BETL Documentation

COLLABORATORS			
	TITLE : BETL Documentation		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY	Lars Kielhorn	October 26, 2012	

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME

Contents

List of Figures

Foreword

This is the documentation on BETL. BETL is an acronym for Boundary Element Template Library. It is intended to support researchers and scientists in their development of robust and reliable Boundary Element applications.

BETL is a C++ template library for the discretisation of boundary integral operators. While it currently implements the discretisation of 3-dimensional boundary integral operators via Galerkin schemes its design principles allow also for the incorporation of other discretisation schemes such as, e.g., the still popular collocation methods.

Over the years this project has grown slightly bigger than it was originally expected. In its original form BETL was just intended to serve as set of methods with rather limited functionality for maintaining an existing industrial Boundary Element solver. But over the years BETL has become a fully autonomous tool on which powerful Boundary Element applications can be build. BETL's strength lies in its use of well-known design principles in conjunction with state of the art C++ language features. This ensures the implementation of robust, extendable, and reliable numerical schemes being somehow related to the discretisation of boundary integral operators. May it help you on your particular field of research.

The BETL software is provided ‘as is’, without expressed or implied warranty.

Part I

The basics of BETL

Chapter 1

A short survey on boundary integral operators

1.1 Introduction

This part of the BETL documentation serves as an introduction to Boundary Element Methods — in particular it is meant to be an introduction to symmetric Galerkin-based three-dimensional Boundary Element Methods. Naturally, this introduction is neither a comprehensive treatise on Boundary Integral Equations nor gives it an detailed overview on their respective discretisation schemes, the Boundary Element Methods. For this the reader is referred to well-known literature like, e.g., [GlEtAl03], [Hckbsch89], [HsWdlnd08], [McLn00], [StrSchwb11], [Stnbch08], [StrdhrEtAl08]. This introductory chapter is basically about the definition of a common language. Hence, even if you are familiar with Boundary Element Methods *(and we assume that you are!)* at least a rough reading through this chapter is recommended since it may help to clarify the notion of expressions.

1.2 Green's identities

For the things that follow we will deduce a formula for integrating by parts being based on the well-known Divergence Theorem

$$\int\limits_{\Omega} \mathrm{div}\mathbf{u}\; dx = \int\limits_{\partial\Omega} \langle \mathbf{u}, \mathbf{n} \rangle \; ds_x \tag{1.1}$$

for some differentiable function $\mathbf{u}$. In 1.1 $\partial\Omega$ is the sufficiently smooth boundary of the domain of interest Ω and $\mathbf{n}$ is its outer normal vector. With the definition

$$\mathbf{u} = \psi \mathbf{grad}\phi \quad \Longrightarrow \quad \mathrm{div}\mathbf{u} = \psi\Delta\phi + \langle \mathbf{grad}\psi, \mathbf{grad}\phi \rangle \tag{1.2}$$

we end up with *Green's first identity*

$$\int\limits_{\Omega} \psi\Delta\phi + \langle \mathbf{grad}\psi, \mathbf{grad}\phi \rangle dx = \int\limits_{\partial\Omega} \psi \frac{\partial \phi}{\partial \mathbf{n}} ds_x \tag{1.3}$$

where

$$\frac{\partial \phi}{\partial \mathbf{n}} = \lim_{\Omega \ni \tilde{x} \to x \in \partial\Omega} \langle \mathbf{grad}\phi\,(\tilde{x})\,, \mathbf{n}\,(x) \rangle \tag{1.4}$$

denotes the normal derivative of the function ϕ . Green's first identity is perfectly suited to be used as starting point for the derivation of *Finite Element Methods* — at least for the Laplace equation. Next, we consider the function $\mathbf{u}$ from 1.1 to be composed by the product of the gradient of ψ times the function ϕ . The gives a similar equation like 1.3 but with swapped differential operators now acting on ψ rather than on ϕ . Subtracting this formula from the Green's first identity yields *Green's second identity*

$$\int\limits_{\Omega} \psi\,\Delta\phi - \Delta\psi\,\phi\; dx = \int\limits_{\partial\Omega} \psi \frac{\partial \phi}{\partial \mathbf{n}} - \frac{\partial \psi}{\partial \mathbf{n}} \phi\; ds_x \tag{1.5}$$

where the symmetric parts of each formula have been cancelled out. Contrary to Green's first identity on which Finite Element Methods can be build Green's second identity forms somehow the basis for the construction of Boundary Element Methods as will be shown in the following sections.

1.3 Fundamental solution and representation formula

Lets assume that we want to solve some boundary value problem for the Laplace equation

$$-\Delta\phi = 0 \quad \text{in } \Omega \subset \mathbb{R}^3, \qquad + \text{boundary conditions} \,. \tag{1.6}$$

We could easily plug 1.6 into Green's second identity by what the domain integral's first term instantly vanishes. For the remaining domain integral we introduce a two point function $U(x-\tilde{x}) = \psi(x)$ for which we demand that

$$-\int_{\mathbb{R}^3} \Delta U(x-\tilde{x})\,\phi(x)\,dx \overset{!}{=} \phi(\tilde{x}) \tag{1.7}$$

holds for $x \neq \tilde{x}$. A function with the above property is commonly denoted as *fundamental solution*. It usually builds the backbone for every Boundary Element Method since it is its main ingredient. For the Laplace operator the fundamental solution is simply

$$U(z) = \frac{1}{4\pi}\frac{1}{|z|} \quad \forall z \in \mathbb{R}^3 \,. \tag{1.8}$$

Finally, inserting the fundamental solution into Green's second identity we end up with

$$\phi(\tilde{x}) = \int_{\partial\Omega} \frac{\partial\phi}{\partial\mathbf{n}(y)} U(y-\tilde{x}) - \phi\,\frac{\partial}{\partial\mathbf{n}(y)} U(y-\tilde{x})\,ds_y \quad \forall \tilde{x} \in \mathbb{R}^3 \setminus \partial\Omega,\, y \in \partial\Omega \tag{1.9}$$

which is known as *representation formula*. The representation formula is a boundary integral equation for the PDE 1.6. It determines the solution of the Laplace equation completely just by the Cauchy data $\{\phi(x), \partial\phi(x)/\partial\mathbf{n}(x)\}$. Unfortunately, the complete Cauchy data is hardly known for some given boundary value problem. It is therefore the main task of Boundary Element Methods to complete the Cauchy data by calculating the missing parts of the Dirichlet data and the Neumann data. This computation can be done via so-called boundary layer potentials. They will be derived in the following section and, essentially, they are what the BETL is all about.

1.4 Trace Operators and Boundary Integral Operators

At this point we need to comment on trace operators which are needed when deriving the respective boundary integral operators. Actually, one trace operator, namely the Neumann trace, has been already introduced in 1.4. Nevertheless, to give a more common notion of those trace operators we will refer them by γ_0 an γ_1 from now on. For the Laplace equation they are defined as

$$(\gamma_0\phi)(x) = \lim_{\Omega \ni \tilde{x} \to x \in \partial\Omega} \phi(\tilde{x}) \,, \qquad (\gamma_1\phi)(x) = \gamma_0\langle \mathbf{grad}\phi(\tilde{x}), \mathbf{n}(x)\rangle \,. \tag{1.10}$$

With those trace operators the representation formula becomes

$$\phi(\tilde{x}) = \int_{\partial\Omega} (\gamma_1\phi)(y)\;(\gamma_0 U)(y-\tilde{x}) - (\gamma_0\phi)(y)\;(\gamma_1 U)(y-\tilde{x})\,ds_y \quad \forall \tilde{x} \in \mathbb{R}^3 \setminus \partial\Omega,\, y \in \partial\Omega \,. \tag{1.11}$$

Finally, the boundary integral operators are obtained by the application of the Dirichlet trace γ_0 and the Neumann trace γ_1 to the representation formula. If the domain Ω where the analysis takes place is an interior domain we end up with the two following boundary integral equations

$$\gamma_0\phi = V\gamma_1\phi + (\frac{1}{2}I - K)\gamma_0\phi \tag{1.12}$$

and

$$\gamma_1\phi = (\frac{1}{2}I + K')\gamma_1\phi + D\gamma_0\phi \ . \tag{1.13}$$

The above equations are usually denoted as the first and the second boundary integral equation, respectively. In case of an exterior boundary value problem those two equations take a similar form as above but with some sign changes in front of the operators. The operator I simply denotes the identity operator. It represents the jump in normal direction across the boundary. Since we will only address Galerkin Boundary Element schemes the factor in front of the identity can be safely set to 0.5. All the remaining operators in 1.12 and in 1.13 are the most common boundary integral operators. They are usually named as the single layer operator, the double layer operator, the adjoint double layer operator, and as the hypersingular integral operator. Their definitions are

$$\begin{aligned}
(Vw)(x) &= \int\limits_{\partial\Omega} U(y-x)\, w(y)\, ds_y \\
(Kw)(x) &= \lim_{\varepsilon\to 0} \int\limits_{\partial\Omega:\ |y-x|\geq\varepsilon} (\gamma_{1,y}U)(y-x)\, w(y)\, ds_y \\
(K'w)(x) &= \lim_{\varepsilon\to 0} \int\limits_{\partial\Omega:\ |y-x|\geq\varepsilon} (\gamma_{1,x}U)(y-x)\, w(y)\, ds_y \\
(Dw)(x) &= -\gamma_{1,x} \lim_{\varepsilon\to 0} \int\limits_{\partial\Omega:\ |y-x|\geq\varepsilon} (\gamma_{1,y}U)(y-x)\, w(y)\, ds_y \ .
\end{aligned} \tag{1.14}$$

Special attention has to be paid to the fact that the fundamental solution is singular at the origin. Usually, this singularity is common for boundary integral operators. Hence, the definitions above contain some special limiting processes already indicating the different kind of singularities those operators consist of. Since the single layer potential is weakly singular the integral exists in an improper sense. These singularities are quite easy to tackle. The situation changes for the double layer potential as well as for the adjoint double layer potential. In general, the occurring integrals have to be understand as strongly singular integrals. They are also referred to as *Cauchy principal values* when a limiting process is applied which approaches the singularity uniformly from every direction. As the name already induces the hypersingular integral operator is somehow the *worst case scenario*. These kind of singularities are commonly denoted as *finite part integrals*. Within a computational scheme integrals of this kind are hardly computable in general. Fortunately there exists a remedy for this drawback. Within Galerkin schemes integration by parts can be used in order to express the associated hypersingular bilinear form via the single layer operator. Examples will be given within the tutorials.

Note that the integral operators do not necessarily need to act on some Dirichlet data $\gamma_0\phi$ or Neumann data $\gamma_1\phi$. In fact, the definition in 1.14 is given for some more or less arbitrary density function w . Hence, one could also think of Boundary Element Methods which are not build upon the representation formula. In fact, those methods exist and they are the so-called *indirect Boundary Element Methods*. Contrary, schemes that utilise the boundary integral equations 1.12 and/or 1.13 are widely known as *direct Boundary Element Methods*.

In the tutorial part of this documentation we will embed the deduced boundary integral operators into Galerkin schemes. In contrast to other possible discretisation schemes like, e.g., Collocation or Nyström methods, the Galerkin methods have the advantage of preserving essential properties of the underlying boundary integral operators also on a discrete level.

Chapter 2

The Concepts of BETL

2.1 Introduction

to be written...

2.2 Defining finite element spaces in BETL

BETL creates finite element spaces by exploiting two independent information sources. On one hand the topological data is needed. This data is either provided via the `Mesh` class or via some intermediate structure which extracts a certain surface patch from the mesh. On the other hand we need some information about the basis a particular finite element space corresponds to. This information is provided by the `FEBasis` class. This section will be roughly divided into two parts. At first, we will deal with a proper definition of the `FEBasis`. Afterwards, the `DoFHandler` will be introduced. This class is of fundamental importance since it combines the topological information with the information about the FE space's basis. Hence, this class actually spans the finite element space.

2.2.1 The FE Basis

For the definition of the finite element space's basis the following information is needed:

1. The element type
2. The approximation order
3. The continuity of the discrete space
4. A main category the finite element space belongs to

Let's start from the back with the explanation of these items. Finite Element spaces are classified by categories or families of functions. These families subsume unique properties of the corresponding space. So far the BETL provides two fundamentally different families. The first one corresponds probably to the most common class of FE spaces. It is the family of *Lagrangian* or *hat* basis functions. These functions feature the well-known property

$$\phi_i(\mathbf{p}_j) = \delta_{ij} \tag{2.1}$$

where ϕ_i is the function corresponding to the i-th local dof, $\mathbf{p}_j$ is the location of the j-th local dof and δ_{ij} is the Kronecker symbol. Those functions are well suited for classical potential problems as well as for problems in acoustics. Another fundamental family is known as the family of *edge* functions. For instance, these functions are important for the discretisation of of Maxwell's equations. In BETL the Lagrangian functions are specified by the structure `LagrangeTraits` while the edge functions are either specified by `EdgeDivTraits` or by `EdgeCurlTraits`, respectively. Both, the `EdgeDivTraits` as well as the

`EdgeCurlTraits` are edge functions. The only difference is that the latter contains the rotated functions of the former. Mathematically, this means that if $\mathbf{u} \in H(\mathrm{div}_\Gamma)$ belongs to the space of functions where the surface divergence is square integrable, that $\mathbf{v} \in H(\mathrm{curl}_\Gamma)$ belongs to the space of functions whose surface curls are square integrable. The connection between these functions is given by the rotation operator

$$\mathbf{R}(\mathbf{w}) = \mathbf{n} \times \mathbf{w} \tag{2.2}$$

such that $\mathbf{v} = \mathbf{R}(\mathbf{u})$. In 2.2 $\mathbf{n}$ is simply the outward normal vector.

The third item from the enumeration above describes how the degrees of freedom *(dofs)* will be distributed. For instance, a dof lying on an edge or on a corner of an element could be shared by all of its neighbouring elements or it may be exclusively attached to the current element. Within BETL, the first scenario is determined by the `Continuous` type and the second one is defined by the `Discontinuous` type. The decision whether the dofs might be distributed continuously or discontinuously might not be necessarily a property of the `FEBasis`. The decision is made herein because of the somehow famous ‘historical reasons’. The continuous/discontinuous dof distribution has been always a part of the `FEBasis`. That’s it. Nevertheless, this may change in future versions of the BETL.

Let’s go ahead. The second item from the enumeration describes the order of a function, i.e., whether the function might be constant, linear, quadratic, or of some higher order. For functions that belong to the Lagrangian family BETL implements constant, linear, and quadratic functions. Contrary, edge-functions are only implemented at lowest-order, i.e., currently only linear edge-functions are supported. The choice of the approximation order is made by the `APPROX_ORDER` enum.

The one item that is left is the first one from the enumeration. With this, one simply determines the general shape of an element, i.e., whether it is a triangle or a quadrilateral. Here, ‘general shape’ means that the geometrical approximation of an element is by no means considered. Since the `FEBasis` is independent of any topological information it could be defined either on a flat or on a curved geometry approximation.

Now, since the description of the `FEBasis` is finished we can give some examples on how this data type is actually constructed.

Example 2.1 Possible finite element basis declarations

```
#include "febasis/febasis.hpp"
using namespace betl;
// lowest order lagrangian functions on flat triangles
typedef FEBasis< Element<3>,
                 CONSTANT,
                 Discontinuous,
                 LagrangeTraits > feb_e3_const_disc_lagr_t;
// lowest order divergence functions on flat triangles
typedef FEBasis< Element<3>,
                 LINEAR,
                 Continuous,
                 EdgeDivTraits > feb_e3_lin_cont_div_t;
// quadratic, continuous functions on flat quadrilaterals
typedef FEBasis< Element<4>,
                 QUADRATIC,
                 Continuous,
                 LagrangeTraits > feb_e4_quad_cont_lagr_t;
// linear, discontinuous functions on curved triangles
typedef FEBasis< Element<6>,
                 LINEAR,
                 Discontinuous,
                 LagrangeTraits > feb_e6_lin_disc_lagr_t;
```

While Example 2.1 depicts the possible choices for a finite element basis there exist also some combinations which are not allowed. Using these combinations within your code yields in rather crucial compiler errors.

Example 2.2 Impossible finite element basis declarations

```
#include "febasis/febasis.hpp"
using namespace betl;
// ERROR: a constant function can't be approximated continuously
typedef FEBasis< Element<3>,
                 CONSTANT,
                 Continuous,
                 LagrangeTraits > feb_e3_const_cont_lagr_t;
// ERROR: constant edge functions do not exist
typedef FEBasis< Element<3>,
                 CONSTANT,
                 Discontinuous,
                 EdgeCurlTraits > feb_e3_const_disc_curl_t;
// ERROR: higher order divergence functions are not implementd right now
typedef FEBasis< Element<3>,
                 QUADRATIC,
                 Continuous,
                 EdgeDivTraits > feb_e3_quad_cont_div_t;
// ERROR: edge functions are only implemented for triangles
typedef FEBasis< Element<4>,
                 LINEAR,
                 Continuous,
                 EdgeDivTraits > feb_e4_lin_cont_div_t;
```

Note that the first and second forbidden combinations in Example 2.2 are disallowed due to logical considerations while the two latter combinations are disallowed because of a lack of implementation. In future versions of BETL there might exist higher order edge functions and there might also exist edge functions for quadrilaterals.

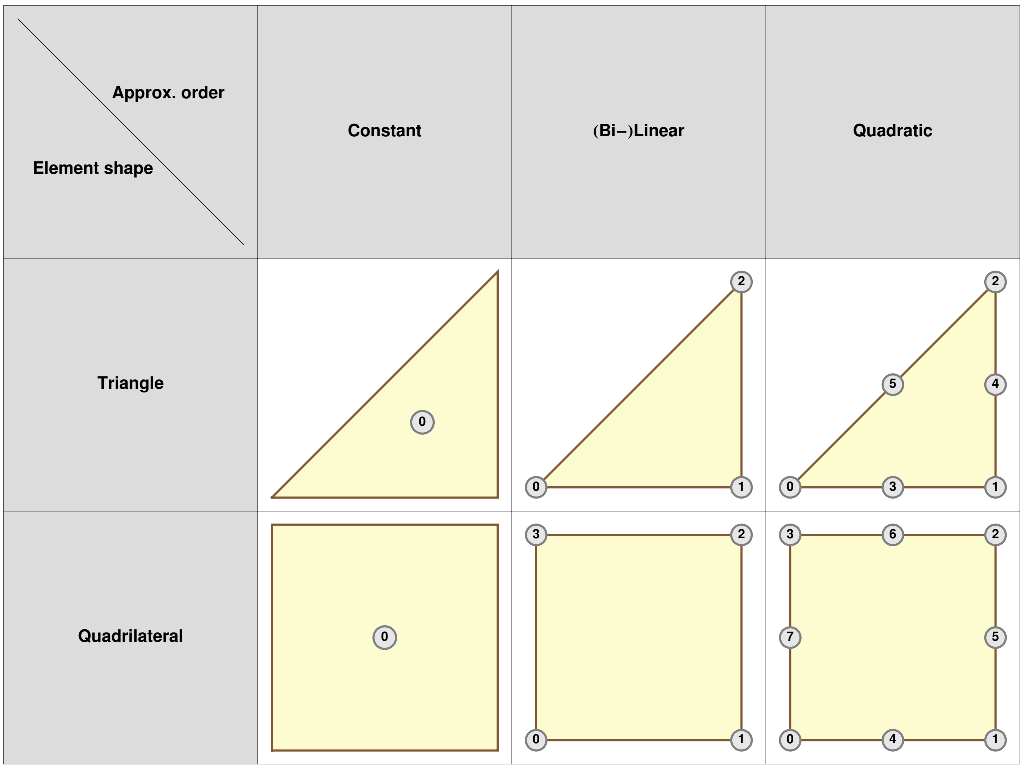

Figure 2.1: Supported Lagrangian bases

In Figure 2.1, the possible choices for Lagrangian based spaces as well as their local dof-numbering are depicted. Additionally, Figure 2.2 shows the local dof distribution for the currently implemented edge bases.

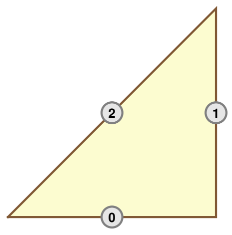

Figure 2.2: Supported edge basis

For completeness, the following formula defines the lowest order edge functions for the $H\left(\mathrm{div}_{\Gamma}\right)$ space

$$\mathbf{u}_e\left(\mathbf{x}\right) = \frac{\mathbf{p}_{i(e)} - \mathbf{x}}{\sqrt{g_\tau}}\,, \quad i\left(e\right) = \left(e+2\right)\mathrm{mod}3\,, \quad e = 0,1,2 \tag{2.3}$$

where g_τ denotes the Gram determinant of the element τ and $\mathbf{p}_{i(e)}$ is the triangle's i-th vertex lying opposite to the edge e .

2.2.2 The dof handler

Once the `FEBasis` has been defined the creation of the finite element space is a quite easy task. As it has been already said at the beginning of this section the finite element space is spanned by an object called `DoFHandler`. This data type combines the bases of the finite element space with the topological information. The following example illustrates the declaration and instantiation of the `DoFHandler`

Example 2.3 A possible dof handler declaration

```
#include "febasis/febasis.hpp"
#include "dof_handler/dof_handler.hpp"
using namespace betl;
// lowest order lagrangian functions on flat triangles
typedef FEBasis< Element<3>,
                 CONSTANT,
                 Discontinuous,
                 LagrangeTraits > feb_e3_const_disc_lagr_t;
// declare the dof handler data type
typedef DoFHandler< feb_e3_const_disc_lagr_t > dofhandler_t;
// instantiate the dofhandler
dofhandler_t dofhandler;
// span the finite element space
dofhandler.distributeDoFs( mesh.e_begin(), mesh.e_end() );
```

In Example 2.3 the `mesh` object has not been declared but one might deduce that this variable represents the mesh on which the analysis will take place. Moreover, the creation of the finite element space's basis has been directly taken from Example 2.1. As you see the creation of the finite element step involves three steps:

- Declare the dof handler type
- Instantiate the dof handler object
- Distribute the dofs by providing two forward iterators to a range of elements

Please refer to Section 4.4 for an example about the usage of the dof handler.

Part II

Working with BETL

Chapter 3

Installing BETL

3.1 Preliminary remarks

This part of the documentation describes the installation of the Boundary Element Template Library. In principle the installation should be the same on all POSIX-like operating systems such as Unix, Linux, and Mac OS X. In addition, since BETL's build process relies on CMake, the library's installation should be also possible with other operating systems like Microsoft Windows. Nevertheless, the installation's focus is clearly on POSIX systems. Up to now, the installation on Windows systems is not sufficiently tested by what it has to be considered as *experimental*.

The installation of BETL requires the following packages:

- CMake (>= 2.8.0)
- Boost (>= 1.46.0)
- g++ (>= 4.4), (alternatives: VC++, icc)
- BLAS and Lapack

Additionally, the use of the following libraries is recommended:

- SuperLU (>= 4.1)
- Intel's MKL (>= 10.3)
- SuiteSparse (CXSparse, UMFPACK)
- AHMED

The term AHMED is an abbreviation for '**A**nother software library on **h**ierarchical **m**atrices for **e**lliptic **d**ifferential equations'. Although the library is not mandatory its use is highly recommended since it accelerates the computations of boundary layer potentials significantly. The AHMED library implements the so-called Adaptive Cross Approximation (ACA) by what the overall algorithm's complexity can be reduced to a logarithmic order instead of its originally quadratic complexity. For non-commercial purpose the library is freely available. Please contact Prof. Dr. Mario Bebendorf for further details.

The use of the SuperLU and the SuiteSparse package becomes advantageous in situations where decompositions of sparse matrices and/or matrix operations other than a matrix-vector product might be necessary. Those operations usually arise within some preconditioning techniques. Of course, you can use also iterative methods for those tasks, but a direct solver is often more convenient since the sparse matrices are usually an order of magnitude smaller than sparse matrices arising from Finite Element discretisations. Alternatively, if you are using the Intel's Math Kernel Library (MKL) you can interface BETL also with the MKL's built-in direct sparse solver Pardiso. When you intend to use AHMED, linking against the MKL might be also a good idea since the MKL offers a highly optimised BLAS functionality on which the ACA heavily relies.

But no matter whether you intend to use the recommended libraries or not, the building process of BETL is not affected by the recommendations above. Hence, in the following section we will describe the installation process of BETL itself. A detailed description on how the additional libraries can be included will follow later.

3.2 Installing BETL

Now, let's come to the compilation of BETL itself. While the library's core functionality is provided as a header-only library there are, nevertheless, a few sources which are organised in form of static libraries. This is what the current section is all about. We will explain the building process of the library as well as its installation process.

The BETL library as well as the tutorials are meant to be out-of-source projects. This means that the all the sources are separated from the build files. The advantage of this strategy is the fact that it is easily possible to have a bunch of build directories. For instance, you might want to have different build directories for debug- or release-builds. Moreover, you might want to have different locations for g++- or icc-compilations. With build folders being well-separated from the sources all that can be done in clean and simple way.

Let's assume that your copy of the BETL is located at `BETL_SOURCE_PATH`. Moreover, we will assume that you have created a build directory outside of the BETL's sources. For instance, a typical location for BETL's sources may be `BETL_SOURCE_-PATH=~/source/betl`. In addition, you may want to build a release version of the BETL libraries at `BETL_BUILD_PAT-H=~/build/betl/lib/release`.

There are two more variables needed for making the compile and installation process work. First, if the Boost libraries are not located at a default path like, e.g., `/usr/include/boost`, `/usr/local/include/boost`, or `/opt/local/i-nclude/boost` we need to define the `BOOST_ROOT` variable. For instance, if you have have a local Boost installation at `~/usr/local/include/boost`

```
export BOOST_ROOT=~/usr/local
```

is a valid declaration within a bash terminal. The second mandatory variable is `BETL_ROOT`. Equivalent to the former `BOOST-_ROOT` variable, this variable determines the target of the BETL installation. Again, we assume that we want to install BETL at a local installation directory. Then, a possible declaration of `BETL_ROOT` is given below.

```
export BETL_ROOT=~/usr/local
```

With the definitions of `BOOST_ROOT` and `BETL_ROOT` the setup for the BETL installation is complete. It remains to instantiate the build system via an appropriate cmake call.

```
cd $BETL_BUILD_PATH
cmake $BETL_SOURCE_PATH/Library/cmk
```

If everything works as expected the outcome of the former call should be something like this:

```
-- The C compiler identification is GNU
-- The CXX compiler identification is GNU
-- Check for working C compiler: /usr/bin/gcc
-- Check for working C compiler: /usr/bin/gcc -- works
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working CXX compiler: /usr/bin/c++
-- Check for working CXX compiler: /usr/bin/c++ -- works
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- OS settings from 'CMakeLists.unix'
-- Boost version: 1.46.0
-- Installation target: $HOME/usr/local
-- Configuring done
-- Generating done
-- Build files have been written to: $HOME/build/betl/lib/release
```

Caution CMake sets the compiler configuration automatically. Unfortunately, it may happen that the system's compiler is not well-suited for compiling BETL. In this case either the environment variable `CXX` should be set to a valid compiler or CMake should be called with the option `-DCMAKE_CXX_COMPILER=`*`name_of_your_compiler`* in order to prescribe a supported compiler. For more information follow this link.

If you are working with a POSIX operating system the CMake call above creates the typical Makefiles from which the library can be build. Alternatively, you may prefer some integrated development environment (IDE) like, e.g., Eclipse, Visual Studio, or XCode. CMake can do this for you via its generator flag. Contact the CMake documentation for more details on that. Here, we will continue with building the library by using Makefiles. This is fairly simple and can be done just by typing:

```
make
make install
```

Note that the latter command possibly demands super user privileges which is typically the case when you have decided to install the library in some system path. Now, when everything has worked without any error messages there will be two `betl` folders in `$BETL_ROOT/include` and `$BETL_ROOT/lib`. While the first path consists of all necessary header files the library path should at least contain the following libraries:

```
ls $BETL_ROOT/lib/betl
libbetlcore.a  libformat_string.a  libresult.a
libclp.a       libgmsh_input.a     libsmpl_input.a
```

If you can detect those libraries in your library path everything went fine and the BETL installation is complete. Congratulations!

Concerning the instantiation of the build system one final remark should be given. If you dislike the definition of some environment variables before calling cmake, it is also possible to declare the `BOOST_ROOT` and `BETL_ROOT` variables directly within the cmake command.

```
cmake -D BOOST_ROOT=~/usr/local -D BETL_ROOT=~/usr/local $BETL_SOURCE_PATH/Library/cmk
```

Note, that as a consequence of this approach you need to define those variables every time you are generating a build system. In other words, the variables are no longer system-wide environment variables but internal cmake variables which must be specified when you are generating the Makefiles either for the library, for the tutorials, or for some self-written applications *(if they rely on cmake)*.

Chapter 4

Tutorial

4.1 Introduction

to be written...

4.2 How to compile the tutorial examples

If you have succeeded with the installation of BETL as it has been described in Chapter 3 the compile process of the tutorial examples is straightforward and should not pose any additional problems.

Throughout this section we assume that you either have the environment variables `BOOST_ROOT` and `BETL_ROOT` properly defined or that you have installed Boost and Betl into some well-known default system paths like, e.g., `/usr` or `/usr/local` such that CMake is able to find them. If you are already familiar with BETL's building process then the building of the tutorial examples is very easy. Again, we want an out-of-source build such that some build directory becomes necessary. For instance, you might want to create a release version of the tutorial examples at `TUT_BETL_BUILD_PATH=~/build/betl/tut/release`. Now, all you need to do is, go to that directory, create the build environment via a CMake call, and build the executables by entering make.

```
cd $TUT_BETL_BUILD_PATH
cmake $BETL_SOURCE_PATH/Tutorial/cmk
make
```

As in Section 4.2 `$BETL_SOURCE_PATH` describes the local copy of BETL's sources. The final `make`-call will create all the executables which will be put either into `$TUT_BETL_BUILD_PATH/bin` or into `~/bin`. The latter path is chosen only if your home-directory contains a `bin`-directory. Contrary, the former path will be selected if `~/bin` does not exist. If you want CMake to put the binaries neither into `$TUT_BETL_BUILD_PATH/bin` nor into `~/bin` you can explicitly define the variable `EXECUTABLE_PATH`. Again, this can be done either by defining an environment variable of that name or by providing the variable directly within the CMake call. For instance, the following call

```
cmake -D EXECUTABLE_PATH=~/betl_bin $BETL_SOURCE_PATH/Tutorial/cmk
```

will enforce CMake to move the executable files to the directory `~/betl_bin`. Additionally, if that directory does not exist CMake will create it. Note that the environment variable `$EXECUTABLE_PATH` overrides the CMake variable given on the command line. Hence, if both variables are used the command line option is neglected and the binaries are always located in the directory being defined by the environment variable.

4.3 Example 1: Processing input files and exporting vtu-files

Actually, the very first example given herein is not about the solution of some boundary value problems nor is it on boundary elements. It simply deals with the presentation of a possible tool-chain which might be used with BETL applications. In particular this example is about

- Creating an input object
- Constructing a mesh
- Exporting the mesh as a vtu-file

Although the input- an export-methods do not belong to the BETL's core functionality they are, nevertheless, a part of the BETL distribution. And, while the use of those structures might be recommended, they are not mandatory since, e.g, the construction of the mesh container does not rely on a particular input-type. The complete source code for this example can be found at `$BETL_ROOT/tutorial/example_1/main.cpp`.

Starting with the coding itself, everything begins with the inclusion of some header files as depicted below.

```
// system includes -------------------------------------------------------------
#include <iostream>
#include <string>
#include <cstdlib>
// reader includes -------------------------------------------------------------
#include "gmsh_input/input.hpp"
// command line parser ---------------------------------------------------------
#include "clp/clp.hpp"
// betl core includes ----------------------------------------------------------
#include "element/element.hpp"
#include "geometry/mesh.hpp"
// exporter includes -----------------------------------------------------------
#include "vtu_exporter/exporter.hpp"
```

Beside the most common C++ headers there are a couple of BETL specific header files. Theses headers are self-explanatory and if not they will become so in the following.

In the main section the command line arguments are passed to a tiny library called CLP. This library implements a `CommandLineParser` whose intention is the extraction of the input's base-name from the command line. Note that all BETL structures and commands are embedded into the namespace `betl`. The base-name, i.e., the name provided on the command line without the final extension is returned by the method `GetBasename()`.

```
int main( int argc, char* argv[] ) {
// parse the command line
betl::clp::CommandLineParser clp;
clp.Parse( argc, argv );

// define the basename
const std::string basename = clp.GetBasename( );
```

The provided base-name serves as an argument for creating an object `input` of type `betl::inp::gmsh::Input`. The `gmsh`-namespace identifies the input as an input-parser for the *gmsh* mesh format. Refer to the gmsh-homepage for details on that file format. Note that the input type takes the number of nodes per element as a non-type template parameter. Hence, with this type only homogeneous meshes, i.e., meshes which contain only one element type, can be processed. Up to now, support for non-homogeneous meshes is not implemented. Possible choices of `nodes_per_element` are

- `3`: flat triangle
- `6`: quadratic triangle
- `4`: bilinear quadrilateral
- `8`: quadratic quadrilateral

```
// create the input
const std::size_t nodes_per_elem = 3;
typedef betl::inp::gmsh::Input< nodes_per_elem > input_t;
input_t input( basename );
```

When the input has been parsed we can proceed with the construction of the mesh. In principle, the mesh itself is nothing but a data container for storing all the elements the model consists of. Hence, the `Mesh` type features the `Element` type as a template argument. As already mentioned at the beginning of this example the mesh can be constructed via several input data types as long as the input provides a certain set of functionality. Please take a look at the doxygen documentation for more information about all the mandatory interfaces the input needs to provide.

```
// with the input, create the mesh
typedef betl::Element< nodes_per_elem > element_t;
typedef betl::Mesh< element_t > mesh_t;
mesh_t mesh( input );
```

Once the mesh is created we can pass it through an `Exporter` object in order to prepare an output to a vtu-file. Naturally, when dealing with numerical methods one would like to pass more data to the exporter than just the topological information but this is left to the following examples. Here, the initialisation of an `Exporter` is rather simple. The instance `exporter` is created with a reference to the mesh object. Afterwards the `Write`-method is called in order to perform the writing of the vtu-file. The resulting file is written to the working directory as `base-name.vtu`.

```
// instantiate the exporter object
typedef betl::output::vtu::Exporter< mesh_t > exporter_t;
exporter_t exporter( mesh );
exporter.Write( basename );
```

Note that the export can be also done via `stream` objects. For instance, the following code snippet writes the contents of the vtu-file directly to the standard output.

```
// write vtu-content directly to stream
std::cout << exporter;
```

With the export of a vtu-file this very first example is done. You can build the example via **`make betl_ex1`**. Once created, simply enter **`betl_ex1`** **`l_shape.msh`** *(the extension can be omitted)* at the prompt. If everything works, the output on screen should look similar to:

```
<<<---------------------------------------------------------------------
                             Parser(GMSH)
------------------------------------------------------------------------
   Version               = 2.2
   Format                = ASCII
   No. of phys. names    = 0
   No. of nodes          = 20
   No. of element types = 1
      No. of elements of type 'TRIA_3' = 36

------------------------------------------------------------------------
   Parsing finished after: 0.000497 sec
------------------------------------------------------------------->>>

<<<---------------------------------------------------------------------
                       Mesh(input)::constructor
             Mesh<E>::creating a Element<3>-template mesh
------------------------------------------------------------------------
                         Some mesh statistics
------------------------------------------------------------------------
  20 vertices created
  36 elements created
------------------------------------------------------------------->>>
```

In addition, in your working directory you should find a file being named as `l_shape.vtu`. The contents of this file can be visualised with tools like, e.g, ParaView or MayaVi. For more information on the file format itself please visit the vtk homepage. The file format's specifications are also available as pdf.

4.4 Example 2: The DofHandler concept

This example is meant to be a practical application of the theoretical preliminaries given in Section 2.2. This tutorial example is directly build upon the previous example stated in Section 4.3 and its complete source code can be found at `$BETL_ROOT/tutorial/example_2/main.cpp`.

In detail, this section deals with

- Defining a finite element space's basis
- Declaring dof handler types and defining the respective instances
- Debug output of the created degrees of freedom.

For the creation of the finite element space the two following header files need to be invoked:

```
#include "febasis/febasis.hpp"
#include "dof_handler/dof_handler.hpp"
```

We will create two bases for two different finite element spaces. One space will be made up the lowest order Lagrangian based functions. The other space will be based upon the lowest order edge functions. While the first space will be approximated discontinuously the latter one will represent a continuous approximation. The declarations of the bases are

```
// define a constant, discontinuous lagrange febasis
typedef FEBasis< element_t,
                 CONSTANT,
                 Discontinuous,
                 LagrangeTraits > feb_e3_const_disc_lagr_t;
// define a liner, discontinuous div-febasis
typedef FEBasis< element_t,
                 LINEAR,
                 Continuous,
                 EdgeDivTraits > feb_e3_lin_cont_div_t;
```

where `element_t` corresponds to the flat triangular element type `betl::Element<3>`. With the declaration of the FE basis the declaration and instantiation of the dof handler is straightforward

```
// define two dofhandler objects
typedef DoFHandler< feb_e3_const_disc_lagr_t > dofhandler_lagr_t;
typedef DoFHandler< feb_e3_lin_cont_div_t    > dofhandler_div_t;
// instantiate dofhandler objects
dofhandler_lagr_t dofhandler_lagr;
dofhandler_div_t  dofhandler_div;
```

For the complete setup of the finite element space it remains to distribute the degrees of freedom on a particular range of elements. Here, the degrees of freedom are distributed on the complete mesh

```
// get element iterators
typedef mesh_t::const_element_iterator const_iterator;
const_iterator begin = mesh.e_begin( );
const_iterator end   = mesh.e_end( );
// distribute dofs
dofhandler_lagr.distributeDoFs( begin, end );
dofhandler_div.distributeDoFs( begin, end );
```

That's it. The method `distributeDoFs` finalises the construction of the finite element space.

From time to time it might be helpful to display some debug data. This debug data might either be displayed directly on screen or via a file-stream into a textfile. In BETL, this output is usually performed by calling the overloaded stream operator (`operator<<`). Its use is illustrated in the following example:

```
// debug output (on screen)
std::cout << dofhandler_lagr << std::endl;
// debug output (to file)
const std::string filename = basename + "_DH_DIV_DEBUG.dat";
std::ofstream dh_div_out( filename.c_str( ) );
dh_div_out << dofhandler_div << std::endl;
dh_div_out.close( );
```

As for the former example this example can be build via **make betl_ex2**. When calling **betl_ex2** **tetrahedron_4.msh** the additional on-screen output should be:

```
<<<----------------------------------------------------------------------
                              DoFHandler
------------------------------------------------------------------------
  distribute dofs: 4 dofs created.
--------------------------------------------------------------------->>>

<<<----------------------------------------------------------------------
                              DoFHandler
------------------------------------------------------------------------
  distribute dofs: 6 dofs created.
--------------------------------------------------------------------->>>
================================================================
       LIST OF DEGREES OF FREEDOM AND THEIR SUPPORT
             (represented by element indices)
================================================================
Index = 0, Support = 0
Index = 1, Support = 1
Index = 2, Support = 2
Index = 3, Support = 3
================================================================
  LIST OF ELEMENTS WITH THEIR ASSOCIATED DEGREES OF FREEDOM
             (represented by dof indices)
================================================================
Ele-Idx = 0, Dofs = 0
Ele-Idx = 1, Dofs = 1
Ele-Idx = 2, Dofs = 2
Ele-Idx = 3, Dofs = 3
```

The output between the <<<- and the >>>-signs features the standard messages generated by the `distributeDoFs`-calls. For the given mesh, four degrees of freedom have been created by the Lagrangian dof handler type while the edge-based finite element space consists of six degrees of freedom. The remainder of the on-screen output is nothing but the Lagrangian dof handler's debug output. The debug output for the edge dof handler can be found in the generated textfile `tetrahedron_4_DH_DIV_DEBUG.dat`. Its content is given below.

```
================================================================
       LIST OF DEGREES OF FREEDOM AND THEIR SUPPORT
             (represented by element indices)
================================================================
Index = 0, Support = 0  3
Index = 1, Support = 0  2
Index = 2, Support = 0  1
Index = 3, Support = 1  2
Index = 4, Support = 1  3
Index = 5, Support = 2  3
================================================================
  LIST OF ELEMENTS WITH THEIR ASSOCIATED DEGREES OF FREEDOM
             (represented by dof indices)
================================================================
Ele-Idx = 0, Dofs = 0 1 2
Ele-Idx = 1, Dofs = 3 4 2
```

```
Ele-Idx = 2, Dofs = 3 1 5
Ele-Idx = 3, Dofs = 4 5 0
```

The output above concludes this section. With the dof handler at hand the general setup of the discretisation scheme is finished. Now we are ready to directly dive into the more boundary element specific topics. This will be done within the following examples.

4.5 Example 3: Generating Boundary Layer potentials

Welcome to Boundary Element Methods! Contrary to the former examples which dealt more or less with the setup of the discretisation model this example is the very first one which is devoted to the discretisation of Boundary Integral Operators. Again, we will utilise the former example from Section 4.4 to build an application which computes two discrete boundary layer potentials as they have been theoretically introduced in Section 1.4. Thereby, the discretised bilinear forms will correspond to those finite element spaces which have been introduced previously. The complete source code for this example may be found at `$BETL_ROOT/tutorial/example_3/main.cpp`.

In detail, the current example deals with

- Defining fundamental solutions
- Defining kernel functions
- Defining a quadrature rule
- Defining integrators
- Defining BEM operators

For example the bilinear corresponding to the Laplace equation's single layer potential reads as

$$\langle w, V_L u \rangle = \int_\Gamma \int_\Gamma w(x)\, U_L(y-x) u(y)\ ds_y ds_x \tag{4.1}$$

where we have skipped the trace operators for simplicity. The Galerkin discretisation of this bilinear form results in the matrix

$$\mathsf{V}_L[k,\ell] = \langle w_k, V_L u_\ell \rangle = \int_{\mathrm{supp}(w_k)} \int_{\mathrm{supp}(u_\ell)} K_{k\ell}(U_L, w_k, u_\ell)\ ds_y ds_x\ , \qquad K_{k\ell}(U_L, w_k, u_\ell) = w_k(x)\, U_L(y-x) u_\ell(y)\ . \tag{4.2}$$

The computation of matrices in form of V_L is what BETL is mainly about. To achieve this in a user convenient fashion, it tries to mimic the mathematical notation by respective data structures. For instance, the expression $K_{k\ell}$ in the equation above is represented by the data structure `GalerkinKernel`. As we will see, the declaration of this data type follows directly its mathematical definition.

To begin with, here are the additional headers which have to be included:

```
#include "traits/complextraits.hpp"
#include "fundsol/fundsol.hpp"
#include "kernel/galerkinkernel.hpp"
#include "integration/galerkinquadrature.hpp"
#include "integration/galerkinintegrator.hpp"
#include "bem_operator/bem_operator.hpp"
```

Following 4.2 we can conclude that we need to define the fundamental solution U_L. We do not need to define one of the functions w_k or u_ℓ since these functions already have been determined by the `FEBasis` and the `DoFHandler`. In BETL the fundamental solution is defined via two enumerators. While the first enumerator determines the underlying partial differential operator the second one defines which boundary potential layer should be instantiated. The following code

```
// declare two fundamental solution data types
typedef FundSol< LAPLACE  , SLP > fs_lapl_slp_t;
typedef FundSol< HELMHOLTZ, SLP > fs_helm_slp_t;
```

declares two fundamental solutions. The first one is nothing but the already-known fundamental solution for the Laplace equation's single layer potential, the second one corresponds to the single layer potential for the Helmholtz operator. Their definitions are given below

$$U_L(z) = \frac{1}{4\pi}\frac{1}{|z|}\,, \qquad U_H(z;\kappa) = \frac{1}{4\pi}\frac{\exp(-\kappa|z|)}{|z|}\,, \qquad z \in \mathbb{R}^3\,. \tag{4.3}$$

Note that the wave-number $\kappa \in \mathbb{C}$ is a complex number by what the evaluation result of the Helmholtz fundamental solution will be a complex number. Later in this example we will comment on how BETL treats complex numbers.

Now we can go ahead with the definition of *kernel*. In BETL the kernel is nothing but the concatenation of the test- and trial-functions together with the fundamental solution. Here is the declaration of two kernels

```
// declare laplace kernel based on the Lagrangian based fe space
typedef GalerkinKernel< fs_lapl_slp_t,
                        feb_e3_const_disc_lagr_t,
                        feb_e3_const_disc_lagr_t > kernel_lapl_t;
// declare helmholtz kernel based on the edge based fe space
typedef GalerkinKernel< fs_helm_slp_t,
                        feb_e3_lin_const_div_t,
                        feb_e3_lin_const_div_t > kernel_helm_t;
```

Clearly, the declarations of the kernel data types follow directly the kernel definition from 4.2.

In general, an analytic integration of the kernel function in 4.2 is impossible. Hence, in most instances we are forced to replace that integration by an appropriate quadrature rule. Contrary to the classical finite element schemes, the quadrature in boundary element methods is by no means straightforward. Due to the singularities of the kernel functions we need to distinct between these four different cases

- the distance of all points on two elements is in any case positive *(regular case)*
- two elements are identical *(coincident case)*
- two elements share a common edge *(edge adjacent case)*
- two elements share a common point *(vertex adjacent case)*.

A general quadrature which tackles those four different cases can be, e.g, found in [StrSchwb11]. The quadrature scheme presented there covers weak and strong singularities for triangular as well as for quadrilateral elements – whether these elements are curved or not. Hence, it represents the Swiss army knife for the evaluation of kernel functions in 3-dimensional Galerkin-based boundary element methods. Within BETL this special quadrature is the the default integration scheme and it is declared by the following code

```
// declare a quadrature rule
/*
 * template paramters:
 * type of tau_y, regular, coincident, edge-adjacent, vertex-adjacent rules
 * type of tau_x, regular, coincident, edge-adjacent, vertex-adjacent rules
 */
typedef GalerkinQuadrature< element_t, 7, 25, 16, 9
                            element_t, 7, 25, 16, 9 > quadrature_t;
```

The `GalerkinQuadrature` data type takes up to 10 template arguments. The first four template arguments specify the quadrature rule for the inner integration on τ_y. Beside the element type one has to prescribe the quadrature rules for the regular, coincident, the edge-adjacent, and for the vertex-adjacent quadrature. The remaining four template parameters specify the quadrature for the outer integration on τ_x. Typically, these quadrature rules equal those for the innter integration. Therefore, the last four template parameters are optional and can be omitted. For the regular integration on triangular elements BETL uses quadrature formulas as they are, e.g, given in [Dnvnt85]. Refer to gausstria.hpp for available quadrature rules on triangles. Independent of the element type, the quadrature for the singular integrations is based on tensor Gauss point rules. Therefore, the remaining pairs of integer numbers in GalerkinQuadrature need to be square numbers defining the Gaussian points on a quadrilateral reference element. The implemented tensor product rules are specified in the header gausstensor.hpp.

With a specific quadrature rule and a particular kernel type we are ready to declare the integration data type named Galerki-nIntegrator. These declarations are quite simple and read as

```
// declare the integrator types
typedef GalerkinIntegrator< kernel_lapl_t, quadrature_t > integrator_lapl_t;
typedef GalerkinIntegrator< kernel_helm_t, quadrature_t > integrator_helm_t;
```

Now, it remains to define instances of some of the above declared data types. The definition of the Helmholtz fundamental solution in 4.3 features the wave-number as a parameter. As already mentioned this parameter is in general a complex number. Since C++ offers no native complex data type there might be more than just one complex data type implementation around. Therefore, BETL offers access to complex data types via ComplexTraits. Note that the complex data type is chosen by the ComplexTraits with help of precompiler directives.

```
// define a wave number
typedef ComplexTraits< double >::complex_type complex_t;
const complex_t kappa( 0., 1. );
```

Now, what follows are the definitions of the fundamental solution, of the kernel, and of the integrator. Thereby, the former instance serves as a constructor argument for the latter one. The following source code illustrates this.

```
// create fundsol, kernel, and integrator instances
fs_lapl_slp_t      fs_lapl_slp;
fs_helm_slp_t      fs_helm_slp( kappa );
kernel_lapl_t      kernel_lapl( fs_lapl_slp );
kernel_helm_t      kernel_helm( fs_helm_slp );
integrator_lapl_t integrator_lapl( kernel_lapl );
integrator_helm_t integrator_helm( kernel_helm );
```

With the instantiation of the integrators we have finished the setup of the more BEM specific tasks. We are now able to perform the integration on two particular boundary elements τ_x and τ_y. In other words, the integrator is a functor which is responsible for the computation of one or of a couple of specific matrix entries. Since we aim at the computation of the complete system matrix V_L we need a structure which combines the DoFHandler *(which stores the information on how to assemble the degrees of freedom)* with the GalerkinIntegrator *(which actually is able to evaluate the matrix entries)*. Within BETL this structure is called BemOperator and its declaration as well as its instantiation is

```
// declare the discrete bem operators
typedef BemOperator< integrator_lapl_t,
                     dofhandler_lagr_t,
                     dofhandler_lagr_t > bem_operator_lapl_t;
typedef BemOperator< integrator_helm_t,
                     dofhandler_div_t,
                     dofhandler_div_t > bem_operator_helm_t;
// instantiate the discrete bem operators
bem_operator_lapl_t bem_operator_lapl( integrator_lapl,
                                       dofhandler_lagr, dofhandler_lagr );
bem_operator_helm_t bem_operator_helm( integrator_helm,
                                       dofhandler_div, dofhandler_div );
```

Finally, the boundary layer potentials are generated via the compute() method

```
// compute the boundary potentials
bem_operator_lapl.compute( );
bem_operator_helm.compute( );
```

While the command `bem_operator_lapl.compute( )` computes the single layer potential for the Laplace equation as it is stated in 4.2 the command `bem_operator_helm.compute( )` evaluates the following bilinear form

$$\mathsf{V}_H[k,\ell] = \langle \mathbf{w}_k, V_H \mathbf{u}_\ell \rangle = \int\limits_{\mathrm{supp}(\mathbf{w}_k)} \int\limits_{\mathrm{supp}(\mathbf{u}_\ell)} K_{k\ell}(U_H, \mathbf{w}_k, \mathbf{u}_\ell)\ ds_y ds_x\ , \qquad K_{k\ell}(U_H, \mathbf{w}_k, \mathbf{u}_\ell) = \mathbf{w}_k(x) \cdot (U_H(y-x)\mathbf{u}_\ell(y))\ . \tag{4.4}$$

Note that the test- and trial-space in 4.4 are based on 2.3.

The generations of the discrete boundary layer potentials V_L and V_H almost conclude this section. But as in the previous section, for debug purposes it is necessary to allow the boundary potentials to be exported either directly to the screen or to a file. The way the output of those potentials is done equals the output of the dof handler objects in the previous section. We simply apply the stream operator to the underlying matrix structures of the respective bem operators. Here is the code for getting the references to the generated matrices.

```
// get references to the created matrix structures
bem_operator_lapl_t::const_reference V_lapl = bem_operator_lapl.giveMatrix( );
bem_operator_helm_t::const_reference V_helm = bem_operator_helm.giveMatrix( );
```

With the references `V_lapl` and `V_helm` the output of the potentials can be done in the following way:

```
// write matrices to files stream
const std::string fname_lapl = basename + "_LAPL.dat";
const std::string fname_helm = basename + "_HELM.dat";
std::ofstream out_lapl( fname_lapl.c_str() );
std::ofstream out_helm( fname_helm.c_str() );
out_lapl << V_lapl;
out_helm << V_helm;
out_lapl.close( );
out_helm.close( );
```

Finally, building this example via **`make betl_ex3`** and calling **`betl_ex3 tetrahedron_4.msh`** produces the two output files `tetrahedron_4_LAPL.dat` and `tetrahedron_4_HELM.dat`, respectively. Those files can then be, e.g., directly processed with GNU Octave. The additional on-screen output for this example shows the constructor messages of the `BemOperator` and some statistics for its `compute()` method.

```
<<<-----------------------------------------------------------------------
                              BemOperator
--------------------------------------------------------------------------
  dimension:                  4 x 4
  symmetry:                   symmetric
  (dense) mem. consumption: 7.62939e-05 MB
  precision:                  Double precision
  numerical type:             d
  acceleration method:        NO_ACCELERATION
----------------------------------------------------------------------->>>

<<<-----------------------------------------------------------------------
                              BemOperator
--------------------------------------------------------------------------
  dimension:                  6 x 6
  symmetry:                   symmetric
  (dense) mem. consumption: 0.000320435 MB
  precision:                  Double precision
  numerical type:             4compIdE
  acceleration method:        NO_ACCELERATION
----------------------------------------------------------------------->>>

<<<-----------------------------------------------------------------------
                        BemOperator::compute()
--------------------------------------------------------------------------
```

```
  computational time:     0.001672 sec
  performed integrations: 16
  saved integrations:     0
-------------------------------------------------------------------->>>

<<<--------------------------------------------------------------------
                        BemOperator::compute()
-----------------------------------------------------------------------
  computational time:     0.006681 sec
  performed integrations: 16
  saved integrations:     0
-------------------------------------------------------------------->>>
```

The output may differ on your system since the numerical type is queried via the `typeid` function from the `typeinfo` header. Moreover, the computational times will surely also differ on your system.

Now, we will have a closer look on the computational results. The entries of V_L are stored in the file `tetrahedron_4_LAPL.dat` with its content given below.

```
   1.8546e-01   7.4552e-02   7.4552e-02   7.4552e-02
   7.4542e-02   7.9824e-02   3.9244e-02   3.9250e-02
   7.4542e-02   3.9250e-02   7.9824e-02   3.9244e-02
   7.4542e-02   3.9244e-02   3.9250e-02   7.9824e-02
```

Since the bilinear form in 4.1 is symmetric and since we apply a symmetric Galerkin discretisation to it we would assume the system matrix to be symmetric as well. Unfortunately, this is not the case in this example. As one already might assume, this is due to the singular integrations. The mesh contains four elements which build a tetrahedron. Thus, the integration routines which are called are only the singular integration schemes for the coincident and edge adjacent cases. In this example the coincident integrations are only called to compute the diagonal entries, all other entries are evaluated by the edge adjacent integration routines. And these edge adjacent integration schemes rely on unsymmetric quadrature schemes. In order to allay this phenomenon we will increase the number of Gaussian points. In doing so we crack a nut with a sledgehammer and prescribe the following quadrature.

```
// declare a quadrature rule
typedef GalerkinQuadrature< element_t, element_t,
                            7, 7,
                            25, 25,
                            64, 64,    // sledgehammer
                            9,  9
                            > quadrature_t;
```

A calculation with the recompiled code reveals the following result for the Laplace equation's single layer potential

```
   1.8546e-01   7.4551e-02   7.4551e-02   7.4551e-02
   7.4551e-02   7.9824e-02   3.9251e-02   3.9251e-02
   7.4551e-02   3.9251e-02   7.9824e-02   3.9251e-02
   7.4551e-02   3.9251e-02   3.9251e-02   7.9824e-02
```

Now, the matrix looks fine. Nevertheless, the number of 64 Gaussian points is ridiculous. However, this example already illustrates the sensitivity of the integration's accuracy with respect to the chosen quadrature rule. We will comment on the integration routines in more detail in one of the upcoming examples.

One final comment has to be given on the fact that obviously all entries of the matrix V_L have been evaluated and that no advantage has been taken out of the bilinear form's symmetry properties. Boundary element methods rely on non-local operators resulting in fully populated system matrices. Both, in terms of memory consumption as well as in terms of computational costs this leads to a quadratic complexity of the algorithm. For practical applications this complexity cannot be tolerated. Thus, we need to incorporate the so-called fast boundary element methods in order to tackle real world problems. From this point of view the dense matrices which have been created here can be considered only as some kind of 'debug matrices'. For instance, they may serve to check some of the system's properties or they may be used to validate the fast boundary element methods which will be introduced later. Therefore, it simply does not make sense to take advantage of the bilinear form's symmetry at this stage.

[Dnvnt85] D.A. Dunavant, "High degree efficient symmetrical Gaussian quadrature rules for the triangle"International Journal for Numerical Methods in Engineering, 21, 1985.

[GlEtAl03] L. Gaul, M. Kögl, and M. Wagner, Boundary element methods for engineers and scientists: an introductory course with advanced topics, Springer, 2003.

[Hckbsch89] W. Hackbusch, Integralgleichungen: Theorie und Numerik, B.G. Teubner, 68, 1989.

[HsWdlnd08] G.C. Hsiao and W.L. Wendland, Boundary integral equations, Springer, 2008.

[McLn00] W.C.H. McLean, Strongly elliptic systems and boundary integral equations, Cambridge University Press, 2000.

[Stnbch08] O. Steinbach, Numerical approximation methods for elliptic boundary value problems: finite and boundary elements, Springer, 2008.

[StrSchwb11] S. Sauter and C. Schwab, Boundary Element Methods, Springer, Copyright © 2011 Springer-Verlag Berlin Heidelberg, ISBN 978-3-540-68092-5, DOI 10.1007/978-3-540-68093-2.

[StrdhrEtAl08] A. Sutradhar, G.H. Paulino, and L.J. Gray, Symmetric Galerkin Boundary Element Method, Springer, 2008.