContinuousScalarSpace(
    context, grid, segmentD, strictlyOnSegment=True)
```

Note that functions belonging to a space constructed in this way will still be continuous on the specified grid segment, but not on the whole grid.

`BoundaryOperator` objects representing the operators V, K etc. from eq. (45) are constructed as usual, with `pwiseConstantsD`, `pwiseLinearsD`, `pwiseConstantsN` and `pwiseLinearsN` passed as the domain, range and dual-to-range spaces, as appropriate. The construction of the `GridFunction` objects representing $\tilde{g}_{\mathrm{D}}$ and $\tilde{g}_{\mathrm{N}}$ requires some extra care. Suppose that the Python functions `evalDirichletTrace()` and `evalNeumannTrace()` return the values of g_{D} and g_{N} on Γ_{D} and Γ_{N}. To find the best approximation in `pwiseLinearsD` of an extension $\tilde{g}_{\mathrm{D}} \in H^{\frac{1}{2}}(\Gamma)$ of $g_{\mathrm{D}} \in H^{\frac{1}{2}}(\Gamma_{\mathrm{D}})$, we cannot simply write

```
dirichletTraceD = createGridFunction(
    context, pwiseLinearsD, pwiseLinearsD, evalDirichletTrace)
```

Instead, we should choose as the dual space (the third argument) a space of functions defined strictly on Γ_{D}:

```
dirichletTraceD = createGridFunction(
    context, pwiseLinearsD, clippedPwiseLinearsD, evalDirichletTrace)
```

In this way, the coefficients of `dirichletTraceD` in the basis of `pwiseLinearsD` are determined solely from the values of $\tilde{g}_{\mathrm{D}}$ on Γ_{D}; the form of the basis functions of `pwiseLinearsD` will then automatically ensure the global continuity of `dirichletTraceD` and its linear decay to zero within one layer of elements adjacent to Γ_{D}.

Since the basis functions of `pwiseConstantsN` do not extend outside Γ_{N}, the best approximation of $\tilde{g}_{\mathrm{N}}$ in `pwiseConstantsN` can be simply obtained with

```
neumannTraceN = createGridFunction(
    context, pwiseConstantsN, pwiseConstantsN, evalNeumannTrace)
```

The construction of a `BlockedBoundaryOperator` and a vector of `GridFunctions` representing the left- and right-hand side of eq. (45), as well as the solution of the resulting system, is done as in section 4.2. This yields the `GridFunctions` `dirichletTraceN` and `neumannTraceD` approximating $\tilde{u} : \Gamma_{\mathrm{N}} \to \mathbb{R}$ and $\tilde{t} : \Gamma_{\mathrm{D}} \to \mathbb{R}$. Typically, one is interested

Table IX. Error of the solution of the Laplace equation with mixed boundary conditions on spherical meshes with varying number of elements. The total number of unknowns is roughly equal to 75% of the number of elements.

	Dirichlet trace		Neumann trace	
#Elem.	Rel. L^2 error	Rel. $H^{\frac{1}{2}}$ error	Rel. L^2 error	Rel. $H^{-\frac{1}{2}}$ error
320	5.87E−4	1.26E−2	3.96E−2	9.36E−3
1280	1.41E−4	4.36E−3	1.95E−2	3.15E−3
5120	3.48E−5	1.53E−3	9.67E−3	1.09E−3
20480	8.65E−6	5.39E−4	4.82E−3	3.82E−4
81920	2.14E−6	1.90E−4	2.41E−3	1.35E−4

in $\gamma_0^{\text{int}} u$ and $\gamma_1^{\text{int}} u$ rather than $\tilde{u}$ and $\tilde{t}$. From eq. (46) $\gamma_0^{\text{int}} u = \tilde{u} + \tilde{g}_{\text{D}}$; however, it is not possible to write

```
dirichletTrace = dirichletTraceD + dirichletTraceN
```

since the two `GridFunctions` on the right-hand side are expanded in different function spaces. The remedy is to create identity operators mapping `pwiseLinearsD` and `pwiseLinearsN` into the space of continuous piecewise linears defined on the whole grid, `pwiseLinears`:

```
pwiseLinears = createPiecewiseLinearContinuousScalarSpace(
    context, grid)
scatterPwiseLinearsD = createIdentityOperator(
    context, pwiseLinearsD, pwiseLinears, pwiseLinears)
scatterPwiseLinearsN = createIdentityOperator(
    context, pwiseLinearsN, pwiseLinears, pwiseLinears)
```

and to use them to bring `dirichletTraceD` and `dirichletTraceN` into a common function space:

```
dirichletTrace = (scatterPwiseLinearsD * dirichletTraceD +
                  scatterPwiseLinearsN * dirichletTraceN)
```

The script `interior_laplace_mixed.py` included in the Supplementary Material is a complete implementation of the ideas outlined in this section. The script solves the Laplace equation inside the unit sphere with Dirichlet boundary conditions imposed on the part of its boundary with $x_2 > x_3/\sqrt{3}$ and Neumann boundary conditions imposed on the rest of the boundary; the functions $g_{\text{D}}(\mathbf{x})$ and $g_{\text{N}}(\mathbf{x})$ are derived from the exact solution (27). Table IX lists the relative $L^2(\Gamma)$- and $H^{\pm\frac{1}{2}}(\Gamma)$-errors of the Dirichlet and Neumann traces obtained using a selection of spherical grids. In all calculations, the ACA tolerance ϵ was set to 10E−7 and the solver tolerance to 10E−8. Regular integrals over pairs of elements were approximated using quadrature rules of order greater by 4 than default, integrals over single elements using quadrature rules of order greater by 2 than default, and singular integrals with quadrature rules of the default order of accuracy. The convergence rates of $\gamma_0^{\text{int}} u$ and $\gamma_1^{\text{int}} u$ match those observed and predicted for pure Dirichlet and Neumann problems discussed in section 4.1.

4.4. Maxwell equations

Exterior Dirichlet problem. In the first example, we will solve the Maxwell equations (12) in the exterior of a unit sphere Ω, on whose surface Γ we impose the Dirichlet boundary conditions derived from the exact solution

$$\mathbf{E}_{\text{exact}}(\mathbf{x}) = \mathbf{e}_\phi h_1^{(1)}(kr) \quad \text{for} \quad \mathbf{x} \in \mathbb{R}^3 \setminus \Omega. \tag{47}$$

Above, $h_1^{(1)}(\cdot)$ is the spherical Hankel function of the first kind and first order, (r, θ, ϕ) are the spherical coordinates anchored at the point $\mathbf{x}_0$ with Cartesian coordinates $(0.1, 0.1, 0.1)$, the symbol $\mathbf{e}_\phi$ denotes the unit vector parallel to $\mathrm{d}\mathbf{x}/\mathrm{d}\phi$ and k is the wave number.

An integral formulation of this problem is given by eq. (18a). To solve it with BEM++, we proceed similarly as in previous sections. We first create a space spanning the lowest-order Raviart-Thomas basis function defined on the grid's elements:

```
space = createRaviartThomas0VectorSpace(context, grid)
```

Afterwards, we construct the necessary operators:

```
slpOp = createMaxwell3dSingleLayerBoundaryOperator(
    context, space, space, space, k, "SLP")
dlpOp = createMaxwell3dSingleLayerBoundaryOperator(
    context, space, space, space, k, "SLP")
idOp = createMaxwell3dIdentityOperator(
    context, space, space, space, "Id")
```

Note that we used the `createMaxwell3dIdentityOperator()` function instead of `createIdentityOperator()`, as the latter function would create an operator with the weak form defined with regard to the standard sesquilinear inner product rather than the antisymmetric pseudo-inner product used in Buffa and Hiptmair's [2003] formalism.

We also need to create a `GridFunction` representing the Dirichlet data $\gamma_\mathrm{D}\mathbf{E}_\mathrm{exact} := \mathbf{E}_\mathrm{exact}|_\Gamma \times \mathbf{n}$. To this end, we first define a Python function returning a three-element array, whose elements will be interpreted by BEM++ as the components of the vector $\gamma_\mathrm{D}\mathbf{E}_\mathrm{exact}$ at a given point:

```
def evalDirichletTraceInc(point, normal):
    x, y, z = point - 0.1
    r = sqrt(x**2 + y**2 + z**2)
    kr = k * r
    h1kr = (-1j - kr) * exp(1j * kr) / (kr * kr)
    field = h1kr * [-y / r, x / r, 0.]
    return np.cross(field, normal)
```

Subsequently, we call `createGridFunction`, as usual:

```
dirichletTraceInc = createGridFunction(
    context, space, space, evalDirichletTraceInc,
    surfaceNormalDependent=True)
```

The rest of the program is fairly standard. The complete script, `exterior_maxwell_dirichlet.py`, can be found in the Supplementary Material.

Table X shows the results obtained by running the above script on a series of spherical grids, with the wavelength-to-element-size ratio λ/h kept constant and approximately equal to 10. Numerical quadrature of regular integrals was done using a quadrature rule of order greater by 2 than the library's default; that of singular integrals, using the default rule. The ACA tolerance ϵ was fixed to $1\mathrm{E}{-4}$. We found that increasing quadrature order or decreasing ACA tolerance had negligible effect on the accuracy of the solution. We used an approximate $\mathcal{H}$-LU decomposition of the operator S to precondition the GMRES iterative solver; the LU accuracy was set to 0.1 and the GMRES solver tolerance to $1\mathrm{E}{-8}$. The solution error initially decreases quickly with element size, and then becomes stable. The fast initial decay is an artifact caused

Table X. Benchmarks for the solution of the Dirichlet problem for Maxwell equations outside a unit sphere. The memory use of the operator $\mathbf{S}$ is listed in megabytes and in percent of the memory use of an equivalent dense operator. The memory and time consumption of $\mathbf{C}$ is very similar and has been omitted for brevity. The number of unknowns is roughly equal to 150% of the number of elements.

		$\mathbf{S}$		Preconditioner		Solver		Rel.
k	#Elem.	Mem. (MB / %)	t (s)	Mem. (MB)	t (s)	#It.	t (s)	L^2 error
1	80	0.2 / 100	0.1	0.2	0.0	7	0.0	4.4E−1
2	320	3.5 / 99	0.5	1.5	0.2	19	0.0	1.1E−1
4	1280	36.3 / 65	2.7	10.1	1.5	37	0.2	4.0E−2
8	5120	243.5 / 27	12.1	70.6	10.8	42	1.7	2.5E−1
16	20480	1547.7 / 11	62.1	543.7	72.0	180	32.9	2.1E−1
32	81920	10355.2 / 4	441.4	4547.9	682.9	255	325.1	2.0E−1

by the fact that the influence of the field singularity at $\mathbf{x}_0$ on the field's behaviour on Γ becomes weaker as the wavelength decreases.

Scattering by a screen. The second, more practical example concerns the calculation of the field radiated by a horn antenna, represented as an open (infinitesimally thin) perfectly conducting screen. The geometry of the object is shown in fig. 8(a). The excitation field $\mathbf{E}_{\text{inc}}$ is generated by a z-oriented electric dipole located on the symmetry axis of the feeding waveguide, 4.5 mm to the left from the input plane of the horn. In the simulation the waveguide is taken to be short-circuited on its left end. As the waveguide is single-moded and at the simulation frequency, corresponding to the wavelength 3 mm, its most slowly evanescent mode has decay length of only 0.56 mm, the excitation field is very close to that of the fundamental mode of the waveguide.

The problem can be reduced to solving the electric-field integral equation

$$\mathbf{S}_k \mathbf{U} = \gamma_{\text{D}} \mathbf{E}_{\text{inc}}, \tag{48}$$

where $\mathbf{U}$ is the jump of $\gamma_{\text{N,ext}}\mathbf{E}$ across the screen. This equation can be derived by subtracting eqs. (17a) and (18a) describing the interior and exterior Dirichlet problems with $\gamma_{\text{D,int}}\mathbf{E} = \gamma_{\text{D,ext}}\mathbf{E} = -\mathbf{E}_{\text{inc}}$ on a perfectly conducting screen of finite thickness d and taking the limit $d \downarrow 0$. Physical considerations [Bouwkamp 1950] additionally impose the *edge condition*: the edge-parallel component of $\mathbf{E}$ vanishes on all edges of the screen. Numerically, this condition is imposed by setting the coefficients of all Raviart-Thomas basis functions associated with the screen edges to zero. This is the default behaviour of BEM++; if necessary, it can be changed by setting the optional parameter `putDofsOnBoundaries` of `createRaviartThomas0VectorSpace()` to `True`.

The `horn_antenna.py` script solving the above problem can be found in the Supplementary Material. The antenna was discretised with 55,006 triangular elements with typical size of $h \approx \lambda/10$ ($\lambda/15$ on the feeding waveguide), which led to a linear system of size 82,393. To improve load balancing, the maximum $\mathcal{H}$-matrix block size was set to 10,000. All remaining calculation parameters were chosen as in the previous section. Assembly of the $\mathcal{H}$-matrix representation of the discretisation of $\mathbf{S}_k$ took 160 s; the compression rate was 4%. The $\mathcal{H}$-LU preconditioner was generated in 198 s. The GMRES solver converged in 105 iterations and 56 s. Figure 8(b) shows the magnitude of the field on the horizontal symmetry plane of the antenna ($z = 0$), and fig. 8(c) the radiation patterns (normalised to their maxima) on the planes horizontal and vertical symmetry planes ($z = 0$ and $x = 0$, respectively).

An example of solving an electromagnetic transmission problem with BEM++ is given in Betcke et al. [2013].

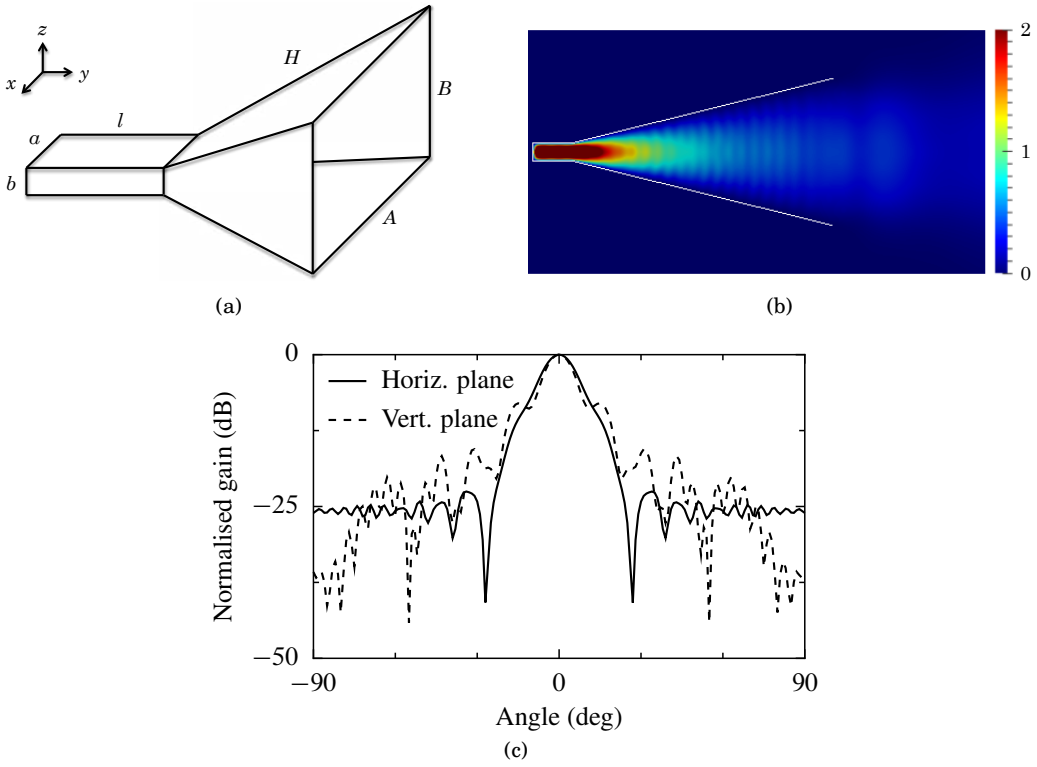

Fig. 8. (a) Geometry of the horn antenna (not to scale). (b) Magnitude of the electric field on the horizontal symmetry plane of the antenna. (c) Normalised radiation patterns generated by the antenna in horizontal and vertical symmetry planes.

5. FUTURE DEVELOPMENTS

The core library is now stable and supports a variety of problems based on Laplace, Helmholtz or Maxwell equations. Some smaller functionalities of the library have not been described here, such as a growing Python module generating meshes of primitive objects (for benchmarking) or the interface to PyTrilinos.

Work is ongoing on the implementation of additional preconditioners, such as algebraic multigrid preconditioners for Laplace-type problems [Of 2008] and Calderon preconditioners for the electric-field integral equation [Andriulli et al. 2008].

A big development focus at the moment is the integration of Fast Multipole Methods for the various kernels. Initial developments are available in an experimental branch and will be included in the main code in one of the upcoming major releases once they are sufficiently optimised.

A significant area of interest is also fast boundary element methods on modern many-core architectures. The current version of BEM++ uses Intel TBB [Intel 2012] to parallelise operations on shared-memory systems. The current implementation performs well on modern multicore systems with 12 to 16 CPU cores. However, parallelisation on true many-core architectures, such as NVIDIA Tesla or Intel MIC requires different strategies and will be a focus of development in the coming years.

ACKNOWLEDGMENTS

BEM++ has profited greatly from many external collaborators. In particular, we would like to thank the HyENA developers at TU Graz for making available their numerical integration routines for BEM++. We

would like to thank Mario Bebendorf from Uni Bonn for his support in integrating AHMED into BEM++, and Günther Of and Olaf Steinbach from TU Graz and Lars Kielhorn from ETH Zurich for many helpful discussions. The DUNE team has been very supportive in integrating the DUNE interface into BEM++, and we would in particular like to acknowledge the DUNE support given to us by Andreas Dedner from the University of Warwick and Oliver Sander from RWTH Aachen who developed the Foamgrid module for DUNE which forms the basis of our grid classes. Many early beta testers have helped the BEM++ development. In particular Peter Monk from the University of Delaware and Gerhard Unger from TU Graz gave significant constructive feedback while testing the early versions of the library. Finally, we would like to thank the anonymous referees for their constructive criticism that significantly improved this paper.

REFERENCES

ANDRIULLI, F., COOLS, K., BAGCI, H., OLYSLAGER, F., BUFFA, A., CHRISTIANSEN, S., AND MICHIELSSEN, E. 2008. A multiplicative Calderon preconditioner for the electric field integral equation. *IEEE Transactions on Antennas and Propagation 56,* 8, 2398–2412.

BARTLETT, R. A. 2007. Thyra linear operators and vectors. Tech. Rep. SAND2007-5984, Sandia National Laboratories, Albuquerque, NM.

BASTIAN, P., BLATT, M., DEDNER, A., ENGWER, C., KLÖFKORN, R., KORNHUBER, R., OHLBERGER, M., AND SANDER, O. 2008a. A generic grid interface for parallel and adaptive scientific computing. part II: Implementation and tests in DUNE. *Computing 82,* 2–3, 121–138.

BASTIAN, P., BLATT, M., DEDNER, A., ENGWER, C., KLÖFKORN, R., OHLBERGER, M., AND SANDER, O. 2008b. A generic grid interface for parallel and adaptive scientific computing. part I: Abstract framework. *Computing 82,* 2–3, 103–119.

BEAZLEY, D. 2003. Automated scientific software scripting with SWIG. *Future Generation Computer Systems 19,* 5, 599–609. Tools for Program Development and Analysis. Best papers from two Technical Sessions, at ICCS2001, San Francisco, CA, USA, and ICCS2002, Amsterdam, The Netherlands.

BEBENDORF, M. 2008. *Hierarchical Matrices: A Means to Efficiently Solve Elliptic Boundary Value Problems*. Lecture Notes in Computational Science and Engineering, 63. Springer, Berlin Heidelberg.

BEBENDORF, M. 2012. Another software library on hierarchical matrices for elliptic differential equations (AHMED). `http://bebendorf.ins.uni-bonn.de/AHMED.html`.

BETCKE, T., ARRIDGE, S., PHILLIPS, J., SCHWEIGER, M., AND ŚMIGAJ, W. 2013. Solution of electromagnetic problems with BEM++. In *Oberwolfach Report*. Vol. 03/2013.

BOUWKAMP, C. J. 1950. On Bethes theory of diffraction by small holes. *Philips Res. Rep. 5*, 321.

BUFFA, A. AND HIPTMAIR, R. 2003. Galerkin boundary element methods for electromagnetic scattering. In *Topics in computational wave propagation*. Springer, 83–124.

BURTON, A. J. AND MILLER, G. F. 1971. The application of integral equation methods to the numerical solution of some exterior boundary-value problems. *R. Soc. London Proc. Ser. A 323*, 201–210.

CHENG, H., CRUTCHFIELD, W. Y., GIMBUTAS, Z., GREENGARD, L. F., ETHRIDGE, J. F., HUANG, J., ROKHLIN, V., YARVIN, N., AND ZHAO, J. 2006. A wideband fast multipole method for the Helmholtz equation in three dimensions. *J. Comp. Phys. 216,* 1, 300–325.

CHERNOV, A. AND SCHWAB, C. 2012. Exponential convergence of Gauss–Jacobi quadratures for singular integrals over simplices in arbitrary dimension. *SIAM J. Numer. Anal. 50,* 3, 1433–1455.

CHERNOV, A., VON PETERSDORFF, T., AND SCHWAB, C. 2011. Exponential convergence of hp quadrature for integral operators with Gevrey kernels. *ESAIM: Mathematical Modelling and Numerical Analysis 45,* 3, 387–422.

COLTON, D. L. AND KRESS, R. 2013. *Inverse acoustic and electromagnetic scattering theory*. Springer.

DUNAVANT, D. A. 1985. High degree efficient symmetrical Gaussian quadrature rules for the triangle. *Int. J. Num. Meth. Engng 21,* 6, 1129–1148.

DUNE 2012. Distributed and Unified Numerics Environment (DUNE). `http://www.dune-project.org`.

EBBESEN, T. W., LEZEC, H. J., GHAEMI, H. F., THIO, T., AND WOLFF, P. A. 1998. Extraordinary optical transmission through sub-wavelength hole arrays. *Nature 391*, 667–669.

ESTRADA, H., CANDELAS, P., URIS, A., BELMAR, F., GARCÍA DE ABAJO, F. J., AND MESEGUER, F. 2008. Extraordinary sound screening in perforated plates. *Phys. Rev. Lett. 101,* 8, 084302.

GARCÍA DE ABAJO, F. J. 2007. Colloquium: Light scattering by particle and hole arrays. *Rev. Mod. Phys. 79*, 1267–1290.

GEUZAINE, C. AND REMACLE, J.-F. 2009. Gmsh: a three-dimensional finite element mesh generator with built-in pre- and post-processing facilities. *Int. J. Num. Meth. Engng 79,* 11, 1309–1331.

GEUZAINE, C. AND REMACLE, J.-F. 2012. Gmsh. `http://geuz.org/gmsh`.

GRÄSER, C. AND SANDER, O. 2012. Dune-FoamGrid. http://users.dune-project.org/projects/dune-foamgrid.

GRIFFITHS, D. J. 1998. *Introduction to Electrodynamics (3rd Edition)*. Benjamin Cummings.

HARBRECHT, H. AND SCHNEIDER, R. 2006. Wavelet Galerkin schemes for boundary integral equations—implementation and quadrature. *SIAM J. Sci. Comp. 27,* 4, 1347–1370.

HARRINGTON, R. F. AND HARRINGTON, J. L. 1996. *Field computation by moment methods*. Oxford University Press.

HEROUX, M. A., BARTLETT, R. A., HOWLE, V. E., HOEKSTRA, R. J., HU, J. J., KOLDA, T. G., LEHOUCQ, R. B., LONG, K. R., PAWLOWSKI, R. P., PHIPPS, E. T., SALINGER, A. G., THORNQUIST, H. K., TUMINARO, R. S., WILLENBRING, J. M., WILLIAMS, A., AND STANLEY, K. S. 2005. An overview of the Trilinos project. *ACM Trans. Math. Softw. 31,* 3, 397–423.

HIPTMAIR, R. 2006. Operator preconditioning. *Computers & Mathematics with Applications 52,* 5, 699–706.

HIPTMAIR, R. AND KIELHORN, L. 2012. BETL — a generic boundary element template library. Tech. Rep. 2012-36, Seminar for Applied Mathematics, ETH Zürich.

INTEL. 2012. Intel threading building blocks. http://threadingbuildingblocks.org.

KIELHORN, L. 2012. Boundary Element Template Library (BETL). http://www.sam.math.ethz.ch/betl.

KIRBY, R. C. 2010. From functional analysis to iterative methods. *SIAM Review 52,* 2, 269–293.

KITWARE. 2012a. Paraview. http://www.paraview.org.

KITWARE. 2012b. Visualizaton Toolkit (VTK). http://www.vtk.org.

KLEINMAN, R. AND MARTIN, P. 1988. On single integral equations for the transmission problem of acoustics. *SIAM J. Appl. Math. 48,* 2, 307–325.

KRESS, R. 1985. Minimizing the condition number of boundary integral operators in acoustic and electromagnetic scattering. *Q. J. Mechanics Appl. Math. 38,* 2, 323–341.

LANGER, U. AND STEINBACH, O. 2007. Coupled finite and boundary element domain decomposition methods. In *Boundary element analysis*, M. Schanz and O. Steinbach, Eds. Springer, 61–95.

LENOIR, M. AND SALLES, N. 2012. Evaluation of 3-d singular and nearly singular integrals in Galerkin BEM for thin layers. *SIAM J. Sci. Comp. 34,* 6, A3057–A3078.

LIU, H. AND LALANNE, P. 2008. Microscopic theory of the extraordinary optical transmission. *Nature 452*, 728–731.

MAISCHAK, M. 2013. Maiprogs. http://www.ifam.uni-hannover.de/~maiprogs.

MESSNER, M., MESSNER, M., URTHALER, P., AND RAMMERSTORFER, F. 2010. Hyperbolic and Elliptic Numerical Analysis (HyENA). http://portal.tugraz.at/portal/page/portal/Files/i2610/files/Forschung/Software/HyENA/html/index.html.

NÉDÉLEC, J.-C. 2001. *Acoustic and electromagnetic equations: Integral representations for harmonic problems*. Springer.

OF, G. 2008. An efficient algebraic multigrid preconditioner for a fast multipole boundary element method. *Computing 82,* 2-3, 139–155.

OF, G., STEINBACH, O., AND WENDLAND, W. L. 2006. The fast multipole method for the symmetric boundary integral formulation. *Ima J Numer Anal 26,* 2, 272–296.

OpenCASCADE 2012. OpenCASCADE: shape gallery. http://www.opencascade.org/showroom/shapegallery.

POLIMERIDIS, A. G. AND MOSIG, J. R. 2010. Complete semi-analytical treatment of weakly singular integrals on planar triangles via the direct evaluation method. *Intl J. Numer. Meth. Engng 83*, 1625–1650.

POLIMERIDIS, A. G., VIPIANA, F., MOSIG, J. R., AND WILTON, D. R. 2013. DIRECTFN: fully numerical algorithms for high precision computation of singular integrals in Galerkin SIE methods. *IEEE Trans. Antennas Propag. 61*, 3112–3122.

RAMACHANDRAN, P. 2004–2005. An introduction to Traited VTK (tvtk). http://docs.enthought.com/mayavi/tvtk/README.html.

RAVIART, P.-A. AND THOMAS, J.-M. 1977. A mixed finite element method for 2-nd order elliptic problems. In *Mathematical aspects of finite element methods*. Springer, 292–315.

RJASANOW, S. AND STEINBACH, O. 2007. *The Fast Solution of Boundary Integral Equations*. Springer, Berlin Heidelberg.

SANDERSON, C. 2012. Armadillo: C++ linear algebra library. http://arma.sourceforge.net.

SAUTER, S. A. AND SCHWAB, C. 2011. *Boundary Element Methods*. Springer Series in Computational Mathematics, 39. Springer, Berlin Heidelberg.

SCHMIDT, K. 2013. Concepts – a numerical C++ library. http://www.concepts.math.ethz.ch.

ŠOLÍN, P. 2005. *Partial differential equations and the finite element method*. Wiley.

STEINBACH, O. 2002. On a generalized $l2 projection and some related stability estimates in sobolev spaces. *Numer. Math. 90,* 4, 775–786.

STEINBACH, O. 2008. *Numerical Approximation Methods for Elliptic Boundary Value Problems*. Springer, New York.

SWIG 2012. Simplified Wrapper and Interface Generator (SWIG). `http://www.swig.org`.

SZABO, B. A. AND BABUŠKA, I. 1991. *Finite element analysis*. Wiley.

Trilinos 2012. Trilinos. `http://trilinos.sandia.gov`.

VAN DEN BOSCH, I. 2013. Puma-EM. `http://puma-em.sourceforge.net`.

WIELEBA, P. AND SIKORA, J. 2011. "BEMLAB"—universal, open source, boundary element method library applied in micro-electro-mechanical systems. *Studies in Appl. Electromagn. Mech. 35*, 173–182.