Eidgenössische Technische Hochschule Zürich
Ecole polytechnique fédérale de Zurich
Politecnico federale di Zurigo
Swiss Federal Institute of Technology Zurich

BETL – A generic boundary element template library

R. Hiptmair and L. Kielhorn

Research Report No. 2012-36
November 2012

Seminar für Angewandte Mathematik
Eidgenössische Technische Hochschule
CH-8092 Zürich
Switzerland

BETL — A generic boundary element template library

RALF HIPTMAIR, ETH Zurich
LARS KIELHORN, ETH Zurich

Many relevant physical phenomena are modeled via second order (elliptic) partial differential operators with constant coefficients. Well-known examples are steady-state heat conduction, electrostatics, eddy-currents, and acoustic or electro-magnetic scattering. In specific scenarios their numerical treatment by Boundary Element Methods is superior to that of volume based discretization schemes. However, up to now there exists no software package which allows for a unified treatment of those models in the context of Boundary Element Methods.

The Boundary Element Template Library (BETL) implements building blocks for 3D Galerkin discretizations of arbitrary boundary integral operators. It equally aims for rapid prototyping of new Boundary Element formulations as well as for the development of efficient and robust industrial strength solvers. To achieve the latter goal BETL provides interfaces to matrix compression algorithms to reduce the Boundary Element method's inherent quadratic complexity to almost linear costs.

BETL's implementation is based on generic software paradigms. Entirely written in C++, its core functionality is implemented as a header-only library, which makes it highly portable and attractive for being interfaced with existing C++ codes. In this work we will discuss the library's core design and demonstrate its use. Examples of specific simulations carried out with BETL will be discussed.

1. INTRODUCTION

Boundary Element Methods have proven to be expedient alternatives and/or additions to Finite Element Methods in many fields of engineering. Boundary Element Methods enjoy significant advantages in certain situations: They facilitate the treatment of exterior domain problems, allow the coupling with Finite Element Methods, and offer the ease of working with surface meshes alone rather than dealing with volume based discretizations. Unfortunately, there exist almost no software packages for the discretization of the boundary integral operators underlying the Boundary Element Method. The *Boundary Element Template Library* (BETL) presented herein is intended to mend this. BETL provides generic building blocks for the Galerkin discretization of a large class of boundary integral operators in 3D. Based upon these building blocks BETL can be used for rapid prototyping as well as for the development of efficient and robust Boundary Element programs for large-scale simulations.

A considerable number of open-source software packages in the context of Finite Elements is available. Examples are Deal.II [Bangerth et al. 2007], LibMesh [Kirk et al. 2006], Dune [Bastian et al. 2008; Dedner et al. 2010], and NGSolve [Schöberl 2012]. Contrary, less attention has been paid to the development of software libraries related to Boundary Element Methods. For obvious reasons: Boundary Element Methods are less versatile than Finite Element Methods and their implementation is more laborious.

There are a few Boundary Element packages, but these codes often do not meet the standards of modern programming techniques, or they implement particular Boundary Element Methods rather than featuring a library design [Liu 2012]. A recent and ongoing development of a Boundary Element Library is BEM++ [Arridge et al. 2012]. In contrast to BETL, BEM++ realizes all abstractions via an object-oriented design approach whereas BETL is build upon generic programming paradigms using the powerful template facilities of C++. In the following, we will give an overview on BETL's design principles and its current functionality.

Authors' addresses: R. Hiptmair and L. Kielhorn, Seminar of Applied Mathematics, ETH Zurich, Rämistrasse 101, CH-8092 Zürich; email: {ralf.hiptmair,lars.kielhorn}@sam.math.ethz.ch

The guiding design principles of BETL are:

— Modularity: lean interfaces and separation of data structures and algorithms, the gist of generic programming. This enables quick replacement and extension of parts of the library.
— Efficiency: static polymorphism offered by the template mechanism yields efficient code with minimum runtime overhead.
— Abstraction: data structures (e.g., those for discrete operators) mimic the mathematical structure of the underlying boundary integral operators.
— Easy interfacing: the header only library along with abstract data types allow smooth integration with/into other C++ codes.

BETL's development goes back to the beginning of 2010 and continues nowadays. Currently *(Nov. 2012)*, BETL provides the following features, most of which are available as core library functionality:

— Support for triangular and quadrilateral polynomial parametric surface meshes that allow for a surface approximation of up to fourth order.
— Piecewise polynomial boundary element spaces: continuous and discontinuous scalar boundary element spaces up to polynomial degree 3, surface edge element (RWG) spaces of lowest order.
— Transformation based quadrature routines for general (Cauchy) singular kernels [Sauter and Schwab 2011] along with semi-analytic evaluation routines for Laplace kernels for lowest order boundary element functions on flat triangular elements [Rjasanow and Steinbach 2007].
— Support of Laplace- and Helmholtz-type fundamental solutions.
— Assembly routines for *all* relevant boundary integral operators
— Seamless integration of the AHMED ACA matrix compression library [Bebendorf 2012]. BETL can exploit multi-threading furnished by AHMED.
— Modules for dense (Boost/UBLAS [Ublas 2012]) and sparse (SuiteSparse, [Davis 2012]) numerical linear algebra. However, the generic programming model makes it easy to link any reasonable general linear algebra package like the linear algebra routines of Trilinos [Heroux et al. 2005] or PETsc [Balay et al. 2012] with BETL.
— BETL comes with iterative solvers (CG, GMRes). In addition, BETL implements interfaces to sparse direct solvers (SuperLU, UMFPACK, Pardiso).
— Operator preconditioning based on dual meshes and Calderón identities [Hiptmair 2006].
— Support of native file formats from the open-source pre- and postprocessing tools Gmsh [Geuzaine and Remacle 2009] and VTK [Schroeder et al. 2006] (ParaView, MayaVi).

BETL's installation is based on the CMake build system [CMake 2012]. The library has been tested for various operating systems (Linux/Unix, OS X, Windows) as well as with different compilers such as the most recent versions of g++, icc, and clang.

BETL is free for academic and non-commercial use. More information about BETL in general and licensing in particular may be found at:

`http://www.sam.math.ethz.ch/betl/`

The remainder of this work is organized as follows: In Sec. 2 we will present the basics of boundary integral operators and boundary element methods. Sec. 3 is devoted to BETL's design principles as well as to its implementation. A set of validating and application-oriented examples is given in Sec. 4 and, finally, Sec. 5 is concerned with a summary and a brief discussion on BETL's limitations.

2. BOUNDARY INTEGRAL EQUATIONS AND BOUNDARY ELEMENT METHODS

Very briefly we recall the basic properties of boundary integral operators as well as the main concepts of their discretizations. A detailed discussion is far beyond the scope of the present work. For excellent and detailed mathematical treatises on boundary integral equations the reader is referred to [Hackbusch 1989; Hsiao and Wendland 2008; McLean 2000; Steinbach 2008]. In addition, the book of [Sauter and Schwab 2011] is highly recommended for both boundary integral equations as well as for an in-depth understanding of Boundary Element Methods. From an engineering point of view, the books of Bonnet [1995], Gaul et al. [2003], and Sutradhar et al. [2008] serve as introductions to that topic.

2.1. Boundary integral operators and Boundary Element Methods

Let $\mathcal{L}$ be a second order elliptic differential operator with constant coefficients. A general boundary (transmission) value problem then reads

$$\begin{aligned} \mathcal{L}u = 0 \qquad & \text{in} \quad \Omega \\ \text{+ boundary (transmission) conditions} \qquad & \text{on} \quad \Gamma \end{aligned} \tag{1}$$

where u denotes an unknown function sought in a *(possibly unbounded)* domain $\Omega \subset \mathbb{R}^3$. The domain's boundary is denoted by Γ and is assumed to be bounded and piecewise smooth. Important examples of second order elliptic partial differential operators with constant coefficients are

- Laplace operator: $\mathcal{L} = -\Delta$
- Helmholtz operator: $\mathcal{L} = -\Delta + \kappa^2$, $\kappa \in \mathbb{C}$
- Maxwell operator: $\mathcal{L} = \mathbf{curl}\,\mathbf{curl} + \kappa^2$, $\kappa \in \mathbb{C}$
- Linear elastostatics: $\mathcal{L} = \mu\,\mathbf{curl}\,\mathbf{curl} - (\lambda + 2\mu)\,\mathbf{grad}\,\mathrm{div}$, λ, μ: Lamé parameters

It can be shown, albeit with considerable effort, that the solution u of (1) satisfies the following variational equations

$$\begin{aligned} -\langle(\tfrac{1}{2}\,\mathrm{Id} + K)\gamma_0 u, \varphi\rangle_\Gamma + \langle V\gamma_1 u, \varphi\rangle_\Gamma = 0 \\ \langle W\gamma_0 u, \phi\rangle_\Gamma - \langle(\tfrac{1}{2}\,\mathrm{Id} - K')\gamma_1 u, \phi\rangle_\Gamma = 0 \end{aligned} \tag{2}$$

for all test functions (ϕ, φ) in suitable trace spaces on Γ.

The expressions γ_0, γ_1 denote the Dirichlet and Neumann trace operators. They map functions on Ω to functions on the boundary. Trace operators are closely connected with the partial differential operators and they are

— for the Laplace/Helmholtz operators:

$$\gamma_0 u(\mathbf{x}) := \lim_{\Omega \ni \tilde{\mathbf{x}} \to \mathbf{x} \in \Gamma} u(\tilde{\mathbf{x}}) \,, \quad \gamma_1 u(\mathbf{x}) := \gamma_0\,\mathbf{grad}_{\tilde{x}}\, u(\tilde{\mathbf{x}}) \cdot \mathbf{n}(\mathbf{x}) \tag{3}$$

— for the Maxwell operator:

$$\boldsymbol{\gamma}_0 \mathbf{u} := \mathbf{n} \times \boldsymbol{\gamma}_\times \mathbf{u} \,, \quad \boldsymbol{\gamma}_1 \mathbf{u} := \boldsymbol{\gamma}_\times(\mathbf{curl}\,\mathbf{u}) \,, \quad \boldsymbol{\gamma}_\times \mathbf{u}(\mathbf{x}) := \lim_{\Omega \ni \tilde{\mathbf{x}} \to \mathbf{x} \in \Gamma} (\mathbf{u}(\tilde{x}) \times \mathbf{n}(\mathbf{x})) \tag{4}$$

The operators occurring in (2) are the identity operator Id, the single layer boundary integral operator V, the double layer boundary integral operator K, its adjoint operator K', and the hypersingular operator W [see Sauter and Schwab 2011, Ch. 3].

Naturally, the detailed definitions of the above boundary integral operators depend on the underlying partial differential operator $\mathcal{L}$. However, all of them follow a similar pattern. For a generic boundary integral operator T we have the following abstract form

$$\langle Tu, w\rangle_\Gamma = \int_\Gamma \int_\Gamma \{D_{\mathbf{x}} w(\mathbf{x}), D_{\mathbf{y}} u(\mathbf{y})\}_{G(\mathbf{y},\mathbf{x})}\, \mathrm{d}s_{\mathbf{y}}\, \mathrm{d}s_{\mathbf{x}} \,. \tag{5}$$

Here, $\{f,g\}_A$ denotes the product

$$\{f,g\}_A := f(\mathbf{x})\,A(\mathbf{y},\mathbf{x})\,g(\mathbf{y}) \tag{6}$$

where f and g are scalar valued functions. The function A is supposed to be real- or complex valued, i.e., $A \in \mathbb{R}|\mathbb{C}$. A natural extension to vector valued functions is given by

$$\{\mathbf{f},\mathbf{g}\}_{\mathbf{A}} := \mathbf{f}^\top(\mathbf{x}) \cdot \mathbf{A}(\mathbf{y},\mathbf{x}) \cdot \mathbf{g}(\mathbf{y}) \tag{7}$$

with $\mathbf{A} \in (\mathbb{R}|\mathbb{C})^{d\times d}$ where $d=3$, or $d=1$. Note that for vector valued functions holds $\{\mathbf{f},\mathbf{g}\}_{\mathbf{A}} = \{\mathbf{g},\mathbf{f}\}_{\mathbf{A}^\top}$.

In (5), the operators $D_{\mathbf{x}}$, $D_{\mathbf{y}}$ are optional. If they are present, they represent first order surface differential operators such as surface curl, surface divergence, or surface gradient operators, respectively, refer to [Hsiao and Wendland 2008] for their proper definitions. Further, the function $G(\mathbf{y},\mathbf{x})$ is a singular function coinciding with or closely related to a *fundamental solution* for $\mathcal{L}$. This fundamental solution is the a key building block of any boundary element method. It is defined as the distributional solution of

$$\mathcal{L}^* G(\mathbf{z}) = \delta(\mathbf{z}) \qquad \mathbf{z} \in \mathbb{R}^3$$

with a Dirac distribution δ as an inhomogeneity. The operator $\mathcal{L}^*$ is the adjoint of $\mathcal{L}$.

To generate Galerkin discretizations of boundary integral operators BETL targets them in the form (5). In the remainder of this section we will discuss particular incarnations of the abstract operator (5). We will focus on the geometric approximation of the boundary as well as on suitable discrete test and trial spaces.

For the boundary approximation we introduce a surface mesh $\mathcal{G}$ as a set of disjoint elements τ

$$\Gamma \approx \Gamma_h := \overline{\cup_{\tau\in\mathcal{G}}\tau}\ .$$

Currently, BETL supports six different element types based on polynomial geometry approximations: 3-noded, 6-noded, and 10-noded triangular elements as well as 4-noded, 8-noded, and 16-noded quadrilateral elements (see Fig. 1). Elements are defined via 2-dimensional triangular or quadrilateral reference elements. Their definitions are [see Sauter and Schwab 2011, Ch. 4]

$$\text{Unit triangle:} \quad \hat{T} := \left\{(\hat{x}_1,\hat{x}_2) \in \mathbb{R}^2 \colon 0 < \hat{x}_2 < \hat{x}_1 < 1\right\}$$
$$\text{Unit square:} \quad \hat{Q} := (0,1)^2\ .$$

We will use the generic notation $\hat{\tau}$ in order to refer to a reference element. Every element τ is then defined as the image of a reference element $\hat{\tau}$ under a component-wise polynomial reference mapping χ

$$\chi\colon \hat{\tau} \to \tau\ . \tag{8}$$

It remains to specify the discrete test- and trial-spaces. These spaces are given with respect to the trace operators (3). Hence, for the Laplace and Helmholtz operators we choose continuous function spaces for the Dirichlet data while Neumann related data will be modeled via discontinuous functions in order to allow jumps at edges or corners. These considerations lead to the following definitions. Let

$$Y_h^\alpha(\Gamma) := \operatorname{span}\{\phi_\ell^\alpha\}_{\ell=1}^L \tag{9}$$

be the finite dimensional space of piecewise continuous basis functions of order α and let

$$Z_h^\beta(\Gamma) := \operatorname{span}\left\{\varphi_k^\beta\right\}_{k=1}^K \tag{10}$$

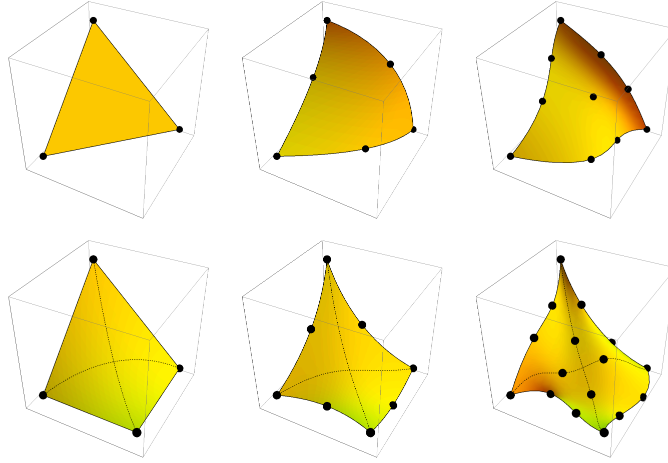

Fig. 1. Supported element types. The first row illustrates the available triangular elements τ. The second row shows the equivalent quadrilateral elements. From left to right: (Bi-)Linear, quadratic, and cubic shape approximations. The shape functions and the local ordering of vertices coincide with the local Lagrangian finite element basis functions (cf. Sec. 3.1, Fig. 3).

be the space of piecewise discontinuous basis functions of order β. The detailed definitions of those spaces can be found in any textbook which covers 2-dimensional Finite Element Methods (e.g., [Braess 2007]).

The discrete spaces for the Maxwell system fundamentally differ from those for the Laplace and Helmholtz operators since the respective trace spaces are of different nature. The traces (4) induce spaces of tangential vector fields. As finite dimensional subspaces BETL currently implements lowest order Raviart-Thomas surface finite elements and a rotated version known as RWG elements or surface edge elements. We adopt the notations

$$\mathbf{Y}_{\perp,h}(\mathrm{curl}_\Gamma, \Gamma) := \mathrm{span}\,\{\boldsymbol{\phi}_\ell\}_{\ell=1}^{L} \tag{11}$$

for the space of surface edge functions, whose surface curls are square integrable, and

$$\mathbf{Z}_{\parallel,h}(\mathrm{div}_\Gamma, \Gamma) := \mathrm{span}\,\{\boldsymbol{\varphi}_k\}_{k=1}^{K} \tag{12}$$

for the space of surface Raviart-Thomas function with square integrable surface divergence. They feature locally supported bases comprising functions associated with the edges of $\mathcal{G}$. Their local basis functions read

$$\boldsymbol{\varphi}_k|_\tau\,(\mathbf{x}) := \frac{1}{\sqrt{g_\tau}}\,(\mathbf{x}_{i_k} - \mathbf{x}) \qquad \boldsymbol{\phi}_\ell|_\tau\,(\mathbf{x}) := \boldsymbol{\varphi}_\ell(\mathbf{x}) \times \mathbf{n}(\mathbf{x})$$

where g_τ is the gram determinant of an element τ and i_k is the index of the node lying opposite to edge k. Further details on those functions can be found in, e.g., [Hiptmair 2007].

Endowed with the boundary approximation as well as the set of discrete test- and trial-spaces we now present the discretized boundary integral operators currently pro-

vided by BETL. The operators for Laplace/Helmholtz boundary value problems are

$$\begin{aligned} V_h[i,k] &:= \langle V\varphi_k^\beta, \varphi_i^\beta\rangle = \int_{\mathrm{supp}(\varphi_i^\beta)}\int_{\mathrm{supp}(\varphi_k^\beta)} \{\varphi_i^\beta, \varphi_k^\beta\}_G \, \mathrm{d}s_{\mathbf{y}}\, \mathrm{d}s_{\mathbf{x}} \;, \\ K_h[i,\ell] &:= \langle K\phi_\ell^\alpha, \varphi_i^\beta\rangle = \int_{\mathrm{supp}(\varphi_i^\beta)}\int_{\mathrm{supp}(\phi_\ell^\alpha)} \{\varphi_i^\beta, \phi_\ell^\alpha\}_{\partial_{\mathbf{n}(\mathbf{y})}G} \, \mathrm{d}s_{\mathbf{y}}\, \mathrm{d}s_{\mathbf{x}} \;, \\ K'_h[j,k] &:= \langle K'\varphi_k^\beta, \phi_j^\alpha\rangle = \int_{\mathrm{supp}(\phi_j^\alpha)}\int_{\mathrm{supp}(\varphi_k^\beta)} \{\phi_j^\alpha, \varphi_k^\beta\}_{\partial_{\mathbf{n}(\mathbf{x})}G} \, \mathrm{d}s_{\mathbf{y}}\, \mathrm{d}s_{\mathbf{x}} \;. \end{aligned} \tag{13}$$

For the Laplace operator the function $G \to G_L$ is replaced with

$$G_L(\mathbf{x},\mathbf{y}) = \frac{1}{4\pi}\frac{1}{|\mathbf{y}-\mathbf{x}|}$$

while for the Helmholtz operator there holds the substitution $G \to G_H^\kappa$ with

$$G_H^\kappa(\mathbf{x},\mathbf{y}) = \frac{1}{4\pi}\frac{\exp(\kappa|\mathbf{y}-\mathbf{x}|)}{|\mathbf{y}-\mathbf{x}|} \;.$$

The expression $\partial_{\mathbf{n}(\cdot)}G$ is the normal derivative $\partial_{\mathbf{n}(\cdot)}G := \mathbf{grad}_{(\cdot)}\, G \cdot \mathbf{n}(\cdot)$. In (13) the expression $\mathrm{supp}(\psi)$ denotes the support of a function ψ.

The hypersingular operator cannot be expressed as a bona fide improper or Cauchy-singular integral operator. Here, the Galerkin approach offers a major benefit, because it can exploit the well-known weak expressions for hypersingular operators that emerge from integration by parts [Maue 1949; Nedelec 1982]. Thus, the bilinear forms arising from the hypersingular operator for the Laplace and Helmholtz operators can be reduced to the weakly singular operators

$$W_h[j,\ell] := \langle W\phi_\ell^\alpha, \phi_j^\alpha\rangle = \int_{\mathrm{supp}(\phi_j^\alpha)}\int_{\mathrm{supp}(\phi_\ell^\alpha)} \{\mathbf{curl}_{\Gamma,\mathbf{x}}\, \phi_j^\alpha, \mathbf{curl}_{\Gamma,\mathbf{y}}\, \phi_\ell^\alpha\}_{G_L} \, \mathrm{d}s_{\mathbf{y}}\, \mathrm{d}s_{\mathbf{x}} \;, \tag{14}$$

for the Laplace operator and

$$\begin{aligned} W_h[j,\ell] := \langle W\phi_\ell^\alpha, \phi_j^\alpha\rangle = &\int_{\mathrm{supp}(\phi_j^\alpha)}\int_{\mathrm{supp}(\phi_\ell^\alpha)} \{\mathbf{curl}_{\Gamma,\mathbf{x}}\, \phi_j^\alpha, \mathbf{curl}_{\Gamma,\mathbf{y}}\, \phi_\ell^\alpha\}_{G_H^\kappa} \, \mathrm{d}s_{\mathbf{y}}\, \mathrm{d}s_{\mathbf{x}} \\ &+ \kappa^2 \int_{\mathrm{supp}(\phi_j^\alpha)}\int_{\mathrm{supp}(\phi_\ell^\alpha)} \{\phi_j^\alpha \mathbf{n}(\mathbf{x}), \phi_\ell^\alpha \mathbf{n}(\mathbf{y})\}_{G_H^\kappa} \, \mathrm{d}s_{\mathbf{y}}\, \mathrm{d}s_{\mathbf{x}} \end{aligned}$$

for the Helmholtz operator.

Finally, the discrete boundary integral operators for the Maxwell system are based on the spaces (11) and (12). Their definitions are

$$\begin{aligned} V_h[i,k] &:= \langle V\boldsymbol{\varphi}_k, \boldsymbol{\varphi}_i\rangle = \int_{\mathrm{supp}(\boldsymbol{\varphi}_i)}\int_{\mathrm{supp}(\boldsymbol{\varphi}_k)} \{\boldsymbol{\varphi}_i, \boldsymbol{\varphi}_k\}_{G_H^\kappa} \, \mathrm{d}s_{\mathbf{y}}\, \mathrm{d}s_{\mathbf{x}} \\ &\qquad - \frac{1}{\kappa^2}\int_{\mathrm{supp}(\boldsymbol{\varphi}_k)}\int_{\mathrm{supp}(\boldsymbol{\varphi}_i)} \{\mathrm{div}_\Gamma\, \boldsymbol{\varphi}_i, \mathrm{div}_\Gamma\, \boldsymbol{\varphi}_k\}_{G_H^\kappa} \, \mathrm{d}s_{\mathbf{y}}\, \mathrm{d}s_{\mathbf{x}} \\ K_h[i,\ell] &:= \langle K\boldsymbol{\phi}_\ell, \boldsymbol{\varphi}_i\rangle = \int_{\mathrm{supp}(\boldsymbol{\varphi}_i)}\int_{\mathrm{supp}(\boldsymbol{\phi}_\ell)} \{\boldsymbol{\varphi}_i, \boldsymbol{\phi}_\ell\}_{\mathbf{G}(G_H^\kappa;\partial_{\mathbf{y}};\mathbf{n}(\mathbf{y}))} \, \mathrm{d}s_{\mathbf{y}}\, \mathrm{d}s_{\mathbf{x}} \\ K'_h[j,k] &:= \langle K'\boldsymbol{\varphi}_k, \boldsymbol{\phi}_j\rangle = \int_{\mathrm{supp}(\boldsymbol{\phi}_j)}\int_{\mathrm{supp}(\boldsymbol{\varphi}_k)} \{\boldsymbol{\phi}_j, \boldsymbol{\varphi}_k\}_{\mathbf{G}'(G_H^\kappa;\partial_{\mathbf{x}};\mathbf{n}(\mathbf{x}))} \, \mathrm{d}s_{\mathbf{y}}\, \mathrm{d}s_{\mathbf{x}} \\ W_h[j,\ell] &:= \langle W\boldsymbol{\phi}_\ell, \boldsymbol{\phi}_j\rangle = \kappa^2\langle V\boldsymbol{\varphi}_\ell, \boldsymbol{\varphi}_j\rangle = \kappa^2 V_h[j,\ell] \;. \end{aligned} \tag{15}$$

The functions $\mathbf{G} \in \mathbb{C}^{3\times 3}$ and $\mathbf{G}' \in \mathbb{C}^{3\times 3}$ are

$$\begin{aligned}\mathbf{G}(G_H^\kappa;\partial_\mathbf{y};\mathbf{n}(\mathbf{y})) &:= \frac{\partial G_H^\kappa}{\partial \mathbf{n}(\mathbf{y})}\mathbf{I} - \mathbf{n}(\mathbf{y}) \otimes \mathbf{grad}_\mathbf{y}\, G_H^\kappa \\ \mathbf{G}'(G_H^\kappa;\partial_\mathbf{x};\mathbf{n}(\mathbf{x})) &:= \frac{\partial - G_H^\kappa}{\partial \mathbf{n}(\mathbf{x})}\mathbf{I} + \mathbf{grad}_\mathbf{x}\, G_H^\kappa \otimes \mathbf{n}(\mathbf{x})\end{aligned} \tag{16}$$

where $\otimes$ denotes the outer product between two vectors. $\mathbf{I}$ stands for the identity matrix. Note that for (16) the following identities hold

$$\begin{aligned}-\mathbf{curl}_\mathbf{x}\left[G_H^\kappa(\mathbf{y}-\mathbf{x})\left(\mathbf{w}(\mathbf{y})\times\mathbf{n}(\mathbf{y})\right)\right] &= \mathbf{G}\cdot\mathbf{w}(\mathbf{y}) \\ \mathbf{curl}_\mathbf{x}\left[G_H^\kappa(\mathbf{y}-\mathbf{x})\mathbf{v}(\mathbf{y})\right]\times\mathbf{n}(\mathbf{x}) &= \mathbf{G}'\cdot\mathbf{v}(\mathbf{y})\ .\end{aligned} \tag{17}$$

The expressions in (17) usually arise in detailed derivations of the boundary integral operators for the Maxwell system, see, e.g., [Hiptmair 2007].

3. DESIGN AND IMPLEMENTATION OF BETL

In the former section we introduced the concept of Galerkin Boundary Element Methods by means of the Laplace, Helmholtz, and Maxwell systems. Now, we will focus on BETL's specific design concepts which are intended to provide sufficient abstractions for the computation of the corresponding discrete boundary integral operators.

3.1. The design of BETL

We distinguish two categories of structures necessary for setting up the discrete boundary integral operators. The first, called the *discretization model*, is connected with the Galerkin discretization of linear operators by means of spaces spanned by locally supported basis functions on a mesh. This is what all finite element methods have in common. The second category, the *BEM model*, comprises structures dealing with tasks that are specifically encountered in Galerkin Boundary Element methods, in particular the local computations yielding matrix entries like, e.g., $V_h[i,j]$ from (13).

On overview of the most important BETL structures in these two categories is given in Fig. 2.

Generic finite element aspects (Discretization model). Structures of this category provide information about the mesh topology along with information about the discrete finite element space. They are essential for the setup of any Galerkin scheme. Below, we give short descriptions of the most important data types:

`Element` Elements are solely defined by the number of vertices: E.g., 6 vertices define a quadratic triangle while 4 vertices define a bilinear quadrilateral (see Fig. 1). Therefore, an `Element` is a container which stores the list of pointers to its vertices. The local numbering of vertices follows common conventions that, e.g., can be found in [Geuzaine and Remacle 2009]. Vertices are ordinary points $\mathbf{x} \in \mathbb{R}^3$. In BETL they are implemented by the structure `Vertex` which, along with the coordinates, additionally provides an unique identifier to each vertex. Both, the local vertex ordering as well as the list of vertex pointer encode the mapping χ from Eqn. (8).

In BETL, element data is solely accessed via the intermediate class `ElementTraits`. More information concerning this structure is given in Sec. 3.2. For examples, see Sec. 4.

`Mesh` A `Mesh` is a container for storing the union of all elements and all vertices. It also stores adjacency information. Access to its topology data is granted via iterator interfaces. This class is designed to be a lightweight

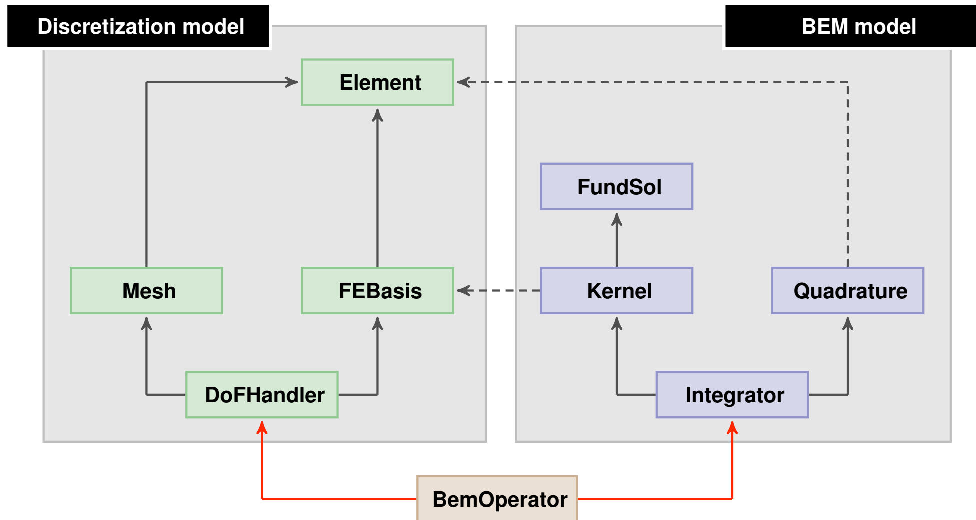

Fig. 2. Outline and connections between the fundamental abstractions of BETL. The notion $A \rightarrow B$ has to be understood as: *Type A depends on type B*. The left part subsumes all classes related to the discretization scheme while the right part bundles the BEM specific classes. The dashed arrows represent dependencies between the BEM specific part and the discretization scheme. However, structures of the discretization model need not be aware of the BEM model. In the end, all structures define the `BemOperator` which computes a discrete boundary integral operator.

class providing only a basic functionality and a limited number of interfaces. For this, it can be easily exchanged with other, more potent mesh classes.

`FEBasis` Based on the `Element` class the `FEBasis` defines functors for the pointwise evaluation of local basis functions assigned to one of the discrete finite element spaces (9), (10), and (11), (12). The local basis functions' numbering for one reference element is given in Fig. 3 and in Fig. 4. Two arguments are passed to the evaluation functors: First, local points on the reference element $\hat{\tau}$, and, secondly, a global element τ. Further details on the `FEBasis` are given in Sec. 3.2. Examples are given in Fig. 7, and in Fig. 15.

`DoFHandler` Based on the `FEBasis`, this class encodes the mapping of local basis functions defined on elements to global basis functions, that is, spanning the Finite Element space either with the complete mesh or with a given set of elements. Fig. 5 gives examples on the global numbering and location of basis functions. Note that the separation of information related to the Finite Element space and topological information allows for the definition of multiple Finite Element spaces on the same geometry. The importance of this concept becomes obvious in the example given in Sec. 4.5. Once the `DoFHandler` has been initialized via the `distributeDoFs` member method it gives access to its data through iterator interfaces.

Remark: The `DoFHandler` structure is directly based on classes of the same name as they have been introduced in the Finite Element Library Deal.II. Refer to Bangerth et al. [2007] for more details about it.

BEM specific aspects (BEM model). The BEM model must deal with the kernel functions' non-locality as well as with their singular nature. These peculiarities are inher-

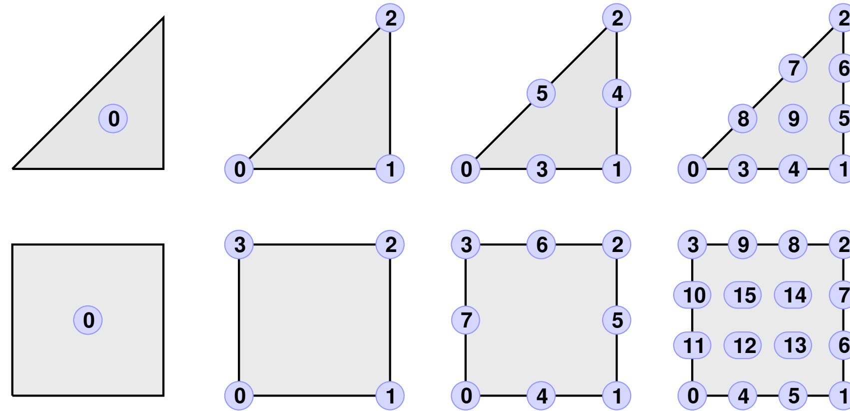

Fig. 3. Local numbering for BETL's Lagrangian finite element basis functions. Upper row from left to right: Constant, linear, quadratic, and cubic bases for triangular elements. Lower row from left to right: Constant, bi-linear, *(serendipity-)*quadratic, and cubic bases for quadrilateral elements.

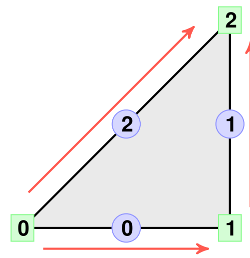

Fig. 4. Local numbering for BETL's RWG/Raviart-Thomas basis functions. Only lowest order edge functions for flat triangular elements are currently supported. The arrows indicate the edges' intrinsic positive orientations based on the element's node numbering *(from smaller to larger node index)*.

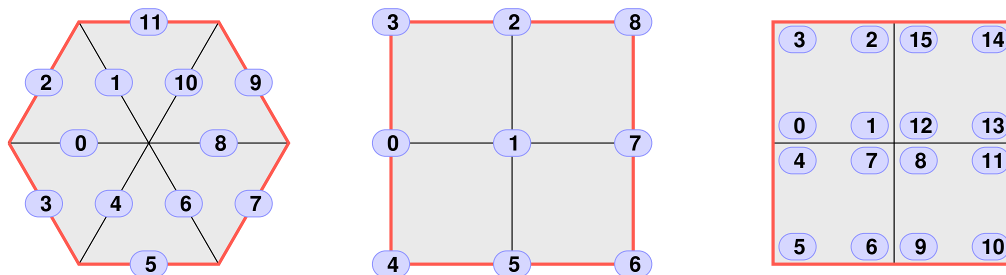

Fig. 5. *Left:* Global numbers of basis functions for edge functions $\boldsymbol{\phi}_h \in \mathbf{Y}_{\perp,h}$ or $\boldsymbol{\varphi}_h \in \mathbf{Z}_{\|,h}$. *Center:* Linear continuous Lagrangian functions $\phi_h^1 \in Y_h^1$. *Right:* Linear discontinuous Lagrangian functions $\varphi_h^1 \in Z_h^1$. The red outer lines represent the boundary's boundary $\partial\Gamma_h$. Degrees of freedom attached to $\partial\Gamma$ may be suppressed such that spaces like, e.g., $Y_h^{1,*}(\phi) := \{\phi \in Y_h^1 : \phi = 0 \text{ on } \partial\Gamma_h\}$, are realized.

ent to Boundary Element Methods and their treatment. For instance, the fact of being faced with singular integrands adds a considerable amount of complexity to the overall numerical scheme: Some very special integrands might be treated by specific and fine tuned quadrature techniques. The design must consider these situations and has to allow for an easy exchange of quadrature and integration schemes. In the following, a brief description of the essential classes is given:

`FundSol` This structure implements the function A (or $\mathbf{A}$) from (6) (or (7)). Hence, it evaluates the fundamental solutions together with their respective traces in form of

$$G(\mathbf{x},\mathbf{y}) = G(|\mathbf{y}-\mathbf{x}|, \tfrac{\mathbf{y}-\mathbf{x}}{|\mathbf{y}-\mathbf{x}|}, \mathbf{n}(\mathbf{x}), \mathbf{n}(\mathbf{y}))\ . \tag{18}$$

The `FundSol` class is implemented as a functor. It might be instantiated with a parameter like, e.g., the wavenumber κ.

`Kernel` By using the `FundSol` class and together with the finite element spaces (cf. Fig. 2), this structure implements the inner products (6) and (7). They are evaluated at given local points $\boldsymbol{\xi}$ and $\boldsymbol{\eta}$ on the reference element $\hat{\tau}$ *(usually these local points are Gaussian points)*

$$k(\boldsymbol{\xi},\boldsymbol{\eta};\varphi,\phi,G) = \{\varphi,\phi\}_{G(\mathbf{x}(\boldsymbol{\xi}),\mathbf{y}(\boldsymbol{\eta}))}\ .$$

`Quadrature` This class generates appropriate pairs of local points $(\boldsymbol{\xi},\boldsymbol{\eta}) \in \mathbb{R}^2 \times \mathbb{R}^2$ on the reference element $\hat{\tau}$ which underlies τ. It also computes the integration weights. More information on the quadrature are given below and in Sec. 3.2. Examples can be found in Sec. 4.3.

`Integrator` This class performs the quadrature, i.e., it evaluates a weighted sum of kernel evaluations. Thus, the quadrature passes its data to the integrator which, then, evaluates the kernel. The integrator is implemented as functor. More details are given below and in Sec. 3.2.

The `Quadrature` and the `Integrator` structures are of such an exceptional importance that we will discuss them in a bit more detail. Both structures must take the kernel's singular behavior into account. The `Integrator` structure determines whether or not a singular quadrature scheme has to be applied. For this we distinct four different situations (cf. Fig. 6):

— Regular case: Two elements feature a positive distance for all points $\mathbf{x} \in \tau_{\mathbf{x}}$, $\mathbf{y} \in \tau_{\mathbf{y}}$
— Vertex adjacent case: Two elements share a common point
— Edge adjacent case: Two elements share a common edge
— Coincident case: Two elements coincide, i.e., $\tau_{\mathbf{x}} \equiv \tau_{\mathbf{y}}$

Naturally, the same considerations hold in the case of quadrilateral elements and also for combinations of quadrilaterals and triangles.

For each of these combinations special quadrature rules exist [Erichsen and Sauter 1998; Sauter and Schwab 2011]. These quadrature rules split the integration domain into several subdomains and apply special transformation rules to the integration points of each subdomain. Within BETL these steps are performed by the `Quadrature` structure. This structure gets the type of singularity from the `Integrator` and then returns the transformed integration points as well as some modified integration weights back to the `Integrator`.

As already mentioned a very few kernels such as the Laplace kernel on flat triangular elements with constant or linear finite element spaces allow for (semi-)analytic integrations. Semi-analytic integrations perform the inner integration exact and use a regular quadrature rule for the remaining outer integration. The resulting formulas

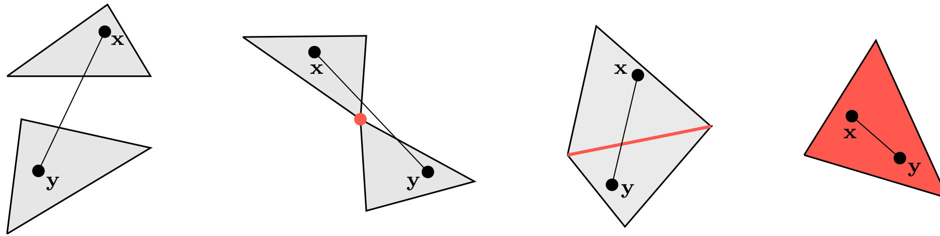

Fig. 6. Distinction of quadrature cases. *From left to right:* Regular integration: $\tau_{\mathbf{x}}$ and $\tau_{\mathbf{y}}$ feature a positive distance for all $\mathbf{x}$ and $\mathbf{y}$. Vertex adjacent integration: $\tau_{\mathbf{x}}$ and $\tau_{\mathbf{y}}$ share a common point. Edge adjacent integration: $\tau_{\mathbf{x}}$ and $\tau_{\mathbf{y}}$ share a common edge. Coincident integration: $\tau_{\mathbf{x}} \equiv \tau_{\mathbf{y}}$.

can be, e.g., found in [Rjasanow and Steinbach 2007] and BETL allows also for their use. In the following section we will present an example on how the default integrator can be replaced by a semi-analytic scheme.

3.2. The implementation of BETL

The implementation of the previously discussed design is entirely done in C++ [Stroustrup 2000]. The reasons for this choice are the language's high level abstraction facilities together with C++'s ability of permitting the development of efficient runtime code which is comparable to that generated by procedural languages like, e.g., C and/or Fortran.

BETL's implementation is almost exclusively based on generic programming techniques. Contrary to all other programming paradigms, this paradigm provides an uncompromising combination of abstraction and efficiency. In C++, generic programming is realized via templates. The most prominent example of a generic software library is the Standard Template Library (STL) [Musser et al. 2009] which now is part of the C++ standard library. Another well-known template library is Boost [Karlsson 2006] which extends the STL significantly. Both of these libraries are exhaustively used within BETL. Basically, generic programming techniques replace the dynamic polymorphism, which in C++ is realized via virtual methods, by a static *(or compile-time)* polymorphism. This allows the compiler for applying the full range of optimization techniques such that performance penalties during runtime can be avoided or at least minimized. For a deep insight into generic software concepts we refer to the textbooks of Vandervoorde and Josuttis [2002], Abrahams and Gurtovoy [2004] or the book of Alexandrescu [2004]. In terms of mathematical software the linear algebra packages Blitz++ [Veldhuizen 1998] and the Matrix Template Library (MTL) [Gottschling et al. 2007; Siek and Lumsdaine 1999] are heavily based on generic software concepts. Linear algebra projects are well-suited to be implemented via generic concepts since it is a natural idea to realize, e.g., a matrix-vector product in a generic manner and then apply this algorithm to single precision, double precision, or even complex data types. However, when it comes to the numerical solution of partial differential equations the separation of algorithms and data types is not that obvious any longer. Maybe for this reason, there exist only a very few Finite Element software packages which are accomplished via generic programming techniques (such as parts of the open source FEM library DUNE [Bastian et al. 2008; Dedner et al. 2010]). In most instances templates occur only as placeholders for numeric data types or as patterns for the dimensionality of the underlying physical problem. A generic implementation of a Finite Element Method has been proposed by Cirak and Cummings [2008] and BETL adopts many of the ideas presented therein.

A very classical and powerful generic design principle is the use of traits classes. Traits are used to map data types to other data types or to establish a connection between numbers and data types or vice versa. This concept is explained in detail in [Vandervoorde and Josuttis 2002] and has been applied in the context of Finite Element Methods in [Cirak and Cummings 2008]. For instance, BETL uses a structure named `ElementTraits` to access all element related data. The use of `ElementTraits` enables the user to exchange the underlying element type by a self-written data type. For instance, the library might be coupled to a Finite Element code which comes with its own element data types. In this case the developer needs to specify only the traits structure for the particular element type. Once this is done no further modifications need to be imposed. Moreover, the traits interface is completely resolved during compile time such that no runtime penalties will occur due to this indirection. BETL's implementation of the `ElementTraits` structure is directly taken from [Cirak and Cummings 2008].

Another example of a traits structure is given by means of the `FEBasis` structure which determines the distribution of the degrees of freedom on one reference element (see Fig. 3 and 4). For the Lagrangian based spaces an abstract declaration of the respective traits class reads as

```
template< enum REFERENCE_ELEMENTTYPE RE, enum APPROX_ORDER AO >
struct LagrangeTraits { /* empty */ };
```

The specialization for a triangular element with linear functions is given by

```
template< >
struct LagrangeTraits< TRIANGLE, LINEAR > {
  static const APPROX_ORDER approximation = LINEAR;
  static const FE_BASIS fe_basis     = LAGRANGE;
  static const unsigned NumFuncFace  = 0;
  static const unsigned NumFuncEdge  = 0;
  static const unsigned NumFuncNode  = 3;
  static const unsigned FuncDim      = 1;
  static const unsigned NumFunc =
    NumFuncFace + NumFuncEdge + NumFuncNode;
};
```

Equivalently, the `LagrangeTraits` structure needs to be specified also for the quadrilateral elements as well as for all remaining discrete spaces. Hence, we end up with eight specializations for this special discrete function space. Again, during compile time this traits interface is completely resolved and will not be present in the resulting binary.

BETL's implementation will be explained by means of the creation of the discrete double layer operator for the Laplace equation with lowest order test- and trail-spaces and by using 2nd order triangular elements. A fully functional code example is given in Fig. 7. Note that this code example is completely equivalent to the design scheme from Fig. 2.

More detailed descriptions of this piece of code are given below.

2,3: Everything starts with the parsing of a 2-dimensional surface mesh. Currently, BETL supports the import of mesh files generated by the open source tool Gmsh [Geuzaine and Remacle 2009]. Note that the import of a mesh file is handled by a separate library so that new mesh file formats can be easily added.

5–8: Here we define the lowest order basis functions for the Galerkin scheme. Since we are aiming at a discretisation of the double layer operator for the Laplace equation (Eqn. (13)) we choose the test- and trial-spaces according to Z_h^0 and Y_h^1, respectively.

```
// elements will be 6-noded curved triangles
typedef Element< 6 > element_t;
Mesh< element_t > mesh( input );
// declare two Lagrangian fe-bases (constant and linear)
typedef FEBasis< element_t, CONSTANT,
                 Discontinuous, LagrangeTraits > test_feb_t;
typedef FEBasis< element_t, LINEAR,
                 Continuous   , LagrangeTraits > trial_feb_t;
// based on the fe-basis declare the dofhandler types
typedef DoFHandler< test_feb_t > test_dh_t;
typedef DoFHandler< trial_feb_t > trial_dh_t;
// instantiate the dofhandler objects
test_dh_t  test_dh
trial_dh_t trial_dh;
// span the discrete fe spaces
test_dh.distributeDoFs( mesh.e_begin(), mesh.e_end() );
trial_dh.distributeDoFs( mesh.e_begin(), mesh.e_end() );
/* declare fundamental solution,
   kernel, quadrature, and integrator */
typedef FundSol< LAPLACE, DLP > fs_t;
typedef GalerkinKernel< fs_t,
                        test_feb_t, trial_feb_t > kernel_t;
typedef GalerkinQuadrature< element_t, 7, 36, 25, 16 > quad_t;
typedef GalerkinIntegrator< kernel_t, quad_t > integrator_t;
// ...and instances
fs_t         fs;
kernel_t     kernel( fs );
integrator_t integrator( kernel );
// declare and instantiate the bem operator
typedef BemOperator< integrator_t,
                     test_dh_t, trial_dh_t > bem_op_t;
bem_op_t bem_op( integrator, test_dh, trial_dh );
// this computes the discrete double layer operator
bem_op.compute( );
// this returns a reference to the computed matrix structure
bem_op_t::const_reference K_h = bem_op.giveMatrix( );
```

Fig. 7. An example program for the discretization of the double layer operator for the Laplace equation (see Eqn. (13))

10–17: The dofhandler encodes the mapping of local basis functions to global basis functions. This mapping is furnished by the `distributeDoFs` method. Here, we distribute the degrees of freedom over the complete mesh such that the respective dofhandler is called with the mesh's begin and end element iterators. The distribution of the degrees of freedom already finalizes the setup of the Galerkin scheme.

20–24: It remains to set up the BEM model. In doing so we will start with the type definitions for the fundamental solution $\partial G_L/\partial \mathbf{n}(\mathbf{y})$. Together with the test- and trial-spaces the fundamental solution's type is then used to declare the kernel. Afterwards the quadrature rules are specified. Beside the element type this structure takes four integer values for the regular (7), coincident (36), edge adjacent (25), and vertex adjacent (16) integrations. Regular integrations on triangular elements are based quadrature schemes from [Dunavant 1985]. All remaining quadrature schemes, i.e., all singular integrations as well as integrations on quadrilateral elements are based standard tensor Gauss-Legendre

rules. Note that the quadrature rule above is a special case of the more general quadrature rule

```
typedef
GalerkinQuadrature< element_t, 7, 36, 25, 16,
                    element_t, 1, 36, 25, 16 > quad_t;
```

where the first four template parameters define the quadrature rules for the inner element $\tau_\mathbf{y}$ while the remaining parameters define the rules for the outer element $\tau_\mathbf{x}$. If the specification for the outer integration is omitted it will be defined implicitly such that it matches the quadrature rule for the inner integration. Once the quadrature type has been declared it is used together with the kernel type for the declaration of the integrator type.

26–28: It remains to create instances of the fundamental solution, of the kernel, and of the integrator. Thereby, the previous instance serves as a constructor argument for the subsequent instance.

30–36: Everything is in place and we can proceed with the final step—the creation of the discrete BEM operator. For this the dofhandler objects are glued together with the integrator. This is achieved by the declaration and instantiation of the `BemOperator` structure. A simple `compute()`-call creates the discrete operator and the final command returns a reference to the underlying matrix structure. In Sec. 3.3 and 3.4 we will comment in more detail on how BETL handles matrices.

The above code example serves as a pattern which may be adjusted for the definition of other discrete boundary integral operators. In what follows we will give some hints and suggestions on how the above code might be modified.

— Change of geometry: Switch to flat triangles

```
typedef Element< 3 > element_t; // simply exchange 6 -> 3
```

— Change of basis: $Z_h^0 \to Z_h^2$, $Y_h^1 \to Y_h^3$

```
typedef FEBasis< element_t, QUADRATIC, // instead of CONSTANT
                 Discontinuous, LagrangeTraits > test_feb_t;
typedef FEBasis< element_t, CUBIC,     // instead of LINEAR
                 Continuous   , LagrangeTraits > trial_feb_t;
```

— Choose a Helmholtz fundamental solution:

```
typedef FundSol< HELMHOLTZ, DLP > fs_t;
// ...
fs_t fs( kappa ); // kappa: complex or real valued wavenumber
```

— Switch to the Maxwell system. This requires two changes. First, edge-based test-trial-spaces need to be declared, and secondly, we need to define the fundamental solution from Eqn. (15) *(Nonetheless, the edge bases work also in conjunction with the Helmholtz and Laplace fundamental solutions)*.

```
// Raviart-Thomas basis
typedef FEBasis< element_t, LINEAR,
                 Continuous, EdgeDivTraits > test_feb_t;
// Rotated Raviart-Thomas basis
typedef FEBasis< element_t, LINEAR,
                 Continuous, EdgeCurlTraits > trial_feb_t;
// ...
typedef FundSol< MAXWELL, DLP > fs_t;
// ...
fs_t fs( kappa ); // kappa: complex or real valued wavenumber
```

BETL's modularity allows also for an easy exchange of computational routines like, e.g., the integrator classes. In Sec. 3.1 it has been already mentioned that the default integrator can be replaced by another—hopefully more sophisticated—integrator. As an example we will replace the general integration structure from Fig. 7 by a more specialized integration routine. In contrast to the former `Integrator` structure the type definition and instantiation of a semi-analytic integration scheme based on the formulas by Rjasanow and Steinbach [2007] is

```
const unsigned num_gp = 7;
typedef GalerkinSemiAnalyticIntegrator< kernel_t,
                                        num_gp > integrator_t;
// create an instance of the integrator
// (but without an instance of the kernel!)
integrator_t integrator;
```

Note that the instantiation of the integrator is not performed via an instance of the kernel. A kernel object is not needed since the inner integration is taken out exact. Only the kernel's type information is needed in order to query whether the integrator's definition allows for an analytic inner integration. Currently, analytic integration formulas exist only for the single- and double-layer operators for the Laplace equation with lowest order test- and trial-spaces for flat triangular elements. All these information are queried via the kernel's type. If only a single type therein does not match the requirements for this special quadrature the compilation of the code will abort.

In BETL the implementation of an integrator is done via the concept of functors [Alexandrescu 2004]. Functors are data types which implement the parenthesis operator `()`. The main difference between a functor and a classical function is that functors may hold a distinct state. For instance, in BETL an integrator typically administrates quadrature related information. However, this particular data is private. The only public interface an integrator type has to provide is a method with the following signature

```
void operator()( const ex_t* element_x,
                 const ey_t* element_y,
                 result_t& result ) const;
```

Above, `ex_t` and `ey_t` denote two *(possibly different)* element types and `result_t` is the type of the integration result. Once a class with this interface and these type declarations has been implemented it can be immediately used for integration purposes. Of course the integrator must be provided with some information about the kernel, but the way this information is prescribed is completely left to the user. This can be seen by the code examples above. The generic integrator gets a kernel object during its instantiation while the semi-analytic integrator creates a sort of kernel object *(involving an anti-derivative)* on its own.

At this point the power of generic programming techniques really pays off. The implementation of new integration schemes can be easily done just by the definition of the above interface. Then, thanks to the template mechanism, these structures can be directly plugged into the code without embedding them into some object-oriented hierarchy. Moreover, within a numerical computation the integrator is easily called a million of times. Thus, a naive object-oriented approach on that level might result in millions of virtual function calls which affects the overall runtime considerably.

3.3. Incorporation of Fast Boundary Element Methods

Boundary integral operators are inherently non-local, and, thus, their Galerkin discretization will invariably lead to dense matrices despite the use of locally supported basis functions. Thus, a straightforward implementation will incur prohibitive $O(N^2)$ cost in terms of computational effort as well as in terms of storage requirements, with

N being the number of degrees of freedom. A remedy is offered by the family of so-called *Fast Boundary Element Methods* which apply matrix compression techniques to reduce the method's overall complexity to an order of $O(N \log N)$, or even to an order of $O(N)$.

The family of Fast Boundary Element Methods comprises several different acceleration techniques like, e.g., Fast Multipole Methods (FMM) [Greengard and Rokhlin 1987; Rokhlin 1985], Wavelet-based BEM methods [Alpert et al. 1993], Panel Clustering [Hackbusch and Nowak 1989], and the Adaptive Cross Approximation (ACA) [Bebendorf and Rjasanow 2003]. Nowadays, the most popular fast techniques are the FMM and the ACA. A detailed description of those methods is far beyond the scope of this work and for a comprehensive overview on those methods the reader is referred to [Nishimura 2002] and [Bebendorf 2008] and the references cited therein. BETL supports different FMM schemes as well as ACA. However, only the interfaces to ACA are well-matured such that we will solely focus on this method in the following. More precisely, we will discuss the implications an incorporation of this method has from an implementation point of view.

The main ingredients to the ACA are twofold: Based on geometric criteria the degrees of freedom are partitioned into groups of hierarchical organized clusters. Once this hierarchy has been established, the second step consists of an approximation of the non-local kernel for admissible pairs of clusters. This kernel approximation relies on a purely algebraic approach. The advantage of algebraic based methods is that they put less conditions onto the involved integral kernels. This fact makes their use attractive within a general purpose BEM library like BETL. In addition, there exists a software library called AHMED [Bebendorf 2012] which implements the ACA as well as its native matrix format—the so-called $\mathcal{H}$-matrix format [Bebendorf 2008; Hackbusch 2009]. Within BETL the ACA can be easily activated in conjunction with AHMED. An example will be given in Sec. 4.5.

The use of ACA demands the creation of a cluster tree. Since this cluster tree is created with respect to the already initialized degrees of freedom its construction is left to the `DoFHandler` instances. However, the particular cluster algorithms may vary. For this reason the implementation of the cluster methods is based on policies [Alexandrescu 2004; Vandervoorde and Josuttis 2002]. Policies are intended to enrich data types by certain methods.

In order to augment the `DoFHandler` class by features related to cluster techniques we define skeletons of the `ClusterPolicy` structures as follows

```
// declare an empty policy
template< enum ACCELERATION ACC >
struct ClusterPolicy {
  /* this empty struct is the default policy */
};

// define a specialization for ACA
template< >
struct ClusterPolicy< ACA > {
public:
  // clusterize the degrees of freedom
  void clusterize( /* args */ );
private:
  cluster_tree_t cluster_tree_;
};
```

Now, the `DoFHandler` utilizes one of the above classes via inheritance. This yields

```
template< typename FEBASIS_T,
```

```
             enum ACCELERATION ACC = NO_ACCELERATION >
class DoFHandler : public ClusterPolicy< ACC > {

  // specific dofhandler implementation

};
```

Within the main program we can create two different types of DoFHandler objects, one that represents the default type and a second DoFHandler type which enhances the default type by a special method

```
// ...
// the default dofhandler
DoFHandler< fe_basis_t > dofhandler;
// here, the 'clusterize' method is available
DoFHandler< fe_basis_t, ACA > dofhandler_aca;
// ...
```

The advantage of an implementation approach via policies consists in the facts that, on one hand, we do not mess up the implementation with routines that are beyond the typical DoFHandler tasks and, secondly, we can easily define a bunch of different ClusterPolicy structures the DoFHandler type may rely on.

The BemOperator is designed similarly to the DoFHandler class, i.e., it implements certain structures also on the basis of policies. For instance, the matrix format differs when one deals with Fast Boundary Element Methods. In case of ACA the BEM-matrices are stored by using the $\mathcal{H}$-matrix format which defines a sparse representation of originally dense matrices. Contrary, in case of using a FMM there exists no assembled matrix at all. The implementation of the BemOperator needs to reflect these different situations. Therefore, the BemOperator is inherited from an AccelerationPolicy which defines the underlying matrix format in the following way

```
// by default the BemOperator will create dense matrices
template< enum ACCELERATION ACC >
struct AccelerationPolicy {
  typedef const dense_matrix_t& const_reference;
};
// derive the BemOperator from this policy in case of ACA
template< >
struct AccelerationPolicy< ACA > {
  typedef const H_matrix_t& const_reference;
};
```

Since Fast Boundary Element Methods define quite exotic matrix formats BETL deals with matrices in a very generic manner. Every data type which defines a numeric type and which implements a matrix-vector product as well as routines for returning the number of rows and the number of columns *is* a matrix type. Once the BemOperator has computed the necessary data one can access the matrix via the following line of code

```
bem_op_t::const_reference A_h = bem_op.giveMatrix( );
```

Above, the BemOperator defines the particular matrix format by its respective AccelerationPolicy. Commonly, this is chosen automatically via a query of the DoFHandler's acceleration-related data types. Note, that the 'matrix' A_h can be used instantaneous within other routines like, e.g., iterative solvers.

Remark: Beside the ACA-support BETL features also implementations of FMMs. BETL incorporates implementations of the Fast Multipole Method as it has been pro-

posed in [Of 2006; Of et al. 2006] as well as a Fast Multipole Method which is well-suited for the treatment of Helmholtz-like kernels [Messner et al. 2012]. However, Fast Multipole Methods do not feature the *black blox* structure that the Adaptive Cross Approximation does. At this time *(Nov. 2012)*, BETL's incorporation of FMMs is at an experimental stage. A seamless integration of FMMs is still to come.

3.4. Sparse operators

We will close this section with a very short discussion on sparse structures. In BETL, as a convention we use the name `Operator` to denote that a data type computes something which behaves like a matrix but it is important to note that the `Operator` should not be confused with a matrix. Operator structures always process some information which are related to the discretization scheme. For this reason they are always defined by help of the `DoFHandler` types. Contrary, a `Matrix` structure represents a mathematical object which can be used in the context of linear algebra.

For sure, the most important `Operator` structure is the `BemOperator`. However, even with boundary element methods one is forced to create sparse matrices. The use of sparse operators is demonstrated in Sec. 4.3 and in Sec. 4.5. Here, we will introduce sparse operators by means of the discrete identity operator which is nothing but a mass matrix. This matrix allows for a classical sparse representation via the CCS *(aka Harwell-Boeing)* format [Duff et al. 1989]. The CCS format is widely used and BETL utilizes also the well-known open source library SuiteSparse [Davis 2012] for sparse matrix computations. The construction of the mass matrix is done very similar to that of the `BemOperator` structure

```
typedef IdentityOperator< test_dh_t , trial_dh_t > id_op_t;
id_op_t id_op( test_dh , trial_dh );
id_op.compute( );
id_op_t::const_reference M_h = id_op.giveMatrix( );
```

As before, the implementation of sparse operators is done in a generic way. Every sparse operator structure is derived from a structure called `RootSparseOperator` which implements the assembly of the sparse matrix. In doing so it needs a functor to be passed from the derived class to the generic assembly routine. For instance, the `IdentityOperator` implements a functor which computes the mass matrices on an element level. Then this functor is passed to the `RootSparseOperator`. The following code snippet illustrates this approach

```
template< typename TESTDOFHANDLER_T ,
          typename TRIALDOFHANDLER_T >
class IdentityOperator :
  public RootSparseOperator< TESTDOFHANDLER_T ,
                             TRIALDOFHANDLER_T >
{
private:
  typedef RootSparseOperator< TESTDOFHANDLER_T ,
                              TRIALDOFHANDLER_T > rso_t;

public:
  // the constructor
  IdentityOperator( TESTDOFHANDLER_T& test_dh ,
                    TRIALDOFHANDLER_T& trial_dh )
    : rso_t( test_dh , trial_dh ) { /* ... */ }

  // the public compute method
  void compute( )
```

```
    {
      element_massmatrix_t element_massmatrix;
      this -> rso_t::compute_( element_massmatrix );
    }

    // further implementations...
};
```

This methodology is applied to all sparse structures. Again, the main idea is that particular sparse operators implement functors for the construction of particular element matrices and that these functors are passed to the generic sparse operator. In the following section, we will give some other examples where the use of sparse structures is advantageous.

4. EXAMPLES

All validation an example codes which have been used for the following computations are included in BETL. They are also publicly available via BETL's homepage

http://www.sam.math.ethz.ch/betl/

4.1. Convergence tests for various FE-spaces and fundamental solutions

We will verify BETL's implementation by means of the Calerdón identity (2). Its first equation is

$$\langle V\gamma_1 u, \varphi\rangle_\Gamma - \langle\left(\tfrac{1}{2}\,\mathrm{Id} + K\right)\gamma_0 u, \varphi\rangle_\Gamma = 0\,. \tag{19}$$

In the following we will perform convergence studies by utilizing the above equation with some prescribed Cauchy data $\{\gamma_0 u, \gamma_1 u\}$. In terms of a Galerkin discretization (19) is equivalent to the linear system

$$V_h \underline{u}_N - \left(\tfrac{1}{2}M_h + K_h\right)\underline{u}_D =: \underline{R}$$

with the discrete single layer operator V_h, the discrete double layer operator K_h, and the discrete identity operator M_h. The vectors $\{\underline{u}_D, \underline{u}_N\}$ contain the coefficients of the interpolated Cauchy data. The vector $\underline{R}$ is the residuum whose behavior will be investigated for a decreasing global mesh size h.

The verification of the Laplace, Helmholtz, and Maxwell operators will be taken out on the unit cube $\overline{\Omega} := [0,1]^3$ with uniform discretisations, i.e., $h = h_i$ for every element τ_i. The quantity h_i denotes the local mesh size

$$h_i := \max_{\mathbf{x}\in\tau_i} |\mathbf{x} - \bar{\mathbf{x}}_{\tau_i}|$$

with $\bar{\mathbf{x}}_{\tau_i}$ being the barycenter of τ_i. The elements τ_i are assumed to be either flat triangular elements, or 4-noded quadrilateral elements. We will perform all tests by using the generic integrator based on the formulas of Sauter and Schwab [2011]. As long as not mentioned otherwise regular integrations are taken out with 21 Gaussian points for triangles, and with 25 Gaussian points for quadrilaterals, respectively. The singular integrations are performed with 49 Gaussian points. We use this fairly high number of Gaussian points in order to minimize the effects of quadrature errors.

Laplace operator. The fundamental solution $G_L(\mathbf{z})$ represents an analytic solution to the Laplace equation for all $\mathbf{z} \neq \mathbf{0}$. Therefore, the tests for the Laplace operator are done with the artificial Cauchy data

$$\gamma_{0,\mathbf{x}} u(\mathbf{x}) := \gamma_{0,\mathbf{x}} G_L(\mathbf{x} - \mathbf{x}^*)\,, \quad \gamma_{1,\mathbf{x}} u(\mathbf{x}) := \gamma_{1,\mathbf{x}} G_L(\mathbf{x} - \mathbf{x}^*) \quad \mathbf{x} \in \Gamma$$

where the source point $\mathbf{x}^*$ is given by $\mathbf{x}^* := [1.6, 0.6, 1.5]^\top \notin \Gamma$.

According to the appendix (Eqn. (37)) the residuum vector $\underline{R}$ satisfies the estimate

$$|\underline{R}(\psi_h^p)| \le Ch^{p+2} \tag{20}$$

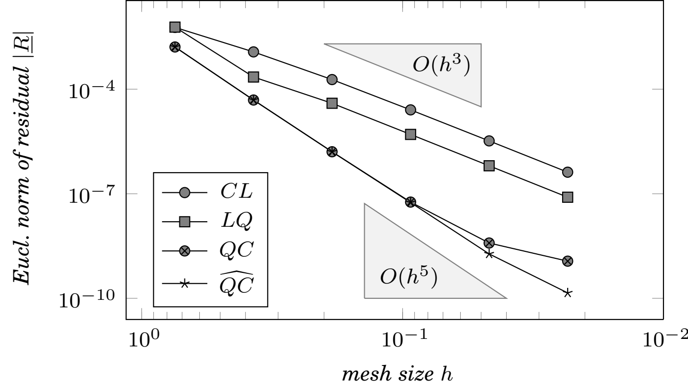

Fig. 8. Laplace operator. Triangular element type. CL: constant approximation for Neumann data, linear approximation of Dirichlet data. LQ: Linear/quadratic approximation of Neumann and Dirichlet data. QC: Quadratic/cubic approximation of Neumann and Dirichlet data. $\widehat{QC}$: Quadratic/cubic approximation but with an increased number of Gaussian points 42 (regular), 64 (singular).

The Fig. 8 and the Tab. I show the results for $|\underline{R}|$ in case of triangular elements. The corresponding results for quadrilateral elements are summarized in Fig. 9 and in Tab. II. In principle both studies convey the same conclusions. For constant test-spaces and linear trial-spaces the norm of the residual is of order $O(h^3)$ although one would expect only an order of $O(h^2)$. A similar super-convergence can be observed for discontinuous quadratic test-spaces and continuous cubic trial-spaces. Again the observed order of $O(h^5)$ is higher than the estimate $O(h^4)$. Since the estimate (20) represents only an upper bound, the occurring convergence rates do not contradict the theoretical prediction. And finally, for discontinuous linear test-spaces and continuous quadratic trial-spaces the observed convergence rates meet the theoretical estimate of order $O(h^3)$.

Table I. Laplace operator. Triangular element type. Abbreviations from Fig. 8

ℓ	h	$\lvert\underline{R}\rvert_{CL}$	$\frac{\lvert\underline{R}\rvert_{CL}^{(\ell-1)}}{\lvert\underline{R}\rvert_{CL}^{(\ell)}}$	$\lvert\underline{R}\rvert_{LQ}$	$\frac{\lvert\underline{R}\rvert_{LQ}^{(\ell-1)}}{\lvert\underline{R}\rvert_{LQ}^{(\ell)}}$	$\lvert\underline{R}\rvert_{QC}$	$\frac{\lvert\underline{R}\rvert_{QC}^{(\ell-1)}}{\lvert\underline{R}\rvert_{QC}^{(\ell)}}$	$\lvert\underline{R}\rvert_{\widehat{QC}}$	$\frac{\lvert\underline{R}\rvert_{\widehat{QC}}^{(\ell-1)}}{\lvert\underline{R}\rvert_{\widehat{QC}}^{(\ell)}}$
0	0.745	5.94_{-3}	—	6.04_{-3}	—	1.63_{-3}	—	1.63_{-3}	—
1	0.373	1.18_{-3}	5.0	2.24_{-4}	27.0	4.89_{-5}	33.3	4.89_{-5}	33.3
2	0.186	1.89_{-4}	6.2	3.95_{-5}	5.7	1.60_{-6}	30.5	1.60_{-6}	30.6
3	0.093	2.55_{-5}	7.4	5.04_{-6}	7.8	5.77_{-8}	27.8	5.60_{-8}	28.5
4	0.047	3.28_{-6}	7.8	6.35_{-7}	7.9	3.86_{-9}	14.9	1.88_{-9}	29.8
5	0.023	4.16_{-7}	7.9	7.96_{-8}	8.0	1.16_{-9}	3.3	1.43_{-10}	13.2

In case of a quadratic/cubic approximation we notice a collapse of convergence for triangular elements (cf. Fig. 8, Tab. I). This is due to quadrature errors since a computation with an increased number of Gaussian points (the $\widehat{QC}$ data set) sustains the convergence rate.

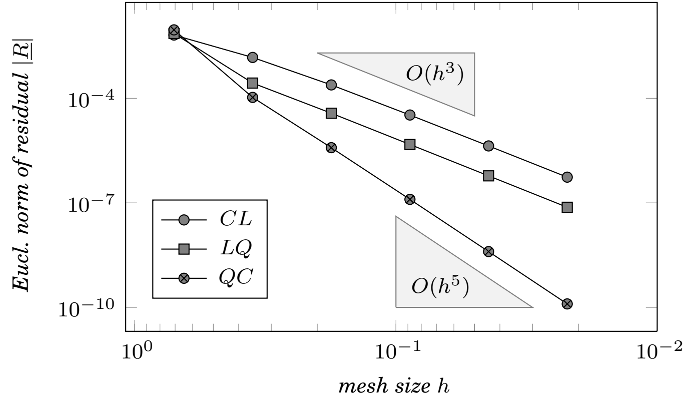

Fig. 9. Laplace operator. Quadrilateral element type. CL: constant approximation for Neumann data, linear approximation of Dirichlet data. LQ: Linear/quadratic approximation of Neumann and Dirichlet data. QC: Quadratic/cubic approximation of Neumann and Dirichlet data.

Table II. Laplace operator. Quadrilateral element type. Abbreviations from Fig. 9

ℓ	h	$\lvert\underline{R}\rvert_{CL}$	$\frac{\lvert\underline{R}\rvert_{CL}^{(\ell-1)}}{\lvert\underline{R}\rvert_{CL}^{(\ell)}}$	$\lvert\underline{R}\rvert_{LQ}$	$\frac{\lvert\underline{R}\rvert_{LQ}^{(\ell-1)}}{\lvert\underline{R}\rvert_{LQ}^{(\ell)}}$	$\lvert\underline{R}\rvert_{QC}$	$\frac{\lvert\underline{R}\rvert_{QC}^{(\ell-1)}}{\lvert\underline{R}\rvert_{QC}^{(\ell)}}$
0	0.707	6.53_{-3}	—	7.25_{-3}	—	9.19_{-3}	—
1	0.354	1.49_{-3}	4.4	2.74_{-4}	26.5	1.06_{-4}	86.7
2	0.177	2.42_{-4}	6.1	3.73_{-5}	7.3	3.83_{-6}	27.7
3	0.088	3.31_{-5}	7.3	4.73_{-6}	7.9	1.26_{-7}	30.4
4	0.044	4.30_{-6}	7.7	5.96_{-7}	7.9	4.00_{-9}	31.5
5	0.022	5.46_{-7}	7.9	7.47_{-8}	8.0	1.26_{-10}	31.7

Helmholtz equation. Analytic solutions for the Helmholtz equation are, e.g., provided by plane waves

$$U_{\kappa,\mathbf{d}}(\mathbf{z}) := \exp(\kappa\,\mathbf{z}\cdot\mathbf{d})\,, \quad \kappa\in\mathbb{C},\ \|\mathbf{d}\| = 1$$

where the vector $\mathbf{d}$ denotes the direction of propagation. The Cauchy data we are using for the upcoming convergence studies are

$$\gamma_{0,\mathbf{x}}u(\mathbf{x}) := \gamma_{0,\mathbf{x}}U_{\kappa^*,\mathbf{d}^*}(\mathbf{x})\,, \quad \gamma_{1,\mathbf{x}}u(\mathbf{x}) := \gamma_{1,\mathbf{x}}U_{\kappa^*,\mathbf{d}^*}(\mathbf{x})$$

with the parameters $\kappa^* = 1\mathrm{i}$ and $\mathbf{d}^* = \sqrt{\frac{10}{39}}\,[1.0, 1.1, 1.3]^\top$. Not surprisingly, the results in case of the Helmholtz operators continue the observations we have already made for the Laplace operators. For a combination of constant/linear as well as quadratic/cubic approximations we notice super-convergence while the linear/quadratic combination fits into the theoretical forecast. Moreover, we observe the same breakdown in convergence as we did for the Laplace equation. And also the reason for this breakdown is the same as before since a computation with an increased number of Gaussian points stabilizes the convergence rates.

Maxwell system. Currently, BETL supports only lowest order approximations of the edge-based test- and trial-spaces $\mathbf{Z}_{\parallel,h}(\mathrm{div}_\Gamma,\Gamma)$ and $\mathbf{Y}_{\perp,h}(\mathrm{curl}_\Gamma,\Gamma)$, respectively. For these lowest order spaces the residual satisfies (Eqn. (38))

$$|\underline{R}(\boldsymbol{\varphi}_h)| \le Ch^0 \tag{21}$$

and, therefore, keeps constant during refinement.

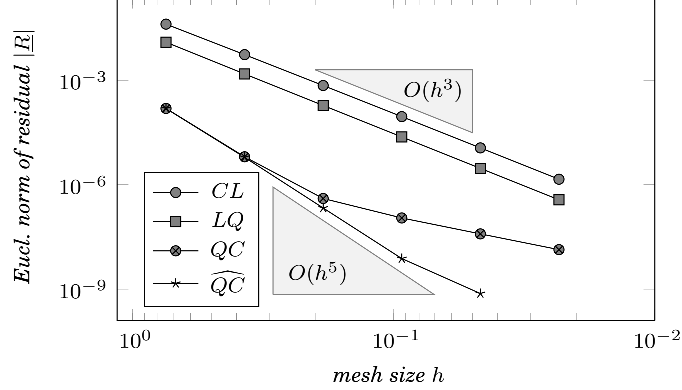

Fig. 10. Helmholtz operator. Triangular element type. CL: constant approximation for Neumann data, linear approximation of Dirichlet data. LQ: Linear/quadratic approximation of Neumann and Dirichlet data. QC: Quadratic/cubic approximation of Neumann and Dirichlet data. $\widehat{QC}$: Quadratic/cubic approximation but with an increased number of Gaussian points 42 (regular), 81 (singular).

Table III. Helmholtz operator. Triangular element type. Abbreviations from Fig. 10

ℓ	h	$\lvert\underline{R}\rvert_{CL}$	$\frac{\lvert\underline{R}\rvert_{CL}^{(\ell-1)}}{\lvert\underline{R}\rvert_{CL}^{(\ell)}}$	$\lvert\underline{R}\rvert_{LQ}$	$\frac{\lvert\underline{R}\rvert_{LQ}^{(\ell-1)}}{\lvert\underline{R}\rvert_{LQ}^{(\ell)}}$	$\lvert\underline{R}\rvert_{QC}$	$\frac{\lvert\underline{R}\rvert_{QC}^{(\ell-1)}}{\lvert\underline{R}\rvert_{QC}^{(\ell)}}$	$\lvert\underline{R}\rvert_{\widehat{QC}}$	$\frac{\lvert\underline{R}\rvert_{\widehat{QC}}^{(\ell-1)}}{\lvert\underline{R}\rvert_{\widehat{QC}}^{(\ell)}}$
0	0.745	4.09_{-2}	0.0	1.24_{-2}	0.0	1.55_{-4}	0.0	1.54_{-4}	—
1	0.373	5.44_{-3}	7.5	1.52_{-3}	8.2	6.33_{-6}	24.5	6.11_{-6}	25.3
2	0.186	7.07_{-4}	7.7	1.89_{-4}	8.0	4.01_{-7}	15.8	2.14_{-7}	28.6
3	0.093	9.02_{-5}	7.8	2.36_{-5}	8.0	1.11_{-7}	3.6	7.55_{-9}	28.3
4	0.047	1.14_{-5}	7.9	2.94_{-6}	8.0	3.86_{-8}	2.9	7.47_{-10}	10.1
5	0.023	1.43_{-6}	7.9	3.68_{-7}	8.0	1.36_{-8}	2.8	—	—

As for the Helmholtz equation we utilize a plane wave solution

$$\mathbf{U}_{\kappa,\mathbf{d},\mathbf{p}}(\mathbf{z}) := (\mathbf{d}\times(\mathbf{p}\times\mathbf{d}))\exp(\kappa\,\mathbf{z}\cdot\mathbf{d}), \quad \kappa\in\mathbb{C},\ \|\mathbf{d}\|=1$$

to construct some Cauchy data. With the direction $\mathbf{d}^* = \sqrt{\frac{10}{39}}\,[1.0, 1.1, 1.3]^\top$, the polarization $\mathbf{p}^* = [0,1,0]^\top$, and the wave number $\kappa^* = 1\mathrm{i}$ we define the Cauchy data as

$$\gamma_{0,\mathbf{x}}\mathbf{u}(\mathbf{x}) := \gamma_{0,\mathbf{x}}\mathbf{U}_{\kappa^*,\mathbf{d}^*,\mathbf{p}^*}(\mathbf{x}), \quad \gamma_{1,\mathbf{x}}\mathbf{u}(\mathbf{x}) := \gamma_{1,\mathbf{x}}\mathbf{U}_{\kappa^*,\mathbf{d}^*,\mathbf{p}^*}(\mathbf{x}).$$

The results of the convergence study are given in Tab. IV. Clearly, this study confirms the theoretical predictions (21).

Table IV. Maxwell operator

ℓ	h	$\lvert\underline{R}\rvert$	$\frac{\lvert\underline{R}\rvert^{(\ell-1)}}{\lvert\underline{R}\rvert^{(\ell)}}$
0	0.745	1.252_0	—
1	0.373	1.414_0	0.886
2	0.186	1.511_0	0.936
3	0.093	1.558_0	0.969
4	0.047	1.582_0	0.985
5	0.023	1.596_0	0.992

4.2. Higher order geometry approximations

We will investigate the effect of various geometry approximation by means of the following exterior model problem

$$\begin{aligned} -\Delta u &= 0 \qquad \text{in } \Omega^+ \\ \gamma_0^+ u &= 1 \qquad \text{on } \Gamma \\ \lim_{|\mathbf{x}|\to\infty} |\mathbf{x}|\, u &= 0\,. \end{aligned}$$

Above, $\Omega^+ := \{\mathbf{x} \in \mathbb{R}^3 \colon |\mathbf{x}| > \frac{1}{2}\}$ denotes the domain outside of a sphere centered at the origin. The boundary is $\Gamma := \{\mathbf{x} \in \mathbb{R}^3 \colon |\mathbf{x}| = \frac{1}{2}\}$. An analytic solution for the Neumann datum is then given by $\gamma_1^+ u = 2$. The discrete boundary integral operator equation for the above problem reads

$$V_h \underline{t} = M_h \underline{1}\,.$$

Since the exact Cauchy data is constant it suffices to discretize both operators V_h and M_h with lowest order test- and trial spaces $\varphi^0 \in Z_h^0(\Gamma)$. The solution $\underline{t}$ is obtained via a direct solver.

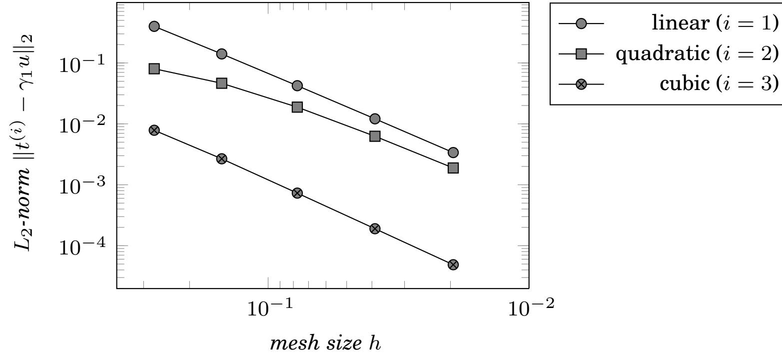

Fig. 11. L_2-error norms $\|t_h^{(i)} - \gamma_1 u\|_2$. $i = 1, 2, 3$ represents linear, quadratic, and cubic shape approximations.

The Fig. 11 and the Tab. V depict the computational results. Clearly, switching from one geometric approximation to another does not affect the solution's convergence rate. However, the overall error changes dramatically with respect to the geometry approximation order. While for this example there seems to be little benefit in using quadratic boundary elements compared to linear elements the use of cubic elements leads to a significant improvement. Already, the first refinement level yields results which are of the same order as those of the linear and quadratic elements on the finest grid.

Table V. L_2-error norms of $\epsilon^{(i)} := t_h^{(i)} - \gamma_1 u$, $i = 1, 2, 3$. Linear (1), quadratic (2), and cubic (3) geometry approximations. The $\{\min, \max\}$-pairs denote the minimal and maximal values in the solution vector $\underline{t}^{(i)}$.

h	$\|\epsilon^{(1)}\|_2$	$\min(\underline{t}^{(1)})$	$\max(\underline{t}^{(1)})$	$\|\epsilon^{(2)}\|_2$	$\min(\underline{t}^{(2)})$	$\max(\underline{t}^{(2)})$	$\|\epsilon^{(3)}\|_2$	$\min(\underline{t}^{(3)})$	$\max(\underline{t}^{(3)})$
0.27	3.99_{-1}	1.892	2.292	8.03_{-2}	1.968	2.098	7.85_{-3}	1.988	2.003
0.15	1.41_{-1}	1.925	2.159	4.64_{-2}	1.942	2.047	2.67_{-3}	1.996	2.003
0.08	4.23_{-2}	1.960	2.063	1.88_{-2}	1.962	2.031	7.30_{-4}	1.998	2.002
0.04	1.21_{-2}	1.977	2.026	6.26_{-3}	1.979	2.017	1.90_{-4}	1.999	2.001
0.02	3.38_{-3}	1.988	2.012	1.89_{-3}	1.989	2.008	4.87_{-5}	1.999	2.000

4.3. Adaptive Cross Approximation

Equivalent to Sec. 4.1, we will perform the ACA tests[1] based on Eqn. (19) which will be evaluated by means of the Laplace operator. A careless use of ACA has its pitfalls and we will present some ideas on how they can be safely circumvented.

As before, some artificial Cauchy data $\{\gamma_0 u, \gamma_1 u\}$ is prescribed on basis of the fundamental solution G_L. The domain $\Omega := \{\mathbf{x} \in \mathbb{R}^3 : x_1^2 + 4(x_2^2 + x_3^2) < 1\}$ is chosen to be an ellipsoid. Fig. 12 illustrates the setup.

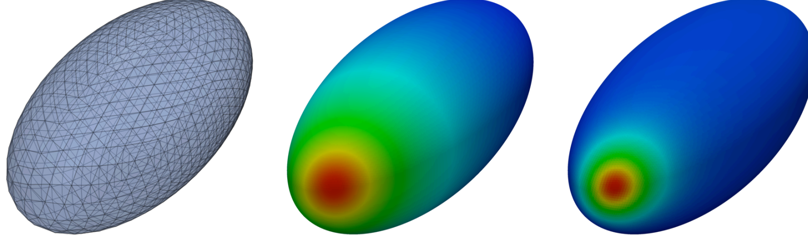

Fig. 12. Left: A discretization of the ellipsoid with 2048 flat triangular elements. Center: The prescribed Dirichlet datum $\gamma_0 u(\mathbf{x}) = \gamma_0 G_L(\mathbf{x} - \mathbf{x}^*)$. Right: The prescribed Neumann datum $\gamma_1 u(\mathbf{x}) = \gamma_1 G_L(\mathbf{x} - \mathbf{x}^*)$. The source point is $\mathbf{x}^* = [1.1, 0.2, 0.3]^\top$.

The results[2] in Tab. VI and Tab. VII are based on lowest order discretisations of Eqn. (19), i.e., the corresponding discrete spaces are $(\psi_h^0, \varphi_h^1) \in Z_h^0(\Gamma) \times Y_h^1(\Gamma)$. For the numerical integration we impose the set

$$Q := \{7, 36, 25, 16\} \tag{22}$$

of quadrature rules for the regular, coincident, edge-adjacent, and vertex-adjacent integrations, respectively.

The last columns of Tab. VI summarizes the residual norms $|\underline{R}_Q|$ which, again, reveal a super-convergence of order $O(h^3)$ (cf. Sec. 4.1). Further, one can detect an almost linear growth in storage requirements as well as in computational time with each refinement level. These observations confirm the theoretical predictions made by the ACA [Bebendorf 2008].

Table VI. ACA approximations $V_{\mathcal{H}} \approx V_h : (\psi_h^0, \psi_h^0) \in Z_h^0(\Gamma) \times Z_h^0(\Gamma)$ of the single layer operator. The compression rate η is the ratio of the size of the $\mathcal{H}$-matrix and that of the fully assembled non-symmetric matrix. The memory storage is given in megabytes. T_Q represents the computational time in seconds. All computations are based on the quadrature rule from Eqn. (22).

ℓ	$\dim(V_{\text{full}})$	$\eta := \frac{\text{mem}(V_{\mathcal{H}})}{\text{mem}(V_{\text{full}})}$	T_Q [s]	$\text{mem}(V_{\mathcal{H}})$ [MB]	$\frac{T^{(\ell)}}{T^{(\ell-1)}}$	$\frac{\text{mem}(V_{\mathcal{H}})^{(\ell)}}{\text{mem}(V_{\mathcal{H}})^{(\ell-1)}}$	$\lvert\underline{R}_Q\rvert$
0	$2,048^2$	0.3660	6	12	—	—	1.54_{-5}
1	$8,192^2$	0.1236	36	63	5.52	5.40	1.82_{-6}
2	$32,768^2$	0.0406	188	332	5.26	5.25	2.17_{-7}
3	$131,072^2$	0.0128	957	1,677	5.10	5.05	2.65_{-8}
4	$524,288^2$	0.0039	4,808	8,131	5.02	4.85	3.33_{-9}

[1] All computations have been done with the following ACA settings: ACA accuracy: $\varepsilon_{\text{ACA}} = 1 \cdot 10^{-5}$, Admissibility condition: $\eta_{\text{ACA}} = 0.5$, Minimum cluster size: $s_{\min} = 30$, Maximum rank: $r_{\max} = 200$. Refer to [Bebendorf 2008] for more information on these control parameters.

[2] All ACA computations are based on the following specifications: *Hardware:* AMD Opteron 6174, 2.20GHz, 128 GB RAM, L2-Cache 512kB. *Software:* Intel Compiler v13.0, Compiler flags: `-O3`, single-threaded, Boost v1.42, Linkage with Intel's Math Kernel Library (MKL) v11.0.

Table VII. ACA approximations $K_{\mathcal{H}} \approx K_h\colon (\psi_h^0, \varphi_h^1) \in Z_h^0(\Gamma) \times Y_h^1(\Gamma)$ of the double layer operator.

ℓ	$\dim(K_h)$	$\eta := \frac{\text{mem}(K_{\mathcal{H}})}{\text{mem}(K_h)}$	T_Q [s]	$\text{mem}(K_{\mathcal{H}})$ [MB]	$\frac{T^{(\ell)}}{T^{(\ell-1)}}$	$\frac{\text{mem}(K_{\mathcal{H}})^{(\ell)}}{\text{mem}(K_{\mathcal{H}})^{(\ell-1)}}$
0	$2,048 \times 1,026$	0.9160	67	15	—	—
1	$8,192 \times 4,098$	0.4063	477	104	7.11	7.09
2	$32,768 \times 16,386$	0.1279	2,541	524	5.32	5.04
3	$131,072 \times 65,538$	0.0404	13,111	2,645	5.16	5.05
4	$524,288 \times 262,146$	0.0123	65,146	12,902	4.97	4.88

The observations continue for the discrete double layer operator. Also for this case we detect an almost linear complexity of the Adaptive Cross Approximation. The results are depicted in Tab. VII. However, the absolute computation times are considerably higher than those for the single layer operator. E.g., for the finest discretization of $524,288$ elements the computation of K_h took 18 hours while the equivalent discrete single layer potential V_h was computed in approximately 1.3 hours. The reason for this blow-up is due to the fact that the ACA-assembly routines differ from those for dense-matrices. Dense matrices are assembled element-by-element whereas ACA demands the computation of an entire matrix entry at once. While this is not critical for the creation of the single-layer potential it massively affects the computation of the double layer potential since the support of the degrees of freedom is enlarged by the choice of piecewise continuous functions $\varphi_h^1 \in Y_h^1(\Gamma)$. Hence, it might happen that the same computation is repeated several times. A possible remedy for this drawback is the minimization of the support of the degrees of freedom. Instead of using a piecewise continuous boundary element space we perform the computations with piecewise discontinuous space functions $\psi_h^1 \in Z_h^1(\Gamma) \supset Y_h^1$. Thus, we end up with the modified discrete operator equation

$$V_h \underline{u}_N - \left(\tfrac{1}{2} M_h + K_h A^\top\right) \underline{u}_D = \underline{R} \tag{23}$$

with

$$K_h\colon (\psi_h^0, \psi_h^1) \in Z_h^0(\Gamma) \times Z_h^1(\Gamma) \quad \wedge \quad A\colon (\varphi_h^1, \psi_h^1) \in Y_h^1(\Gamma) \times Z_h^1(\Gamma) \ .$$

The newly introduced matrix A embeds continuous boundary element spaces into discontinuous spaces. In BETL, the corresponding structure is called `EmbeddingOperator`. Its implementation as well as its use is equivalent to that of the identity operator (see p. 18)

```
typedef EmbeddingOperator< dofhandler_Yh_t ,
                           dofhandler_Zh_t > embedding_op_t;
embedding_op_t A_op( dofhandler_Yh , dofhandler_Zh );
A_op.compute( );
embedding_op_t::const_reference A = A_op.giveMatrix( );
```

The ACA results for the modified system (23) are given in Tab. VIII. We observe that both the compression rates η as well as the overall computation times T_Q are almost halved. However, these benefits do not come without costs. The total memory requirement for storing the compressed matrix $K_{\mathcal{H}}$ increases by a factor 3 which is owed to the fact that the total number of degrees of freedom has been increased extensively.

By utilizing the space $Z_h^1(\Gamma)$ we restrict the support of one degree of freedom to a single element. Nevertheless, this space features three degrees of freedom per element and ACA might need to compute all the corresponding matrix entries via three distinct integration calls. In order to eliminate this additional overhead BETL buffers already computed values in a map container with global row and column indices as keys. This container works as follows: Whenever an element combination occurs for the first time

Table VIII. ACA approximations $K_{\mathcal{H}} \approx K_h : (\psi_h^0, \psi_h^1) \in Z_h^0(\Gamma) \times Z_h^1(\Gamma)$ of the double layer operator.

ℓ	$\dim(K_h)$	$\eta := \frac{\text{mem}(K_{\mathcal{H}})}{\text{mem}(K_h)}$	T_Q [s]	$\text{mem}(K_{\mathcal{H}})$ [MB]	$\frac{T^{(\ell)}}{T^{(\ell-1)}}$	$\frac{\text{mem}(K_{\mathcal{H}})^{(\ell)}}{\text{mem}(K_{\mathcal{H}})^{(\ell-1)}}$
0	$2,048 \times 6,144$	0.6272	53	60	—	—
1	$8,192 \times 24,576$	0.2140	294	329	5.58	5.46
2	$32,768 \times 98,304$	0.0690	1,528	1,696	5.20	5.16
3	$131,072 \times 393,216$	0.0213	7,589	8,363	4.97	4.93
4	$524,288 \times 1,572,864$	0.0063	36,354	39,813	4.79	4.76

the integration routine is called and the computed values are then stored within the cache container. As soon as another entry for this particular element combination is needed it will be queried and immediately erased from the container.

Table IX. ACA approximations of $K_{\mathcal{H}} \approx K_h : (\psi_h^0, \psi_h^1) \in Z_h^0(\Gamma) \times Z_h^1(\Gamma)$ with activated cache routines. $\widetilde{Q}(5)$ denotes the quadrature rule from Eqn. (24).

ℓ	$\dim(K_h)$	dim(Cache)	T_Q [s]	$T_{\widetilde{Q}(5)}$ [s]	$\lvert\underline{R}_Q\rvert$	$\lvert\underline{R}_{\widetilde{Q}(5)}\rvert$
0	$2,048 \times 6,144$	5,859	31	16	1.54_{-5}	1.53_{-5}
1	$8,192 \times 24,576$	6,726	180	87	1.82_{-6}	1.81_{-6}
2	$32,768 \times 98,304$	6,901	963	454	2.17_{-7}	2.21_{-7}
3	$131,072 \times 393,216$	9,172	4,883	2,262	2.66_{-8}	2.95_{-8}
4	$524,288 \times 1,572,864$	36,684	24,232	11,052	3.32_{-9}	4.79_{-9}

Tab. IX shows the maximum size of the cache containers during the creation of the discrete double layer potential. Although there is a blow-up in cache size for the finest grid its total size of approximately $37,000$ entries remains to be rather small. Therefore, the cache does not require relevant additional memory resources. The default maximum size of the caching routines is set to $100,000$ entries. If this limit is exceeded the cache will be erased.

The computation times T_Q are given in column 4 of Tab. IX. Compared to the computation times for the non-cached calculations we observe a further reduction by a factor of approximately 1.5 such that the computation of $K_{\mathcal{H}}$ on the finest grid lasts about $6^3/_4$ hours. This duration is perfectly reasonable when one considers the facts that the dimensions of K_h outrange the size of the discrete single layer operator V_h by a factor of 6 and that the discrete single layer potential $V_{\mathcal{H}}$ takes 1.3 hours to be created (cf. Tab. VI).

Trying to avoid repeated computations is one possibility to accelerate the Adaptive Cross Approximation. Another way to boost the calculations is to minimize the computational costs within each integration. This can be achieved by taking the fundamental solutions' asymptotic behavior into account. The kernel functions are of order $O(\frac{1}{|\mathbf{y}-\mathbf{x}|^\alpha})$, $\alpha = 1, 2$ such that it is reasonable to decrease the number of Gaussian points for an increasing distance $|\mathbf{y} - \mathbf{x}|$. Therefore, we define a heuristic set of quadrature points

$$\widetilde{Q}(L) := \{n_L^{[7,3]}(\tau_{\mathbf{x}}, \tau_{\mathbf{y}}), 36, 25, 16\} \tag{24}$$

with

$$n_L^{[N_1,N_2]}(\tau_{\mathbf{x}}, \tau_{\mathbf{y}}) := \begin{cases} N_1 & \text{if } \frac{\text{dist}(\tau_{\mathbf{x}}, \tau_{\mathbf{y}})}{\max(h_{\mathbf{x}}, h_{\mathbf{y}})} \leq L \\ N_2 & \text{else} \end{cases} .$$

Above, $h_{\mathbf{x}}$ and $h_{\mathbf{y}}$ denote the local mesh sizes of two boundary elements $\tau_{\mathbf{x}}$ and $\tau_{\mathbf{y}}$, respectively. Their distance is approximated by

$$\operatorname{dist}(\tau_{\mathbf{x}},\tau_{\mathbf{y}}) = |\bar{\mathbf{y}}-\bar{\mathbf{x}}| - \tfrac{1}{2}\,(h_{\mathbf{x}}+h_{\mathbf{y}})$$

where $\bar{\mathbf{x}}$, $\bar{\mathbf{y}}$ again represent the barycenters of the two elements $\tau_{\mathbf{x}}$ and $\tau_{\mathbf{y}}$. BETL implements the class `HeuristicGalerkinIntegrator` which is based on quadrature rules in form of (24). Using this class instead of the default integration routine leads, again, to a dramatic improvement of the computational times. Column five of Tab. IX shows the computation times with respect to a relative distance $L = 5$. With these settings the creation of $K_{\mathcal{H}}$ on the finest grid is finished already after 3 hours instead of the former $6^3/4$ hours. It is equally important to note that the residual norm $|\underline{R}_{\widetilde{Q}(5)}|$ has not been reasonably affected by choosing a reduced quadrature scheme.

Finally, there is a third way for accelerating the creation of discrete boundary integral operators by means of ACA. The AHMED library comes with parallel versions of the ACA routines which are based on OpenMP. In this work, only the single-threaded routines of AHMED have been used but BETL supports also the multi-threaded versions. These routines scale almost perfectly [Bebendorf and Kriemann 2005] such the computation of the above operator $K_{\mathcal{H}}$ can be performed in about 10 minutes on the same workstation with 24 cores.

4.4. Single trace formulations for acoustic scatterers

BETL is well-suited for rapid prototyping. We will demonstrate this by means of a multi-domain Helmholtz transmission problem for composite scatterers

$$\begin{aligned} -\Delta u - \omega_i^2 u = 0 \qquad \text{in} \quad \Omega_i\,, i = 1\ldots N \\ \text{+ hom. Dirichlet and Neumann jumps on interfaces } \Gamma_{ij} := \Omega_i \cap \Omega_j \\ \text{+ inhom. Dirichlet and/or Neumann jumps on the boundary } \Gamma \\ \text{+ radiation conditions} \end{aligned} \tag{25}$$

where N denotes the total number of subdomains and $\omega \in \mathbb{R}$ is the wavenumber. Details on boundary element formulations for this problem can be found in [Hiptmair and Jerez-Hanckes 2012] or [Claeys et al. 2012].

Boundary element formulations for the numerical solution of (25) are based on the Calderón operator

$$A_\omega := \begin{pmatrix} -K_\omega & V_\omega \\ W_\omega & K'_\omega \end{pmatrix} \,.$$

For simplicity, we will consider only two subdomains, an interior domain Ω^- and an exterior domain Ω^+. For both subdomains there holds the systems of boundary integral equations

$$\begin{aligned} \left(-\tfrac{1}{2}\,\mathrm{Id} + A_{\omega_-}\right)\begin{pmatrix}\gamma_0^- u \\ \gamma_1^- u\end{pmatrix} = 0 \qquad \text{for} \qquad \Omega^- \\ \left(-\tfrac{1}{2}\,\mathrm{Id} - A_{\omega_+}\right)\begin{pmatrix}\gamma_0^+ u_s \\ \gamma_1^+ u_s\end{pmatrix} = 0 \qquad \text{for} \qquad \Omega^+\,. \end{aligned} \tag{26}$$

The function u_s denotes the scattered wave and u is the total field which is $u = u_{inc} \in \Omega^-$ and $u = u_s + u_{inc} \in \Omega^+$. The incident field is given by u_{inc}.

Subtracting the set of exterior boundary integral equations from their interior counterparts and imposing a variational form yields

$$\langle (A_{\omega_-} + A_{\omega_+}) \begin{pmatrix} \gamma_0^- u \\ \gamma_1^- u \end{pmatrix}, \begin{pmatrix} \varphi \\ \phi \end{pmatrix} \rangle = \langle \begin{pmatrix} \gamma_0^- u_{inc} \\ \gamma_1^- u_{inc} \end{pmatrix}, \begin{pmatrix} \varphi \\ \phi \end{pmatrix} \rangle \qquad (\varphi, \phi) \in Z_h(\Gamma) \times Y_h(\Gamma) .$$

The formulation above is a special case of the classical single trace formulation

$$\sum_{i=1}^{N} L_i^* A_{h,\omega_i} L_i \begin{bmatrix} \underline{u} \\ \underline{t} \end{bmatrix} = L_{ext}^* M_h \begin{bmatrix} \underline{u}_{inc} \\ \underline{t}_{inc} \end{bmatrix} \tag{27}$$

where the matrices A_{h,ω_i} denote the discrete Calderón identities while the sparse matrices L_i are localization operators which restrict the degrees of freedom on the complete mesh to the i-th subdomain. L_i^* is the adjoint of L_i. The implementation of the system (27) within BETL can be done in a short time frame since BETL

— can deal with multiple domains. This is furnished by the `MultiMesh` structure:

```
// instead of Mesh< element_t >
typedef MultiMesh< element_t > mesh_t;
mesh_t mesh( input );
// this returns the begin iterator to the elements of the i-th
// subdomain
mesh_t::const_element_iterator begin = mesh.begin( i )
```

— implements not only all discrete boundary integral operators but also discrete Calderón operators
— supports a general framework to construct sparse operators. Hence, implementing the localization operators becomes an easy task.

Instead of subtracting the equations in (26) from each other one can also add them. This gives the so-called *2nd kind formulation*

$$(M_h - A_{h,\Delta\omega}) \begin{bmatrix} \underline{u} \\ \underline{t} \end{bmatrix} = M_h \begin{bmatrix} \underline{u}_{inc} \\ \underline{t}_{inc} \end{bmatrix}, \quad \Delta\omega := \omega_- - \omega_+ \tag{28}$$

with the newly introduced discrete operator $A_{h,\Delta\omega}$. Again, the implementation of the system (28) does not pose serious problems since the operator $A_{h,\Delta\omega}$ demands only the implementation of modified fundamental solutions like

$$G_H^{\Delta\omega}(\mathbf{z}) := G_H^{\omega_-}(\mathbf{z}) - G_H^{\omega_+}(\mathbf{z}) .$$

In BETL fundamental solutions are implemented as functors. As long as these functors supply an interface as it is stated in Eqn. (18) they can directly be used.

The left part of Fig. 13 depicts a simulation based on the single trace formulation (27) with three subdomains. The picture's right part shows the solution according to the system (28). The prescribed incident field is given by plane wave with $\omega = \omega^+ = 1$. The wave number for all interior domains is $\omega^- = 2$. The given result is the real part of the total field, i.e., $\mathrm{Re}(\underline{u})$. All operators have been assembled as dense matrices and the solution has been obtained by a direct solver.

The Fig. 14 illustrates the solution $\mathrm{Re}(\underline{t})$. As above, the left part is the solution according to the system (27) while the right part is the solution for the 2nd kind formulation (28). Clearly, some artefacts in the 1st kind solution are obvious. This is due to the extremely ill-conditioned system matrices which, in fact, is one of the main motivations to develop more sophisticated Boundary Element formulations such as, e.g., 2nd kind formulations. BETL's data structures and algorithms are well-suited to give support for the development and testing of novel Boundary Element formulations.

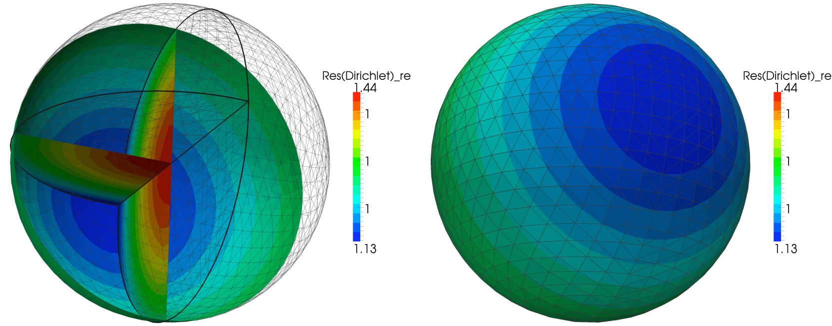

Fig. 13. Single trace formulations of the 1st and 2nd kind: Dirichlet results.

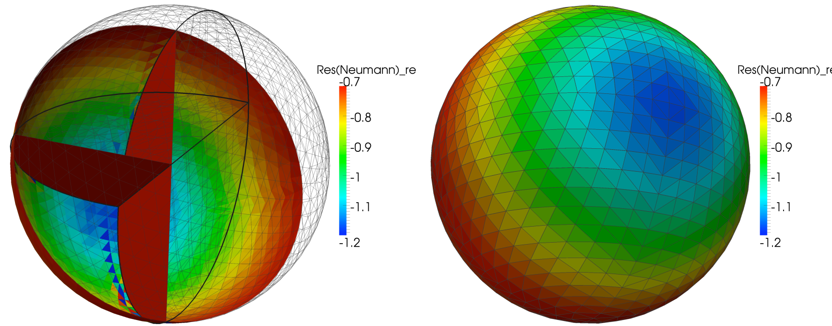

Fig. 14. Single trace formulations of the 1st and 2nd kind: Neumann results.

4.5. Efficiently solving a Neumann Boundary Value Problem

BETL comes with iterative solvers such as CG or GMRes. In combination with ACA and suitable preconditioners they can be used to build efficient Boundary Element solvers. We will demonstrate this by means of the interior Neumann boundary value problem for the Laplace equation

$$\begin{aligned} -\Delta u &= 0 \qquad \text{in} \quad \Omega \\ \gamma_1 u &= g \qquad \text{on} \quad \Gamma \,. \end{aligned} \tag{29}$$

In order to establish a well-posed problem we need to consider the solvability condition

$$\int_\Gamma g(\mathbf{x}) \, \mathrm{d}s_{\mathbf{x}} = 0 \,.$$

Further, since for a constant u_0 any $\tilde{u} = u + u_0$ solves (29) we demand

$$\int_\Gamma \gamma_0 u(\mathbf{x}) \, \mathrm{d}s_{\mathbf{x}} = 0 \tag{30}$$

to fix the solution. For a Boundary Element formulation corresponding to (29) the second equation of (2) is utilized

$$\langle W\gamma_0 u, \phi\rangle + \alpha\langle \gamma_0 u, 1\rangle\langle \phi, 1\rangle = \langle (\tfrac{1}{2}\,\mathrm{Id} - K')g, \phi\rangle \,. \tag{31}$$

The discrete hypersingular operator is semi-definite. To establish a positive definite operator we need to incorporate (30). This is reflected by the term $\alpha\langle\gamma_0 u, 1\rangle\langle\phi, 1\rangle$. For simplicity, we will use the scaling parameter $\alpha = 1$. For more information see [Steinbach 2008].

We will discretize the variational form (31) with lowest order test- and trial-spaces on flat triangular elements, i.e., the discrete hypersingular operator is of the form $W_h\colon (\phi_h^1, \phi_h^1) \in Y_h^1(\Gamma) \times Y_h^1(\Gamma)$. For lowest order spaces, the involved surface curls become constant expressions and, therefore, they can be extracted from the integral kernel (cf. Eqn. (14)). On a discrete level, this allows for the representation

$$W_h := \sum_{i=1}^{3} C_{h,i} \widehat{V}_h C_{h,i}^\top$$

where $\widehat{V}_h$ is the single layer operator $\widehat{V}_h\colon (\varphi_h^0, \varphi_h^0) \in Z_h^0(\Gamma) \times Z_h^0(\Gamma)$ and the matrices $C_{h,i}\colon (\phi_h^1, \varphi_h^0) \in Y_h^1(\Gamma) \times Z_h^0(\Gamma)$ are sparse matrices containing the components of the constant surface curls

$$C_{h,i}[\ell, k] := \begin{cases} \mathbf{curl}_\Gamma\, \phi_{h,\ell}[i] & \text{if} \quad \varphi_{h,k} \in \operatorname{supp}(\phi_{h,\ell}) \\ 0 & \text{else} . \end{cases}$$

The discrete operator equation corresponding to (31) then reads

$$\widetilde{W}_h \underline{u} = (\tfrac{1}{2}\widetilde{M}_h - K_h')\underline{g}$$

with $\widetilde{W}_h := W_h + \underline{s} \otimes \underline{s}$. The stabilization $\underline{s} \otimes \underline{s}$ is given by the vector $\underline{s} := \widetilde{M}_h \underline{1}$. The spaces for the discrete identity and the discrete adjoint double layer operator are $\widetilde{M}_h | K_h'\colon (\phi_h^1, \varphi_h^0) \in Y_h^1(\Gamma) \times Z_h^0(\Gamma)$.

A well-known preconditioner for the above system is [Steinbach and Wendland 1998]

$$C_W^{-1} := M_h^{-1} \widetilde{V}_h M_h^{-1}$$

with the single layer potential $\widetilde{V}_h\colon (\phi_h^1, \phi_h^1) \in Y_h^1(\Gamma) \times Y_h^1(\Gamma)$ and the discrete identity M_h being also is discretized by means of the space $Y_h^1(\Gamma)$.

So far, the discrete scheme demands the creation of the two discrete single layer operators $\widehat{V}_h$ and $\widetilde{V}_h$, respectively. To reduce the costs, it is, however, desirable to create only one discrete operator. Corresponding to Sec. 4.3 the discrete operator $\widetilde{V}_h$ virtually begs for a discretization based upon discontinuous test- and trial-spaces. Hence, the strategy is as follows:

— Discretize the single layer operator $V_h\colon (\varphi_h^1, \varphi_h^1) \in Z_h^1(\Gamma) \times Z_h^1(\Gamma)$
— Create discrete embeddings $B\colon Z_h^0 \subset Z_h^1$, $A\colon Y_h^1 \subset Z_h^1$
— Define $\widehat{V}_h := B V_h B^\top$, and $\widetilde{V}_h := A V_h A^\top$
— Don't forget the right hand-side: Define $K_h' := A\widetilde{K}_h'$, $\widetilde{K}_h'\colon (\varphi_h^1, \varphi_h^0) \in Z_h^1(\Gamma) \times Z_h^0(\Gamma)$

With these preliminary considerations we are ready to start the implementation. We will discuss its most important details in the following.

In Fig. 15, the discrete spaces $Z_h^0(\Gamma)$, $Z_h^1(\Gamma)$, and $Y_h^1(\Gamma)$ are declared and defined via respective dofhandler objects. Only the spaces $Z_h^0(\Gamma)$ and $Z_h^1(\Gamma)$ are enriched with some particular ACA functionality (cf. Sec. 3.3).

Once the spaces are in place, we proceed with the declaration and construction of the discrete sparse operators. This is depicted in Fig. 16.

It remains to compute the discrete boundary integral operators V_h, $\widetilde{K}_h'$. Since the setup of the BEM model follows directly that from Fig. 7 we will skip the details and assume that proper integrator types and respective integrator instances have been

```
typedef Element<3> element_t;
// the bases of Z^0, Z^1, Y^1
typedef FEBasis< element_t, CONSTANT, Discontinuous,
                 LagrangeTraits > fe_Z0_t;
typedef FEBasis< element_t, LINEAR, Discontinuous,
                 LagrangeTraits > fe_Z1_t;
typedef FEBasis< element_t, LINEAR, Continuous,
                 LagrangeTraits > fe_Y1_t;
// ...corresponding dofhandler type declarations
typedef DoFHandler< fe_Z0_t, ACA > dh_Z0_t; // use ACA!
typedef DoFHandler< fe_Z1_t, ACA > dh_Z1_t; // use ACA!
typedef DoFHandler< fe_Y1_t      > dh_Y1_t; // no ACA
// instantiate objects and distribute degrees of freedom
dh_Z0_t dh_Z0;
dh_Z1_t dh_Z1;
dh_Y1_t dh_Y1;
dh_Z0.distributeDoFs( mesh.e_begin(), mesh.e_end() );
dh_Z1.distributeDoFs( mesh.e_begin(), mesh.e_end() );
dh_Y1.distributeDoFs( mesh.e_begin(), mesh.e_end() );
// Call ACA initialization routines for dh_Z0, dh_Z1
// The object 'aca_settings' stores the aca parameters
dh_Z0.clusterize( aca_settings );
dh_Z1.clusterize( aca_settings );
```

Fig. 15. *All* discrete spaces $Z_h^0(\Gamma)$, $Z_h^1(\Gamma)$, $Y_h^1(\Gamma)$

```
// embedding operators
typedef EmbeddingOperator< dh_Z0_t, dh_Z1_t > B_op_t;
typedef EmbeddingOperator< dh_Y1_t, dh_Z1_t > A_op_t;
// BETL provides classes to create sparse matrices
// containing constant surface derivatives
typedef CurlOperator< dh_Y1_t, dh_Z0_t > C_op_t;
// discrete identites
typedef IdentityOperator< dh_Y1_t, dh_Z0_t > M_YZ_op_t;
typedef IdentityOperator< dh_Y1_t, dh_Y1_t > M_YY_op_t;
// ...create instances
B_op_t    B_op( dh_Z0, dh_Z1 );
A_op_t    A_op( dh_Y1, dh_Z1 );
C_op_t    C_op( dh_Y1, dh_Z0 );
M_YZ_op_t M_YZ( dh_Y1, dh_Z0 );
M_YY_op_t M_YY( dh_Y1, dh_Y1 );
// ...compute matrices
B_op.compute( );
A_op.compute( );
C_op.compute( );
M_YZ_op.compute( );
M_YY_op.compute( );
```

Fig. 16. This constructs *all* discrete sparse operators

created. The code excerpt in Fig. 17 is intended to illustrate the seamless integration of ACA.

```
   // type definitions for discrete bem operators
46 typedef BemOperator< integrator_V_t,
                        dh_Z1_t, dh_Z1_t > V_op_t;
48 typedef BemOperator< integrator_K_adj_t,
                        dh_Z1_t, dh_Z0_t > K_adj_op_t;
50 // ...instantiate operators
   V_op_t     V_op    ( integrator_V    , dh_Z1, dh_Z1 );
52 K_adj_op_t K_adj_op( integrator_K_adj, dh_Z1, dh_Z0 );
   // BEM operators need to be initialized with ACA settings.
54 // 'aca_settings': The same as for the DoFHandler-setup.
   V_op.setup( aca_settings );
56 K_adj_op.setup( aca_settings );
   // this computes the operators via ACA
58 V_op.compute( );
   K_adj_op.compute( );
```

Fig. 17. Seamless ACA integration for the fast computation of discrete BEM operators

Now, all operators, the sparse as well as the BEM operators are in place and everything can be glued together. Since we are aiming at an iterative solver scheme it remains to implement the applications of $\widetilde{W}_h$ and of C_W^{-1} to a vector, i.e., we have to implement the matrix-vector products for these operators. In BETL, matrix-vector products are always of the form $\underline{y} \leftarrow \underline{y} + \alpha A \underline{x}$ and they are implemented via the following interface

```
// amux method of A: compute y <- op( A ) x + y
// op = 'T' :: use A^T
template< typename T >
void amux( T alpha, const T* x, T* y, char op ) const;
```

So far, BETL does not implement any expression templates and it is not clear whether it ever will do so. Though there is no doubt that expression templates come in quite handy they have some serious drawbacks [see Weinberger et al. 2012, Operator Overloading].

Any class that features an `amux` method can be used within an iterative solver scheme. Hence, we can create a composite object named `HyperMatrix` which implements the matrix-vector product

$$\widetilde{W}_h \underline{x} = \sum_{i=1}^{3} C_{i,h} B V_h B^\top C_{i,h}^\top \underline{x} + \underline{s}\,(\underline{s} \cdot \underline{x})$$

as successive calls of the respective matrix-vector products of every single involved operator *(Not forget to mention the scalar product $\underline{s} \cdot \underline{x}$)*. The preconditioner is constructed in a similar way. In BETL, preconditioners perform the operation $\underline{x} \leftarrow P^{-1}\underline{x}$. They are implemented as functors with the interface declaration

```
template< typename T >
void operator()( T* x ); // application x <- inv(P) x
```

For the present example the preconditioner interface needs to implement

$$C_W^{-1} \underline{x} = M_h^{-1} A V_h A^\top M_h^{-1} \underline{x}\,.$$

The implementations of the composite structures for the hypersingular operator and for the preconditioner are straightforward, and, thus, will be skipped here.

However, the preconditioner C_W^{-1} demands an application of the inverse identity operator. We will briefly comment on how BETL deals with inverse applications of sparse matrices.

Beside the possibility of working with an iterative solver to realize $M_h^{-1}\underline{x}$ BETL allows for the incorporation of very efficient direct sparse solvers. Currently, it provides interfaces to Pardiso [Schenk and Gärtner 2004], SuperLU [Demmel et al. 1999; Li 2005], and UMFPACK [Davis 2004]. Matrices can be accessed via traits classes. The following code snippet illustrates how a direct solver scheme for sparse matrices is defined and how the operation $\underline{x} \leftarrow M_h^{-1}\underline{x}$ can be performed

```
// declare the matrix traits for sparse matrices in CCS format
typedef MatrixTraits< CCS, double >      MT;
// direct solver scheme. Also possible: PARDISO, SUPERLU
typedef MT::SolveDirect< UMFPACK >       directSolver;
// declare the factorize module
typedef directSolver::Factorize          factorize;
// define a data type which holds the factorization
typedef factorize::decomposition_type    decomposition_t;
// initialize the factorization pointer
decomposition_t* decomp = factorize::initDecomposition( M );
// decompose M
factorize::computeDecomposition( M, decomp );
// apply inv(M) x
factorize::FBSubstitution( decomp, x );
```

Switching from one direct solver scheme to another is only about the exchange of `UMFPACK` with `PARDISO`, or `SUPERLU`, respectively. Except its last line, the code above is executed when the preconditioner is initialized. The last routine `FBSubstitution` is called each time the preconditioner is applied to a vector $\underline{x}$.

It remains to comment on the iterative solver. At the moment, BETL implements CG as well as GMRes solvers. A typical solver instantiation as well as the respective solution method read

```
// define the solver type.
// Exchange CG with GMRes for a different solver
typedef solver::CG< double > solver_t;
// create a solver instance
solver_t solver( MatrixSize, max_iter, accuracy );
// solve
solver.solve( hyper_matrix, rhs, sol, inv_C_W );
```

A solver object is created with some general parameters such as the matrix dimension, the maximum number of iterations, and a prescribed accuracy. The solution itself is then obtained via the `solve` routine. As mentioned several times, BETL relies on very general definitions of matrices. The same holds for preconditioners. Hence, the `solve` routine's interface is

```
// MATRIX_T : Must support an amux-routine, nothing more
// PRECOND_T: Must be a functor. P( NUMERIC_T* ) must be valid
// NUMERIC_T: Given at solver declaration
template< typename MATRIX_T, typename PRECOND_T >
std::pair< bool, int > solve( const MATRIX_T&   A,
                              const NUMERIC_T* b,
                              NUMERIC_T*        sol,
                              PRECOND_T&        P ) const;
```

The routine's return value is a pair containing information about the solver's success and about the total number of iterations.

Note that, in order to preserve interoperability the interfaces of the preconditioner, that of the matrix-vector products, and the solvers' interfaces are consciously chosen to be as simple as possible. A vector is assumed to store its elements in contiguous storage locations. Hence, no matter what vector-classes might be used, only the pointer to the underlying storage arrays are passed. For this reason, the interfaces above don't pay any attention to the memory locations' validity. It is left to the user to assure that no dead or invalid links are passed.

The implementation will be validated by means of an example whose geometry is given in Fig. 18. It shows a tube with non-zero boundary data on its cross sections and zero boundary data on its skin surface. Note that, from a physical point of view, the solution of the problem (29) might be interpreted as some electric surface current density. In this context the prescribed boundary data represents an excitation current and the tube might be considered as a current driven wire.

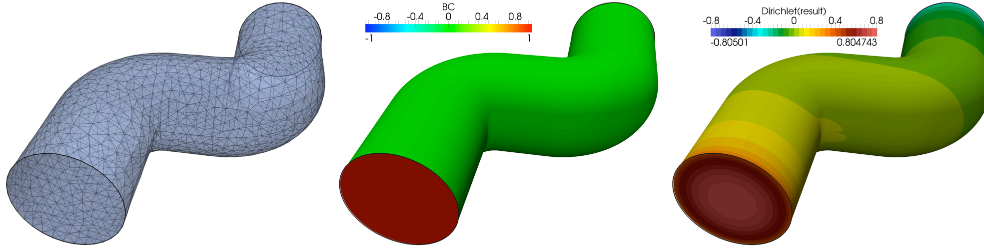

Fig. 18. Left: A discretization with $2,480$ flat triangular elements. Center: The prescribed boundary conditions, $g = \pm 1$ on the cross sections, $g = 0$ on the skin surface. Right: The solution $\underline{u}$ for a model consisting of $634,880$ elements.

For the measurements we choose a solver accuracy of $\varepsilon_{solver} = 1 \cdot 10^{-5}$ and the ACA accuracy is set to $\varepsilon_{\text{ACA}} = 1 \cdot 10^{-4}$. The remaining ACA parameters are those of Sec. 4.3. The same holds for the hardware and software specifications. The computations of the discrete boundary integral operators are performed by using the default integrator together with the quadrature rule (22) and activated cache structures. Tab. X summarizes the most interesting information of the presented solver scheme. Clearly, the most time consuming part consists in the creation of the single layer operator. The preconditioner's setup has no significant influence on the overall computation time. The solution times T_{sol} are also moderate compared to the creation times T_V. Finally, the efficiency of the preconditioner is confirmed by the stable iteration numbers.

Table X. Measurements for the Neumann-BVP solver. T_V is the time in seconds needed for the construction of the single layer operator. The setup of the preconditioner is based on using UMFPACK for factorization. T_P denotes its setup time. T_{sol} is the solution time.

#Elem	$\eta := \frac{\text{mem}(V_{\mathcal{H}})}{\text{mem}(V_{\text{full}})}$	T_V [s]	T_P [s]	T_{sol} [s]	#It.	$\left\lvert\int_\Gamma u_h \,\mathrm{d}s\right\rvert$
2,480	0.05212	25	0.008	0.57	10	2.216_{-6}
9,920	0.01669	116	0.042	2.48	9	1.140_{-6}
39,680	0.00515	543	0.231	16.93	9	3.491_{-7}
158,720	0.00154	2,498	1.460	60.35	9	2.825_{-7}
634,880	0.00045	11,908	11.985	448.37	9	7.064_{-8}

4.6. A FEM-BEM coupling application

For the solution of the following magnetostatic transmission problem BETL has been integrated into an industrial Finite Element solver. The magnetostatic field equations are

$$\begin{aligned} \mathbf{curl}\,\tfrac{1}{\mu}\mathbf{B} &= \mathbf{j} && \text{in} \quad \Omega^- \\ \mathbf{curl}\,\tfrac{1}{\mu_0}\mathbf{B} &= \mathbf{0} && \text{in} \quad \Omega^+ \\ \operatorname{div}\mathbf{B} &= 0 && \text{in} \quad \Omega^- \cup \Omega^+ \end{aligned}$$

with the transmission and decay conditions

$$[\![\mathbf{B}\cdot\mathbf{n}]\!]_\Gamma = 0\,, \qquad [\![\mathbf{H}\times\mathbf{n}]\!]_\Gamma = \mathbf{0}\,, \qquad \lim_{|\mathbf{x}|\to\infty} |\mathbf{x}|\,\mathbf{H} = \mathbf{0}\,.$$

Above, $[\![\cdot]\!]_\Gamma$ denotes the jump across the interface Γ, $\mathbf{j}$ is a given solenoidal current density, $\mathbf{B}$ is the unknown magnetic flux density, and $\mathbf{H}$ represents the magnetostatic field. Additionally, the exterior domain Ω^+ is equipped with a constant permeability μ_0 whereas the interior domain Ω^- features a permeability $\mu(\mathbf{x})$.

The interior domain is treated by a standard Finite Element Method based on a vector potential $\mathbf{A}$ such that $\mathbf{B} = \mathbf{curl}\,\mathbf{A}$. The domain discretization is performed with lowest order edge-elements on tetrahedral meshes [Nedelec 1980]. For the exterior domain we employ a reduced scalar potential φ with $\mathbf{grad}\,\varphi := -\frac{1}{\mu_0}(\mathbf{B}-\mathbf{B}_0)$. The vector field $\mathbf{B}_0$ is called excitation field and it can be computed via a Biot-Savart integration which, for the given problem, admits for a boundary integral representation

$$\mathbf{B}_0 = -\mu_0\,(V(\mathbf{n}\times\mathbf{j})) \tag{32}$$

with V being the single layer operator according to the Laplace equation. Kuhn and Steinbach [2002] deduced the method for simply connected domains. Later Pusch and Ostrowski [2008] enhanced the scheme to multiply connected domains. We therefore refer to those publications for more details about this method. While both former works rely on dense representations of the involved boundary integral operators, within BETL we can easily turn on the Adaptive Cross Approximation once the final linear block system

$$\begin{bmatrix} \mu_0 W_h & -(\frac{1}{2}M_h + K_h)^\top Q \\ Q^\top(\frac{1}{2}M_h + K_h) & A_h + \frac{1}{\mu_0}Q^\top V_h Q \end{bmatrix} \begin{bmatrix} \underline{\varphi} \\ \underline{a} \end{bmatrix} = \begin{bmatrix} \underline{f}_1(\mathbf{B}_0) \\ \underline{f}_2(\mathbf{B}_0,\mathbf{j}) \end{bmatrix} \tag{33}$$

has been implemented. The matrices W_h, K_h, and V_h are the Laplacian discrete hypersingular operator, the discrete double layer operator, and the discrete single layer operator, respectively. The used test- and trial-spaces are the lowest order spaces $Z_h^0(\Gamma)$ and $Y_h^1(\Gamma)$. Further, the matrix A_h denotes the finite element matrix, M_h is the discrete identity operator, and the matrix Q represents the discrete coupling between the reduced scalar potential $\underline{\varphi}$ and the vector potential $\underline{a}$.

For the solution of the linear system (33) we use an iterative solver scheme in conjunction with a well-suited preconditioner similar to that stated in [Kuhn and Steinbach 2002].

Below, we recall the main implementation tasks:

— Extract a surface triangular mesh from the tetrahedral volume mesh.
— Pass this surface mesh to BETL in order to create W_h, K_h, V_h, and M_h.
— The FEM-matrix A_h is assembled using BETL's sparse matrix structures. However, this assembly cannot use any `DoFHandler` structures since they cannot deal with volume meshes.

— Assemble the coupling matrix $Q\colon H_h(\mathbf{curl},\Omega) \to Z_h^0(\Gamma)$ that maps the Finite Element space onto the Boundary Element space. Again, the FEM part is treated without the help of `DoFHandler` objects.
— Assemble matrices into a block-system, i.e., implement a matrix-vector product for the compound linear system as it has been explained in Sec. 4.5.
— Implement the preconditioner. Again, this is done in a similar way as it has been described in Sec. 4.5.
— Pass composite structures to BETL's solver routines.

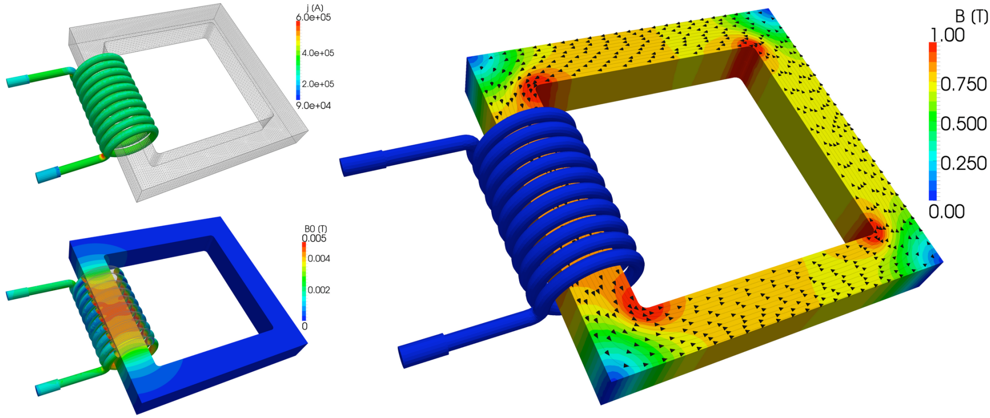

Fig. 19. Magnetostatic FEM-BEM coupling. Upper left: Given DC distribution $\mathbf{j}$ based on an excitation current $I = 100\mathrm{A}$. Lower left: Excitation field $\mathbf{B}_0(\mathbf{j})$ as it is given in Eqn. (32). Right: The computed magnetic flux density $\mathbf{B}$.

The Fig. 19 shows a coil made of copper with conductivity $\sigma_{\text{cu}} = 5.7 \cdot 10^7 (\Omega\mathrm{m})^{-1}$ which encloses a magnetic core with a relative permeability $\mu_r = 1000$. The coil is the current driven body and it carries an excitation current of $I = 100\mathrm{A}$. The model consists of $479,070$ tetrahedra for the volume discretization and of $70,798$ flat triangular elements for the surface mesh. The Tab. XI depicts some convergence results for different spatial discretizations as well as the respective iteration numbers for the iterative solver scheme.

Table XI. Magnetostatic computations based on Eqn. (33). The prescribed solver accuracy has been chosen to be $\varepsilon_{\text{Solver}} = 1 \cdot 10^{-4}$.

No. of Finite Elements	No. of Boundary Elements	No. of Iterations	$\|\mathbf{B}\|^2_{L_2(\Omega^-)}$
38,796	14,532	121	5.848_{-3}
72,204	22,206	134	5.982_{-3}
168,169	40,338	132	6.102_{-3}
204,951	46,362	135	6.107_{-3}
479,070	70,798	131	6.109_{-3}

5. SUMMARY AND OPEN ISSUES

As shown, BETL is a modular and efficient software library with well-defined, lean interfaces. BETL features sufficient abstractions to mimic the mathematical notion of boundary integral operators within C++ program codes.

While BETL has proven to be an efficient tool for many applications, we want to address some of BETL's limitations and absent features:

— A careful choice of numerical quadrature formulas is necessary to compute entries of BEM Galerkin matrices economically and sufficiently accurately. All of the so far implemented integration routines lack adaptivity and none of them offers any error control. An accurate performance in "geometrically challenging" situations cannot be guaranteed. All examples have confirmed that the matrix assembly is the most time consuming part. Thus, highly efficient kernel integrations can deliver a huge benefit.
— The set of discrete finite element spaces need to be enhanced. Higher order discrete edge-based spaces are highly anticipated.
— The *discretization model* entirely relies on static polymorphism. Therefore BETL cannot deal with hybrid surface meshes comprising, e.g., both triangles and quadrilaterals *(However, the integrator interfaces are designed with respect to different element types)*. For the same reason, BETL supports no variable polynomial degree in boundary element spaces ("hp-BEM").
— A local mesh refinement of surfaces meshes is yet to come.
— Point evaluations of representation formulas off the boundary are still missing.
— BETL cannot deal with volume meshes. A `DoFHandler` type for FEM meshes would significantly simplify FEM-BEM coupling schemes. Moreover, it would facilitate the construction of Newton potentials. These are also not in place.
— BETL seamlessly integrates AHMED's multi-threading capabilities. However, a BETL port to distributed memory clusters is still missing.
— While Fast Multipole Methods (FMM) are implemented within BETL, their integration has to be improved. Currently, the user interfaces to FMMs are in a pre-release state.

Note that implementing most of the missing features will not be a problem within the BETL framework. However, if hp-BEM functionality is desired one might be enforced to set up another discretization model.

APPENDIX

The numerical examples from Sec. 4 have been performed by means of of the Calderon identity. It remains to deduce estimates for the residual vectors.

Evaluating the first boundary integral equation for an interior domain Ω^- yields the identity

$$0 = V(\gamma_1 u) - \left(\tfrac{1}{2}\,\mathrm{Id} + K\right)\gamma_0 u \qquad \text{in } \Omega^- \,. \tag{34}$$

Note, that the above equation holds if u is the exact solution to the underlying partial differential equation. Let X_0 and X_1 be the respective boundary element spaces, then the boundary integral operators map continuously

$$V\colon X_1 \to X_0\,, \qquad K\colon X_0 \to X_0\,.$$

Based on the discrete trial and test spaces

$$X_{0,h} \subset X_0 \quad \wedge \quad X_{1,h} \subset X_1$$

we introduce interpolation operators

$$I_0 \colon X_0 \to X_{0,h} \,, \qquad I_1 \colon X_1 \to X_{1,h} \ .$$

Evaluating (34) for interpolated data gives the residuum $r \in X_{0,h}$

$$r := V(I_1\gamma_1 u) - \left(\tfrac{1}{2}\operatorname{Id} + K\right)(I_0\gamma_0 u) \ . \tag{35}$$

Due to linearity (35) is equivalent to

$$r = Ve_1(u) - \left(\tfrac{1}{2}\operatorname{Id} + K\right)e_0(u) \tag{36}$$

with the interpolation errors

$$e_\ell(u) := (I_\ell - \operatorname{Id})\,\gamma_\ell u \,, \quad \ell = 0, 1 \ .$$

In BETL a boundary integral equation is always considered in weak form. Thus, in the context of a Galerkin discretization the residuum equation (36) becomes

$$\langle r, \psi_h \rangle = \langle Ve_1, \psi_h \rangle - \langle \left(\tfrac{1}{2}\operatorname{Id} + K\right)e_0, \psi_h \rangle \,, \quad \psi_h \in X_{1,h}$$

In the numerical experiments in Sec. 4 we have computed the vector of coefficients $R[i] = \langle r, \psi_{h,i} \rangle$, where $\{\psi_{h,i}\}_i$ represents the basis of $X_{1,h}$. By the boundary integral operators' continuity properties and by using the Cauchy-Schwartz inequality we obtain the following estimate

$$|\langle r, \psi_{h,i} \rangle| \leq C_K \|e_0\|_{X_0} \|\psi_{h,i}\|_{X_1} + C_V \|e_1\|_{X_1} \|\psi_{h,i}\|_{X_1} \ .$$

Above, C_K and C_V are two positive constants.

Scaling estimates for the characteristic basis functions $\psi_{h,i}$ can be deduced by inverse inequalities [see Sauter and Schwab 2011, Ch. 4.4]. For the relevant finite element spaces these estimates are:

— Lagrangian basis: $Y_h^\alpha(\Gamma)$, $Z_h^\beta(\Gamma) \subset X_1$

$$\|\psi_{h,i}\|_{X_1} \leq Ch^{\frac{3}{2}} \ .$$

— Edge basis: $\mathbf{Y}_{\perp,h}(\operatorname{curl}_\Gamma, \Gamma)$, $\mathbf{Z}_{\|,h}(\operatorname{div}_\Gamma, \Gamma) \subset X_1$,

$$\|\boldsymbol{\psi}_{h,i}\|_{X_1} \leq Ch^{\frac{1}{2}} \,,$$
$$\|\operatorname{div}_\Gamma \boldsymbol{\psi}_{h,i}\|_{X_1} \leq Ch^{-\frac{1}{2}} \ .$$

For interpolation error estimates we also refer to [see Sauter and Schwab 2011, Ch. 2.1.7]. Their derivation is based on interpolation in Sobolev spaces. Here, these estimates are:

— Lagrangian basis:

$$\|e_0\|_{X_0} \leq Ch^{\alpha + \frac{1}{2}} \,, \qquad Y_h^\alpha(\Gamma) \subset X_0 \,,$$
$$\|e_1\|_{X_1} \leq Ch^{\beta + \frac{3}{2}} \,, \qquad Z_h^\beta(\Gamma) \subset X_1 \ .$$

— Edge basis:

$$\|e_i\|_{X_i} \leq Ch^{\frac{3}{2}} \,, \quad i = 0, 1 \quad \mathbf{Y}_{\perp,h}(\operatorname{curl}_\Gamma, \Gamma),\ \mathbf{Z}_{\|,h}(\operatorname{div}_\Gamma, \Gamma) \subset X_1 \ .$$

The combination of the scaling estimates and the error estimates yields the following estimates for the residual functionals:

— Lagrange basis *(assume $\alpha > \beta$)*:

$$|\langle r, \psi_{h,i} \rangle| \leq \tilde{C} h^{\beta+3} \ .$$

— Edge basis:

$$|\langle r, \boldsymbol{\psi}_{h,i} \rangle| \leq \tilde{C} h \ .$$

The residual estimates in Sec. 4.1 are stated for the Euclidean vector norm $|\cdot|$. In this norm we obtain the final estimates

— Lagrange basis *(assume $\alpha > \beta$)*:

$$|\underline{R}| \leq \tilde{C} h^{\beta+2} \ , \tag{37}$$

— Edge basis:

$$|\underline{R}| \leq \tilde{C} h^{0} \ . \tag{38}$$

REFERENCES

ABRAHAMS, D. AND GURTOVOY, A. 2004. *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond*. C++ In-Depth Series. Addison-Wesley Professional.

ALEXANDRESCU, A. 2004. *Modern C++ Design: Generic Programming and Design Patterns Applied*. C++ In-Depth Series. Addison-Wesley.

ALPERT, B., BEYLKIN, G., COIFMAN, R., AND ROKHLIN, V. 1993. Wavelet-like bases for the fast solutions of second-kind integral equations. *SIAM Journal for Scientific Computing 14,* 1, 159–184.

ARRIDGE, S., BETCKE, T., PHILLIPS, J., SCHWEIGER, M., AND ŚMIGAJ, W. (accessed October, 2012). BEM++: An open source boundary element library. `http://www.bempp.org`.

BALAY, S., BROWN, J., BUSCHELMAN, K., EIJKHOUT, V., GROPP, W., KAUSHIK, D., KNEPLEY, M., MCINNES, L. C., SMITH, B., AND ZHANG, H. 2012. *PETsc Users Manual* 3.3 Ed. Mathematics and Computer Science Division, Argonne National Laboratory.

BANGERTH, W., HARTMANN, R., AND KANSCHAT, G. 2007. deal.II – a General Purpose Object Oriented Finite Element Library. *ACM Transactions on Mathematical Software 33,* 4, 24/1–24/27.

BASTIAN, P., BLATT, M., DEDNER, A., ENGWER, C., KLÖFKORN, R., KORNHUBER, R., OHLBERGER, M., AND SANDER, O. 2008. A generic grid interface for parallel and adaptive scientific computing. Part II: implementation and tests in DUNE. *Computing 82*, 121–138. 10.1007/s00607-008-0004-9.

BEBENDORF, M. 2008. *Hierarchical matrices*. Lecture Notes in Computational Science and Engineering Series, vol. 63. Springer, Berlin.

BEBENDORF, M. (accessed October, 2012). Another software library on hierarchical matrices for elliptic differential eqautions (ahmed). Online. http://bebendorf.ins.uni-bonn.de/AHMED.html.

BEBENDORF, M. AND KRIEMANN, R. 2005. Fast parallel solution of boundary integral equations and related problems. *Computing and Visualization in Science 8*, 121–135.

BEBENDORF, M. AND RJASANOW, S. 2003. Adaptive Low-Rank Approximation of Collocation Matrices. *Computing 70*, 1–24.

BONNET, M. 1995. *Boundary integral equation methods for solids and fluids*. Wiley, New York.

BRAESS, D. 2007. *Finite Elemente* 4th Ed. Springer-Verlag, Berlin, Heidelberg.

CIRAK, F. AND CUMMINGS, J. C. 2008. Generic programming techniques for parallelizing and extending procedural finite element programms. *Engineering with Computers 24,* 1, 1–16.

CLAEYS, X., HIPTMAIR, R., AND JEREZ-HANCKES, C. 2012. Multi-trace boundary integral equations. Tech. Rep. 2012-20, Seminar for Applied Mathematics, Swiss Federal Institute of Technology Zurich.

CMake (accessed November 2012). CMake: Cross Platform Build. Online. `http://www.cmake.org`.

DAVIS, T. A. 2004. Algorithm 832: UMFPACK V4.3—an unsymmetric-pattern multifrontal method. *ACM Transactions on Mathematical Software 30,* 2, 196–199.

DAVIS, T. A. (accessed October, 2012). Suite Sparse: a Suite of Sparse matrix packages. Online. http://www.cise.ufl.edu/research/sparse/SuiteSparse/.

DEDNER, A., KLÖFKORN, R., NOLTE, M., AND OHLBERGER, M. 2010. A generic interface for parallel discretization schemes: abstraction principles and the DUNE-FEM module. *Computing 90*, 165–196.

DEMMEL, J. W., GILBERT, J. R., AND LI, X. S. 1999. *SuperLU Users' Guide*. Berkeley National Laboratory.

DUFF, I. S., GRIMES, R. G., AND LEWIS, J. G. 1989. Sparse matrix test problems. *ACM Transactions on Mathematical Software 15,* 1, 1–14.

DUNAVANT, D. A. 1985. High degree efficient symmetrical Gaussian quadrature rules for the triangle. *International Journal for Numerical Methods in Engineering 21*, 1129–1148.

ERICHSEN, S. AND SAUTER, S. A. 1998. Efficient automatic quadrature in 3-d Galerkin BEM. *Computer Methods in Applied Mechanics and Engineering 157*, 215–224.

GAUL, L., KÖGL, M., AND WAGNER, M. 2003. *Boundary element methods for engineers and scientists: an introductory course with advanced topics*. Engineering Online Library. Springer.

GEUZAINE, C. AND REMACLE, J.-F. 2009. Gmsh: A 3-D finite element mesh generator with built-in pre- and post-processing facilities. *International Journal for Numerical Methods in Engineering 79,* 11, 1309–1331.

GOTTSCHLING, P., WISE, D. S., AND ADAMS, M. D. 2007. Representation-transparent matrix algorithms with scalable performance. In *Proceedings of the 21st annual international conference on Supercomputing*. ICS '07. ACM, New York, NY, USA, 116–125.

GREENGARD, L. AND ROKHLIN, V. 1987. A Fast Algorithm for Particle Simulations. *Journal of Computational Physics 73*, 325–348.

HACKBUSCH, W. 1989. *Integralgleichungen – Theorie und Numerik*. Leitfäden der angewandten Mathematik und Mechanik LAMM Series, vol. 68. B.G. Teubner.

HACKBUSCH, W. 2009. *Hierarchische Matrizen: Aglorithmen und Analysis*. Springer-Verlag Berlin Heidelberg.

HACKBUSCH, W. AND NOWAK, Z. P. 1989. On the fast matrix multiplication in the boundary element method by panel clustering. *Numerische Mathematik 54*, 463–491.

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 Transactions on Mathematical Software 31,* 3, 397–423.

HIPTMAIR, R. 2006. Operator Preconditioning. *Computers and Mathematics with Applications 52,* 5, 699–706. Hot Topics in Applied and Industrial Mathematics.
HIPTMAIR, R. 2007. Boundary element methods for eddy current computation. In *Boundary Element Analysis*, M. Schanz and O. Steinbach, Eds. Lecture Notes in Applied and Computational Mechanics Series, vol. 29. Springer Berlin / Heidelberg, 213–248.
HIPTMAIR, R. AND JEREZ-HANCKES, C. 2012. Multiple traces boundary integral formulation for Helmholtz transmission problems. *Advances in Computational Mathematics 37*, 39–91.
HSIAO, G. AND WENDLAND, W. 2008. *Boundary integral equations*. Applied mathematical sciences. Springer.
KARLSSON, B. 2006. *Beyond the C++ Standard Library: An introduction to Boost*. Addison-Wesley Professional.
KIRK, B. S., PETERSON, J. W., STOGNER, R. H., AND CAREY, G. F. 2006. `libMesh`: A C++ Library for Parallel Adaptive Mesh Refinement/Coarsening Simulations. *Engineering with Computers 22,* 3–4, 237–254.
KUHN, M. AND STEINBACH, O. 2002. Symmetric coupling of finite and boundary elements for exterior magnetic field problems. *Mathematical Methods in the Applied Sciences 25*, 357–371.
LI, X. S. 2005. An overview of SuperLU: Algorithms, implementation, and user interface. *ACM Transactions on Mathematical Software 31,* 3, 302–325.
LIU, Y. (accessed October, 2012). Fast Multipole Boundary Element Method Software. Online. http://urbana.mie.uc.edu/yliu/Software/.
MAUE, A. W. 1949. Zur Formulierung eines allgemeinen Beugungsproblems durch eine Integralgleichung. *Zeitschrift für Physik 126,* 7–9, 601–618.
MCLEAN, W. 2000. *Strongly elliptic systems and boundary integral equations*. Cambridge University Press.
MESSNER, M., SCHANZ, M., AND DARVE, E. 2012. Fast directional multilevel summation for oscillatory kernels based on chebyshev interpolation. *Journal of Computational Physics 231,* 4, 1175–1196.
MUSSER, D. R., DERGE, G. J., AND SAINI, A. 2009. *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library* 3rd Ed. Addison-Wesley Professional.
NEDELEC, J. 1980. Mixed finite elements in $\mathbb{R}^3$. *Numerische Mathematik 35*, 315–341.
NEDELEC, J. 1982. Integral equations with non integrable kernels. *Integral Equations and Operator Theory 5*, 563–672.
NISHIMURA, N. 2002. Fast multipole accelerated boundary integral equation methods. *Applied Mechanics Review 55,* 4, 299–324.
OF, G. 2006. BETI-Gebietszerlegungsmethoden mit schnellen Randelementverfahren und Anwendungen. Ph.D. thesis, Institut für Angewandte Analysis und Numerische Simulation, Universität Stuttgart.
OF, G., STEINBACH, O., AND WENDLAND, W. L. 2006. The fast multipole method for the symmetric boundary integral formulation. *IMA Journal of Numerical Analysis 26,* 2, 272–296.
PUSCH, D. AND OSTROWSKI, J. 2008. Robust FEM-BEM Coupling for Magnetostatics on multi-connected Domains. *IEEE Transactions on Magnetics 46*, 3177–3180.
RJASANOW, S. AND STEINBACH, O. 2007. *The Fast Solution of Boundary Integral Equations*. Mathematical and Analytical Techniques with Applications to Engineering Series, vol. 7. Springer Science+Business Media, LLC, New York.
ROKHLIN, V. 1985. Rapid solution of integral equations of classical potential theory. *Journal of Computational Physics 60,* 2, 187–207.

SAUTER, S. A. AND SCHWAB, C. 2011. *Boundary Element Methods*. Springer-Verlag, Berlin Heidelberg.

SCHENK, O. AND GÄRTNER, K. 2004. Solving Unsymmetric Sparse Systems of Linear Equations with PARDISO. *20,* 3, 475–487.

SCHÖBERL, J. (accessed November 2012). NGSolve Finite Element Libray. Online. http://sourceforge.net/projects/ngsolve/.

SCHROEDER, W., MARTIN, K., AND LORENSEN, B. 2006. *The Visualization Toolkit: An Object-Oriented Approach To 3D Graphics* 4th Ed. Kitware, Inc. publishers.

SIEK, J. G. AND LUMSDAINE, A. 1999. The Matrix Template Library: Generic Components for High-Performance Scientific Computing. *Computing in Science and Engineering 1*, 70–78.

STEINBACH, O. 2008. *Numerical Approximation Methods for Elliptic Boundary Value Problems: Finite and Boundary Elements*. Texts in applied mathematics. Springer.

STEINBACH, O. AND WENDLAND, W. 1998. The construction of some efficient preconditioners in the boundary element method. *Advances in Computational Mathematics 9*, 191–216.

STROUSTRUP, B. 2000. *The C++ Programming Language* 3rd Ed. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

SUTRADHAR, A., PAULINO, G., AND GRAY, L. 2008. *Symmetric Galerkin Boundary Element Method*. Springer.

Ublas (accessed November 2012). uBLAS: Basic Linear Algebra Library. Online. http://www.boost.org/doc/libs/1_52_0/libs/numeric/ublas/doc/index.htm.

VANDERVOORDE, D. AND JOSUTTIS, N. M. 2002. *C++ Templates: The Complete Guide*. Addison-Wesley Professional.

VELDHUIZEN, T. L. 1998. Arrays in Blitz++. In *In Proceedings of the 2nd International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE98*. Springer-Verlag, 223–230.

WEINBERGER, B., SILVERSTEIN, C., EITZMANN, G., MENTOVAI, M., AND LANDRAY, T. 2012. Google C++ Styling Guide. Tech. rep., Google, Inc. http://code.google.com/p/google-styleguide/.

Research Reports

No.	Authors/Title
12-36	*R. Hiptmair and L. Kielhorn* BETL A generic boundary element template library
12-35	*S. Mishra, N.H. Risebro, Ch. Schwab and S. Tokareva* Numerical solution of scalar conservation laws with random flux functions
12-34	*R. Hiptmair, Ch. Schwab and C. Jerez-Hanckes* Sparse tensor edge elements
12-33	*R. Hiptmair, C. Jerez-Hanckes and S. Mao* Extension by zero in discrete trace spaces: Inverse estimates
12-32	*A. Lang, S. Larsson and Ch. Schwab* Covariance structure of parabolic stochastic partial differential equations
12-31	*A. Madrane, U.S. Fjordholm, S. Mishra and E. Tadmor* Entropy conservative and entropy stable finite volume schemes for multi-dimensional conservation laws on unstructured meshes
12-30	*G.M. Coclite, L. Di Ruvo, J. Ernest and S. Mishra* Convergence of vanishing capillarity approximations for scalar conservation laws with discontinuous fluxes
12-29	*A. Abdulle, A. Barth and Ch. Schwab* Multilevel Monte Carlo methods for stochastic elliptic multiscale PDEs
12-28	*E. Fonn, Ph. Grohs and R. Hiptmair* Hyperbolic cross approximation for the spatially homogeneous Boltzmann equation
12-27	*P. Grohs* Wolfowitz's theorem and consensus algorithms in Hadamard spaces
12-26	*H. Heumann and R. Hiptmair* Stabilized Galerkin methods for magnetic advection
12-25	*F.Y. Kuo, Ch. Schwab and I.H. Sloan* Multi-level quasi-Monte Carlo finite element methods for a class of elliptic partial differential equations with random coefficients
12-24	*St. Pauli, P. Arbenz and Ch. Schwab* Intrinsic fault tolerance of multi level Monte Carlo methods
12-23	*V.H. Hoang, Ch. Schwab and A.M. Stuart* Sparse MCMC gpc Finite Element Methods for Bayesian Inverse Problems