On single- and multi-trace implementations for scattering problems with BETL

L. Kielhorn, ETH Zürich, Seminar of Applied Mathematics

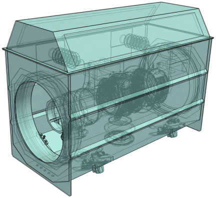
Söllerhaus Workshop 2012

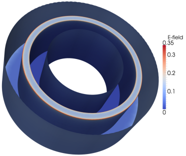
Why a BEM library?

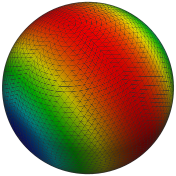
- The mathematician: Boundary integral equations (BIEs) are an indispensable tool for the analysis of linear PDEs and their BVPs. Lovely fractional Sobolev spaces!

Why a BEM library?

- The mathematician: Boundary integral equations (BIEs) are an indispensable tool for the analysis of linear PDEs and their BVPs. Lovely fractional Sobolev spaces!
- The application engineer: BIE-discretisation schemes are of interest for a couple of real-world problems.

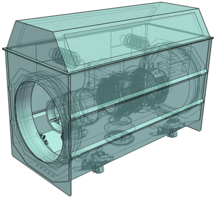


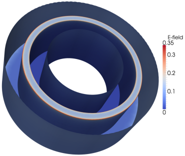


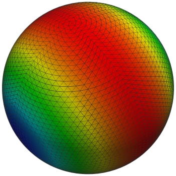


Why a BEM library?

- The mathematician: Boundary integral equations (BIEs) are an indispensable tool for the analysis of linear PDEs and their BVPs. Lovely fractional Sobolev spaces!
- The application engineer: BIE-discretisation schemes are of interest for a couple of real-world problems.



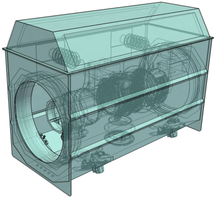


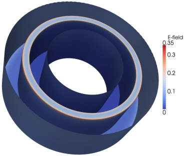


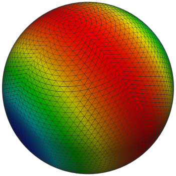
- The PhD-student: Come on, implementing Boundary Element Methods is cumbersome, annoying, tedious and error proning. It does not pay off!

Why a BEM library?

- The mathematician: Boundary integral equations (BIEs) are an indispensable tool for the analysis of linear PDEs and their BVPs. Lovely fractional Sobolev spaces!
- The application engineer: BIE-discretisation schemes are of interest for a couple of real-world problems.







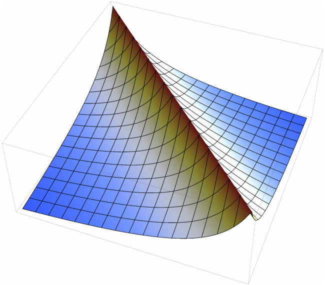
- The PhD-student: Come on, implementing Boundary Element Methods is cumbersome, annoying, tedious and error proning. It does not pay off!
- BETL aims to save the PhD-student, to support the engineer with rapid developments of new BEMs, and to please the mathematician (students have more time to focus on math)

Contents

BETL's one and only purpose

Compute something like

$$A[i,j] = \int_{\mathrm{supp}(\phi_i)} \phi_i(\mathbf{x}) \int_{\mathrm{supp}(\psi_j)} G(\mathbf{y}-\mathbf{x})\, \psi_j(\mathbf{y})\, \mathrm{d}s_{\mathbf{y}}\, \mathrm{d}s_{\mathbf{x}}$$

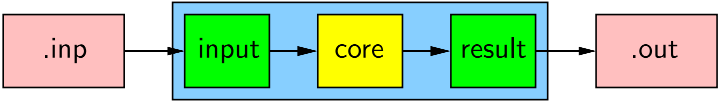


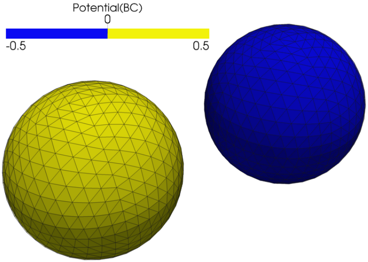
- BI operators are non-local, "Everything is connected with everything!"
- $G \approx \frac{1}{|\mathbf{y}-\mathbf{x}|}$ is rational and singular for $\mathbf{y} \to \mathbf{x}$. At least, here all the beauty of BIOs is lost!

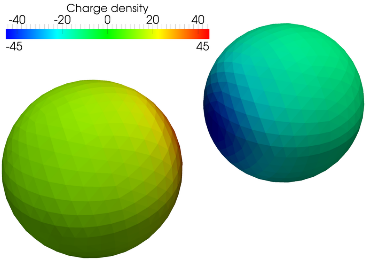
The workflow library-driven BEM Applications

- There exists a zoo of different BEMs $\Longrightarrow$ Avoid the all-in-one solution
- Write specific applications utilizing three main libraries:

  input, core, result

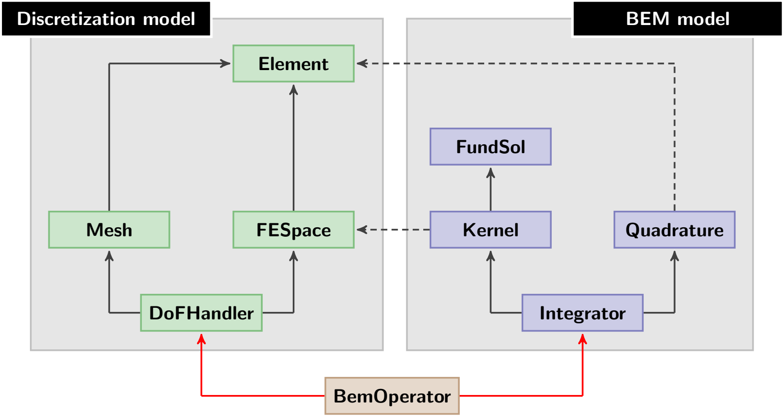






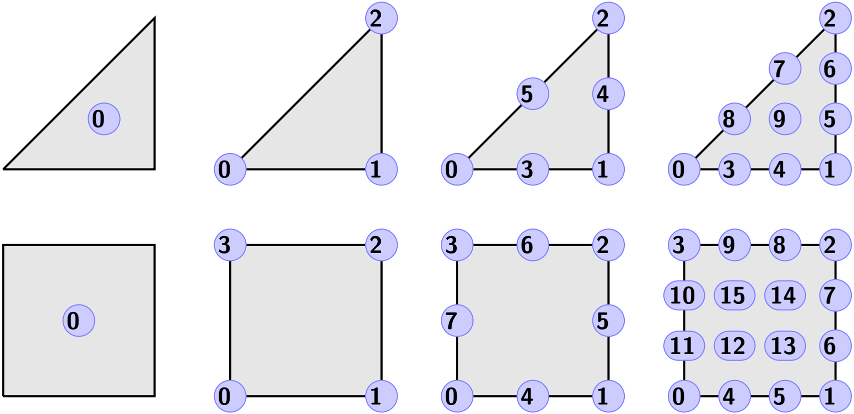
The main design of the core library

$$A[i,j] = \int_{\mathrm{supp}(\phi_i)} \phi_i(\mathbf{x}) \int_{\mathrm{supp}(\psi_j)} G(\mathbf{y}-\mathbf{x})\, \psi_j(\mathbf{y})\, \mathrm{d}s_{\mathbf{y}}\, \mathrm{d}s_{\mathbf{x}}$$



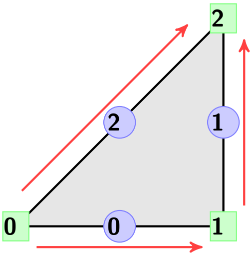
The Finite Element Basis

- Lagrangian basis functions (*Hat functions*)



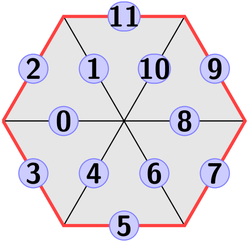
The Finite Element Basis (cont'd)

- *(Lowest order)* Edge functions

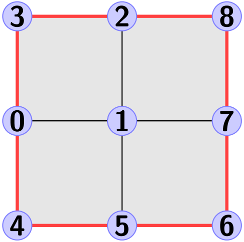


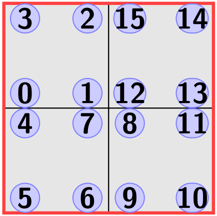
The Dofhandler concept

- On basis of the Finite Element basis distribute the dofs
- Distribution of edge dofs



- Distribution of Lagrangian dofs (continuous/discontinuous)





The design criteria of a BEM library

- Guarantee a robust and efficient runtime behavior!
- Develop flexible and easy-to-use interfaces!
- No redundancies. Implement things only once!
- Make use of well established libraries like, e.g., STL, BOOST, MKL, SUPERLU, . . . !
- Separate data-structures from algorithms!
- Encapsulate data, i.e., avoid global variables!
- Make use of dynamic memory management!

$\Longrightarrow$ **Use C++. Exploit the C++-Template-Mechanism**

BETL in action – Compute the Single Layer Potential

```
// define the element type and instantiate the mesh
Mesh< element_t > mesh( input );
```

BETL in action – Compute the Single Layer Potential

```
// define the element type and instantiate the mesh
Mesh< element_t > mesh( input );

// define the boundary element basis
typedef FEBasis< element_t,LINEAR,Discontinuous,LagrangeTraits > slp_basis_t;
```

BETL in action – Compute the Single Layer Potential

```
// define the element type and instantiate the mesh
Mesh< element_t > mesh( input );

// define the boundary element basis
typedef FEBasis< element_t,LINEAR,Discontinuous,LagrangeTraits > slp_basis_t;

// the dofhandler type and its instance
typedef DoFHandler< basis_t > dofhandler_t;
dofhandler_t dof_handler;
dof_handler.distributeDoFs( mesh.e_begin(), mesh.e_end() );
```

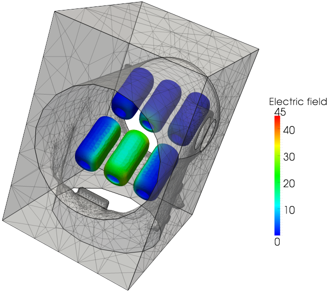
BETL in action – Compute the Single Layer Potential

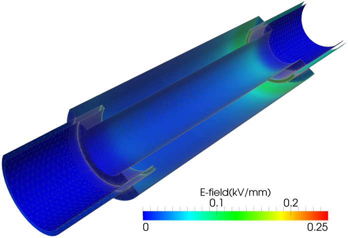
```
// define the element type and instantiate the mesh
Mesh< element_t > mesh( input );

// define the boundary element basis
typedef FEBasis< element_t,LINEAR,Discontinuous,LagrangeTraits > slp_basis_t;

// the dofhandler type and its instance
typedef DoFHandler< basis_t > dofhandler_t;
dofhandler_t dof_handler;
dof_handler.distributeDoFs( mesh.e_begin(), mesh.e_end() );

// the fundamental solution type and its instance
typedef FundSol< LAPLACE, SLP > fs_t;
fs_t fs;
// the kernel type and its instance
typedef GalerkinKernel< fs_t, dofhandler_t::FunctionType > kernel_t;
kernel_t kernel( fs );
// the integrator type and its instance
typedef GalerkinIntegrator< kernel_t, QuadratureRule<1,2> > integrator_t
integrator_t integrator( kernel )
```

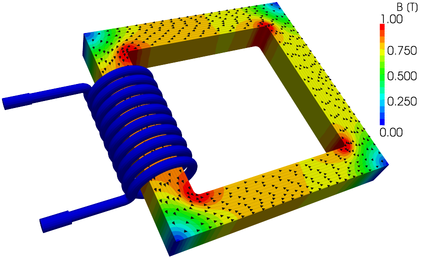
BETL in action – Compute the Single Layer Potential

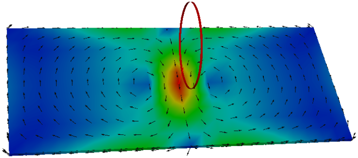
```
// define the element type and instantiate the mesh
Mesh< element_t > mesh( input );

// define the boundary element basis
typedef FEBasis< element_t,LINEAR,Discontinuous,LagrangeTraits > slp_basis_t;

// the dofhandler type and its instance
typedef DoFHandler< basis_t > dofhandler_t;
dofhandler_t dof_handler;
dof_handler.distributeDoFs( mesh.e_begin(), mesh.e_end() );

// the fundamental solution type and its instance
typedef FundSol< LAPLACE, SLP > fs_t;
fs_t fs;
// the kernel type and its instance
typedef GalerkinKernel< fs_t, dofhandler_t::FunctionType > kernel_t;
kernel_t kernel( fs );
// the integrator type and its instance
typedef GalerkinIntegrator< kernel_t, QuadratureRule<1,2> > integrator_t
integrator_t integrator( kernel )

// the type of the matrix generator and an instance
typedef DiscreteOperator< integrator_t, dofhandler_t > discrete_operator_t;
discrete_operator_t discrete_operator( integrator, dof_handler );
// finally, this computes the matrix A
discrete_operator.compute( );
```

BETL Applications

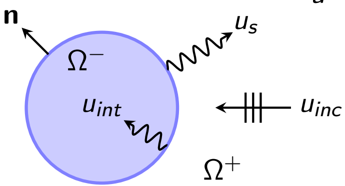








Transmission problem for acoustic scattering



$u = u_{int}$ in Ω^-, $u = u_s + u_{inc}$ in Ω^+

$$
\begin{aligned}
-\Delta u - \omega_\sigma^2 u &= 0 && \text{in } \Omega^\sigma \\
[\![\gamma_D u]\!] &= 0 && \text{on } \Gamma \\
[\![\gamma_N u]\!] &= 0 && \text{on } \Gamma
\end{aligned}
$$

+radiation condition for u_s

- Calderón operator:

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

- BIEs

$$\Omega^-: \qquad \left(-\tfrac{1}{2}I + A_{\omega_-}\right)\begin{pmatrix} \gamma_D^- u \\ \gamma_N^- u \end{pmatrix} = 0$$

$$\Omega^+: \qquad \left(-\tfrac{1}{2}I - A_{\omega_+}\right)\begin{pmatrix} \gamma_D^+ u_s \\ \gamma_N^+ u_s \end{pmatrix} = 0$$

Acoustic scattering: Boundary Integral representations

- BIEs

$$\left(-\tfrac{1}{2}I + A_{\omega_-}\right)\begin{pmatrix}\gamma_D^- u \\ \gamma_N^- u\end{pmatrix} = 0$$

$$\left(-\tfrac{1}{2}I - A_{\omega_+}\right)\begin{pmatrix}\gamma_D^- u \\ \gamma_N^- u\end{pmatrix} = -\begin{pmatrix}\gamma_D^- u_{inc} \\ \gamma_N^- u_{inc}\end{pmatrix}$$

- Subtract exterior BIEs from interior BIEs (1st kind)

$$\left(A_{\omega_-} + A_{\omega_+}\right)\begin{pmatrix}\gamma_D^- u \\ \gamma_N^- u\end{pmatrix} = \begin{pmatrix}\gamma_D^- u_{inc} \\ \gamma_N^- u_{inc}\end{pmatrix}$$

- Add exterior and interior BIEs (2nd kind)

$$\left(I - A_{\Delta\omega}\right)\begin{pmatrix}\gamma_D^- u \\ \gamma_N^- u\end{pmatrix} = \begin{pmatrix}\gamma_D^- u_{inc} \\ \gamma_N^- u_{inc}\end{pmatrix}, \quad \Delta\omega = \omega_- - \omega_+$$

On the discretisation of $\tilde{A} = A_{\omega_-} + A_{\omega_+} \wedge \tilde{A} = I - A_{\Delta\omega}$

- Galerkin scheme

$$\langle \tilde{A} \begin{pmatrix} \gamma_D^- u \\ \gamma_N^- u \end{pmatrix}, \begin{pmatrix} \psi \\ \varphi \end{pmatrix} \rangle = \langle \begin{pmatrix} \gamma_D^- u_{inc} \\ \gamma_N^- u_{inc} \end{pmatrix}, \begin{pmatrix} \psi \\ \varphi \end{pmatrix} \rangle$$

- Test- and trial-spaces may differ for 1st kind and 2nd kind formulation
- Single layer and double layer operators are in place.
- But: What's about an efficient implementation of the hypersingular operator for the Helmholtz kernel?
  - The hypersingular kernels in A_{ω_-} and A_{ω_+} demand a realisation via integration by parts
  - The hypersingular operator in $A_{\Delta\omega}$ is not hypersingular. Can be implemented via a classical approach.

The hypersingular operator (needed for 1st kind form.)

- Continuous representation

$$\langle D_\omega u, w \rangle = \int_\Gamma \int_\Gamma G_\omega(\mathbf{y}-\mathbf{x})\, \mathbf{curl}_{\Gamma,\mathbf{y}}\, u \cdot \mathbf{curl}_{\Gamma,\mathbf{x}}\, w \, \mathrm{d}s_{\mathbf{y}}\, \mathrm{d}s_{\mathbf{x}} - \omega^2 \int_\Gamma \int_\Gamma G_\omega(\mathbf{y}-\mathbf{x})\, u\, w\, \mathbf{n}_{\mathbf{y}} \cdot \mathbf{n}_{\mathbf{x}} \, \mathrm{d}s_{\mathbf{y}}\, \mathrm{d}s_{\mathbf{x}}$$

- Discrete form for lowest order function spaces

$$D_h = \sum_{i=1}^{3} C_i B V_h B^T C_i^T - \omega^2 A \left(\sum_{i=1}^{3} N_i V_h N_i^T \right) A^T$$

Needed FE-spaces in BETL:

```
// pw linear discontinuous space    :: V, B, N
typedef FEBasis<Element<3>,LINEAR  ,Discontinuous,LagrangeTraits> slp_fes_t;
// pw constant space                :: B, C
typedef FEBasis<Element<3>,CONSTANT,Discontinuous,LagrangeTraits> const_fes_t;
// pw linear continuous space       :: C, A
typedef FEBasis<Element<3>,LINEAR  ,Continuous   ,LagrangeTraits> lin_fes_t;
```

Creating the discrete operators V_h, C, N, B, A

- Recalling the discrete form

$$D_h = \sum_{i=1}^{3} C_i B V_h B^T C_i^T - \omega^2 A \left(\sum_{i=1}^{3} N_i V_h N_i^T \right) A^T$$

- Dofhandler types

```
typedef DoFHandler< slp_fes_t   > slp_dh_t;
typedef DoFHandler< const_fes_t > const_dh_t;
typedef DoFHandler< lin_fes_t   > lin_dh_t;
```

- With an integrator type the bem-operator's definition is

```
typedef DiscreteOperator< integrator_t, slp_dh_t > slp_operator_t;
```

- ... and the sparse operators' definitions read

```
typedef curl_operator      < const_fes_t, lin_fes_t > curl_op_t;
typedef normal_operator    < slp_fes_t  , slp_fes_t > normal_op_t;
typedef adjacency_operator< const_fes_t, slp_fes_t > B_op_t;
typedef adjacency_operator< lin_fes_t  , slp_fes_t > A_op_t;
```

- Next step: Create instances and perform the computations

```
slp_operator_t slp_operator( integrator, slp_dh );
B_op_t         B_op        ( const_dh  , slp_dh );
slp_operator.compute( );
B_op.compute( ); // ...and do the same for all other operators!
```

What—in fact— has to be done...

- Once the operators have been computed the discrete Calderón operator is given by

$$A_h(V_h, K_h) = \begin{bmatrix} -K_h & BV_hB^T \\ D_h(V_h) & K_h^\top \end{bmatrix}$$

- BETL provides methods to build block system out of matrices or blocks of matrices
- However, BETL encapsulates the Calderón operator in a simple structure

```
// declare calderon operator type
typedef driver::helmholtz::CalderonOperator< NO_ACCELERATION > calderon_op_t;
// create instances of calderon operators
calderon_op_t calderon_ext( omega_ext );
calderon_op_t calderon_int( omega_int );
// initialize it with element iterators
calderon_ext.initialize( begin, end );
calderon_int.initialize( begin, end );
// compute them
calderon_ext.compute( );
calderon_int.compute( );
```

Notes on the 2nd kind formulation

- 2nd kind formulation demands the implementation of new kernels for $V_{\Delta\omega}$, $K_{\Delta\omega}$, and $D_{\Delta\omega}$

```
// skeleton for the implementation of V_\Delta\omega
class FundSol< HELMHOLTZ_DIFF, SLP > {
  FundSol( complex_t omega_1, complex_t omega_0 ) { /* ... */ }
  void operator()( const ArgRad r, const ArgGP X, const ArgGP Y,
                   ResultType& result )
  { /* your implementation goes here... */ }
};
```

- Now you can use it in the same way as the built-in functions

```
// define and instantiate the modified Green's function
typedef FundSol< HELMHOLTZ_DIFF, SLP > fs_t;
fs_t fs( omega_int, omega_ext );
// declare a GalerkinKernel in the same way as before
typedef GalerkinKernel< fs_t, dofhandler_t::FunctionType > kernel_t;
```

- Naturally, everything can be encapsulated in a simple data structure again

```
// declare calderon operator type
typedef driver::helmholtz_diff::CalderonOperator< ACA > calderon_op_t;
// create instance
calderon_op_t calderon( omega_int, omega_ext );
// initialize it with element iterators
calderon.initialize( begin, end );
// compute it
calderon.compute( );
// ...compute mass matrix -> glue everything together...
```

Enhancing the scheme...

- The first kind formulation

$$\left(A_{h,\omega_-} + A_{h,\omega_+}\right) \begin{bmatrix} \underline{u} \\ \underline{t} \end{bmatrix} = \begin{bmatrix} M_{ND} & 0 \\ 0 & M_{DN} \end{bmatrix} \begin{bmatrix} \underline{u}_{inc} \\ \underline{t}_{inc} \end{bmatrix}$$

is just a special case of the classical Single Trace Formulation

$$\sum_{d=0}^{D} L_d^* A_{h,\omega_d} L_d \begin{bmatrix} \underline{u} \\ \underline{t} \end{bmatrix} = L_{ext}^* M \begin{bmatrix} \underline{u}_{inc} \\ \underline{t}_{inc} \end{bmatrix}$$

- L_d are localisation operators

$$L_d : \text{dofs on } \Gamma \quad \rightarrow \quad \text{dofs on } \Gamma_d$$

- Thanks to their sparsity the implementation of the Localisation operators can be easily done. BETL provides generic routines for creating sparse matrices.

A final enhancement

- Multi-trace formulations usually reveal the following matrix structure

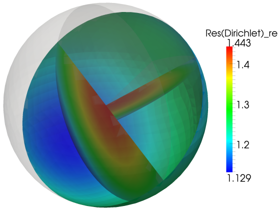
$$\left(\operatorname{diag}(A_{\omega_d}) + \begin{bmatrix} A^{00}_{\omega_{ext}} & \frac{1}{2}M^{01} + A^{01}_{\omega_{ext}} \\ \frac{1}{2}M^{10} + A^{10}_{\omega_{ext}} & A^{11}_{\omega_{ext}} \end{bmatrix}\right) \begin{bmatrix} \begin{bmatrix} \underline{u}^0 \\ \underline{t}^0 \end{bmatrix} \\ \begin{bmatrix} \underline{u}^1 \\ \underline{t}^1 \end{bmatrix} \end{bmatrix} = \underline{f}(u_{inc})$$

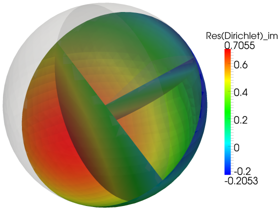
- What extra work has to be done from an implementation point of view? In fact not much:

```
// declare calderon operator type
typedef driver::helmholtz::CalderonOperator< ACA > calderon_op_t;
// create instance
calderon_op_t A_01( omega_ext );
// initialize it with different element iterators for test- and trial-spaces
A_01.initialize( begin_test , end_test,
                 begin_trial, end_trial );
// compute it!
A_01.compute( );
// do the same with the mass matrix
typedef identity_operator< dofhandler_test_t, dofhandler_trial_t > id_op_t;
id_op_t M_01( dofhandler_test, dofhandler_trial );
```

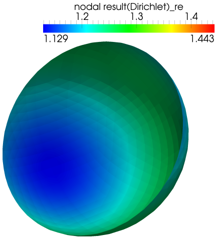
Tiny case study: $\omega_{ext} = 1$, $\omega_{int} = 2$, $r_{sphere} = 0.5$

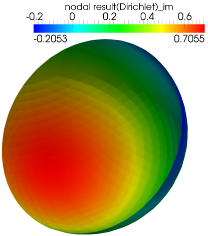
- Single trace formulation results (#Elements: 3648)





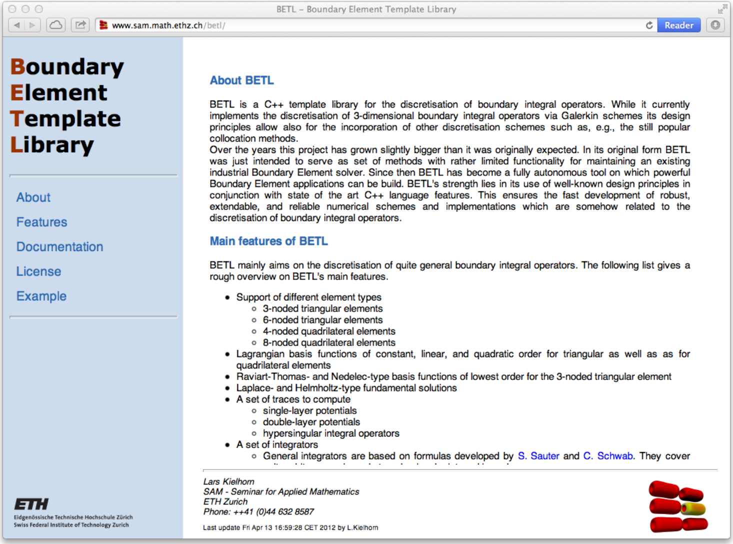
- 2nd kind formulation results (#Elements: 2048)





BETL's homepage

- Visit BETL at: `www.sam.math.ethz.ch/betl`



BETL – Boundary Element Template Library

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

Reader

Boundary Element Template Library

About

Features

Documentation

License

Example

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

About BETL

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. Since then 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 fast development of robust, extendable, and reliable numerical schemes and implementations which are somehow related to the discretisation of boundary integral operators.

Main features of BETL

BETL mainly aims on the discretisation of quite general boundary integral operators. The following list gives a rough overview on BETL's main features.

- Support of different element types
  - 3-noded triangular elements
  - 6-noded triangular elements
  - 4-noded quadrilateral elements
  - 8-noded quadrilateral elements
- Lagrangian basis functions of constant, linear, and quadratic order for triangular as well as as for quadrilateral elements
- Raviart-Thomas- and Nedelec-type basis functions of lowest order for the 3-noded triangular element
- Laplace- and Helmholtz-type fundamental solutions
- A set of traces to compute
  - single-layer potentials
  - double-layer potentials
  - hypersingular integral operators
- A set of integrators
  - General integrators are based on formulas developed by S. Sauter and C. Schwab. They cover

Lars Kielhorn
SAM - Seminar for Applied Mathematics
ETH Zurich
Phone: ++41 (0)44 632 8587

Last update Fri Apr 13 16:59:28 CET 2012 by L.Kielhorn

Conclusion

- BETL is an efficient, modular, extendable and an easy-to-use BEM library
- BETL provides Laplace- and Helmholtz-type fundamental solutions
- BETL provides flat/curved triangles/quadrilaterals
- BETL provides constant, linear, and quadratic FE-Spaces for Nodal based Functions (continuous/discontinuous)
- BETL provides lowest order edge-elements for, e.g., Eddy Current simulations (continuous/discontinuous)
- BETL provides preconditioners for the most common integral operators
  - Operator preconditiong via dual meshes (Implementation for Lagrangian FE-spaces is finished. Implementation for Edge based FE-spaces is *almost* finished)
  - ABPX (Artificial Bramble-Pasciak-Xu) for the Laplacian single layer potential (G.Of)

Conclusion

- BETL is interfaced with Fast Boundary Element Methods
  - BETL utilizes AHMED's parallelism (OpenMP,MPI) [T.Klug, TU Munich]
  - BETL is interfaced with classical Fast Multipole Methods (FMM) [by G.Of, TU Chemnitz/Graz] *(experimental)*
  - BETL is interfaced with a directional FMM [by M.Messner & E.Darve] *(experimental)*
- BETL provides a set of integrators (complete generic integrators as well as semi-analytic integrators)
- BETL has been tested with gnu & Intel compilers
- BETL utilises cmake as a build system: Linux, MacOS X & Windows
- BETL relies on well tested open-source libraries

Conclusion

- Up to now BETL has been applied to
  - Electrostatic problems
  - Magnetostatic problems
  - Optimization problems
  - Eddy current problems
  - Single-/Multi-trace formulations for the Helmholtz operator
- What the BETL does not offer?
  - No n-d discretizations of BI operators (with $n \neq 3$)
  - No collocation schemes
  - No evaluations of representation formulae
  - No adaptive integrators (quasi-singular kernels!)
  - No support for heterogeneous meshes
  - No adaptivity at all (e.g., *hp*-BEM demands modifications on the 'model analysis', i.e., modifications on the dofhandler)

Outlook

- Improve the documentation
- Improve the test routines
- Apply BETL to optimization problems
- Apply BETL to real-world problems again
- Implement stable quadrature schemes [based on work of C.Schwab]
- Implement higher order spaces for edge-functions
- Improve the MPI-parallelisation (load-balancing!)
- Incorporate NURBS as FE-Space (isogeometric-approach)