Tutorial

K. Mang, J. Kratzke, S. Gawlok, M. Baumann

Error Estimation on Convex Bent Domains for the Poisson Equation

modified on February 8, 2016

Version 1.6

Contents

Error Estimation on Convex Bent Domains for the Poisson Equation

1 Introduction

HiFlow^3 is a multi-purpose Finite Element software providing powerful tools for efficient and accurate solution of a wide range of problems modeled by partial differential equations (PDEs). Based on object-oriented concepts and the full capabilities of C++ the HiFlow^3 project follows a modular and generic approach for building efficient parallel numerical solvers. It provides highly capable modules dealing with the mesh setup, Finite Element spaces, degrees of freedom, linear algebra routines, numerical solvers, and output data for visualization. Parallelism - as the basis for high performance simulations on modern computing systems - is introduced on two levels: coarse-grained parallelism by means of distributed grids and distributed data structures, and fine-grained parallelism by means of platform-optimized linear algebra back-ends.

1.1 How to Use the Tutorial?

You find the example code (poisson_adaptive.cc, poisson_adaptive.h), a parameter file for the first numerical example (poisson_adaptive.xml) and a Makefile, which you only need when using HiFlow^3 as a library (see 1.1.1), in the folder `/hiflow/examples/poisson`. The geometry data (*.inp, *.vtu) is stored in the folder `/hiflow/examples/data`.

1.1.1 Using HiFlow^3 as a Library

First install HiFlow^3. Therefore, follow the instructions listed on `http://www.hiflow3.org/`, see "Documentation"-"Installation". To compile and link the tutorial correctly, you may have to adapt the Makefile depending on the options you chose in the cmake set up. Make sure that the variable `HIFLOW_DIR` is set to the path, where HiFlow^3 was installed. The default value ist `/usr/local`. When you set the option `WITH_METIS` to `ON` in the cmake set up, you have to make sure to link to the metis library in the Makefile (`www.cs.umn.edu/~metis`). `-lmetis` must be added to the end of the line in the Makefile, where the target `poisson_adaptive` is build (see second option in the Makefile, which is marked as a comment). By typing `make` in the console, in the same folder where the source-code and the Makefile is stored, you compile and link the tutorial. To execute the tutorial, type **./poisson_adaptive** `/"path_to"/poisson_tutorial.xml /"path_to_mesh_data"/` . This tutorial is currently implemented in 2D.

1.1.2 Using HiFlow^3 as a Developer

First build and compile HiFlow^3. Go to the directory `/build/example/poisson`, where the binary **poisson_adaptive** is stored. Type **./poisson_adaptive**, to execute the program. You need to make sure that the default parameterfile poisson_adaptive.xml is stored in the same directory as the binary, and that the geometry data specified in the parameter file is stored in `/hiflow/examples/data`. Alternatively, you can specify the path of your own xml-file with the name of your xml-file (first) and the path of your geometry data (second) in the comment

line, i.e. **./poisson_adaptive** `/"path_to_parameterfile"/"name_of_parameterfile".xml /"path_to_geometry_data"/`.

2 Mathematical Setup

2.1 Problem

Note: For simplification, we explain the mathematical setup and the examples only for the two dimensional case. However, it is easy to extend the theory and the implementation to one or three dimensions.
Our aim is to solve the Poisson problem with a homogeneous Dirichlet boundary condition on a domain $\Omega \subset \mathbb{R}^2$ with sufficiently smooth but convexly or concavely bent parts of the boundary $\partial\Omega$. With this background we want to find a computable upper bound for the error of the Finite

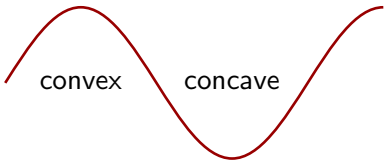

Figure 1: convexly and concavely bent boundary

Element method on domains with bent boundaries. As a next step we will test and evaluate the new constructed a-posteriori error estimator for domains with bent boundaries. For given $f \in C(\Omega)$, the weak formulation of the Poisson problem

$$\begin{aligned} -\Delta u &= f, \quad \text{in } \Omega, \\ u &= 0, \quad \text{on } \partial\Omega, \end{aligned} \tag{1}$$

is given by

$$\begin{aligned} u \in V \qquad & (\nabla u, \nabla \phi) = (f, \phi) \qquad \forall \phi \in V, \\ u_h \in V_h \subset V \qquad & (\nabla u_h, \nabla \phi_h) = (f, \phi_h) \qquad \forall \phi_h \in V_h. \end{aligned}$$

Here, $(\cdot, \cdot)$ is the L^2-scalar product on Ω or on the discrete ansatz space Ω_h. The Lax-Milgram theorem guarantees the existence of u and u_h.
The solution u_h resulting from the Finite Element method will be an approximation of the exact solution u. The error $(u - u_h)$ measures the quality of the discrete solution. The natural error norm for a elliptic partial differential equation is the H^1-norm. For the sake of simplicity we will focus on the case of convex boundaries. However, it should be possible to construct an a-posteriori error estimator for domains with concavely bent boundaries as well by using similar methods.

2.2 A-posteriori error estimation

Being defined locally, an a-posteriori error estimator provides information about where the biggest contributions to the error are to be expected. As we are talking about an a-posteriori estimator it just should depend on the computable quantities f, u_h and Ω_h. The standard residual based error estimator works well for domains with polygonal boundaries.

The following theory assumes that we discretize with a regular triangulation, where is $\Delta u_h|_T = 0$.

2.2.1 The standard a-posteriori error estimator

Under the assumption that $u \in V$ fulfills the variational formulation of our problem and that $u_h \in V_h$ is the linear Finite Element approximation we know that the error $e_h := u - u_h$ can be estimated by

$$\|\nabla e_h\|_{L^2(\Omega)} \leq c\,\eta_h, \quad \eta_h := \left(\sum_{T \in \Omega_h} (\rho_T^2 + \sum_{E \in \partial T} \rho_E^2) \right)^{\frac{1}{2}}. \tag{2}$$

Here, the cell residuals ρ_T and the edge residuals ρ_E are defined by

$$\rho_T := h_T \, \|f\|_{L^2(T)},$$
$$\rho_E := \frac{1}{2} \, h_E^{\frac{1}{2}} \, \|[n_E \cdot \nabla u_h]\|_{L^2(E)}.$$

The notation $[,]$ refers to the jump over an inner edge between two neighbor cells T_1 and T_2, defined by

$$[n_E \cdot \nabla u_h] := \begin{cases} n_{T_1} \cdot \nabla u_h|_{T_1} + n_{T_2} \cdot \nabla u_h|_{T_2}, & E \subset \overline{T}_1 \cap \overline{T}_2,\, T_1 \neq T_2 \\ 0, & E \subset \partial\Omega. \end{cases} \tag{3}$$

We have to pay attention that the theory assumes that we discretize with an triangulation, in the numerical implementiation with Hiflow^3 we use a quadrangular grid. Then the residual term ρ_T looks like:

$$\rho_T := h_T \, \|f + \Delta u_h\|_{L^2(T)},$$

because $\|\Delta u_h\| \neq 0$ as automatically in the case of a triangulation.

2.2.2 The extended a-posteriori error estimator

The residual based energy error estimator for domains with convex bent boundaries is composed of the standard energy error estimator above and additive terms which deal with the bent boundaries. In the convex case the error estimate again takes the following form:

$$\|\nabla e_h\|_{L^2(\Omega)} \leq c\,\eta_h. \tag{4}$$

However, η_h is now given by

$$\eta_h := \left(\sum_{T \in \Omega_h} \left(\rho_T^2 + \sum_{E \in \partial T} \rho_E^2 + \sum_{E \in \partial T} \rho_A^2 + \sum_{S = S_T} \rho_S^2\right) \right)^{\frac{1}{2}},$$

with the cell residuals ρ_T, the edge residuals ρ_E, the boundary edge residuals ρ_A and the boundary cell residuals ρ_S, defined as

$$\rho_T := h_T \, \|f\|_{L^2(T)} \text{ respectively } \rho_T := h_T \, \|f + \Delta u_h\|_{L^2(T)},$$
$$\rho_E := \frac{1}{2} \, h_E^{\frac{1}{2}} \, \|[n_E \cdot \nabla u_h]\|_{L^2(E)},$$
$$\rho_A := h_{T_A} \, \|n_A \cdot \nabla u_h\|_{L^2(A)},$$
$$\rho_S := h_A^2 \, \|f\|_{L^2(S)}.$$

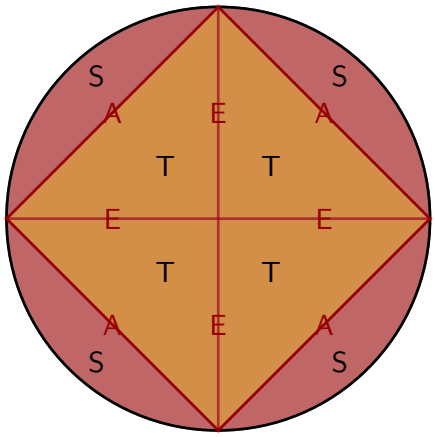

Figure 2: A simple Finite Element grid on the unit square

In (2), T is an element of the triangulation, E denotes the inner edge between two cells, A denotes a boundary edge of the discrete space Ω_h and S a boundary cell between A and $\partial\Omega$ as in Figure 2. That means in particular that the residuals ρ_S and ρ_A disappear in the sum over all cells as soon as T is an inner cell, which is not bordering on a boundary cell S. S_T is a boundary cell above an outer cell T. ρ_E is defined over the jump over an inner edge as in Equation 3
Clearly, the parts ρ_T describe the classical residual of the Poisson equation $-\Delta u = f$. The jumps over the edges in ρ_E measure the smoothness of the discrete solution. As a consequence, the edge residuals of u disappear if u is a C^1-function, see [5, page 91]. For the evaluation of the energy error estimator the data f have to be known and we have to compute the jumps on the respective edges. Furthermore, we need to compute the outer normal vectors along the boundary edges A. ρ_S must be computed approximatively, as the domain S_h is not known in general. Especially the constant c as a prefactor of the estimator consists of two constants c_I and c_T. The interpolation constant c_I is usually estimated, too. The constant c_T which measures the degree of overlapping can be computed; see [5, page 91].
For the implementation, we cut down on a linear Finite Element ansatz and just solve the Poisson equation on the two dimensional unit circle as the continuous domain.

For the proof of the extended error estimator and more details we refer to the source [4, page 18].

2.3 Implementation of the extended a-posteriori error estimator

To implement the extended error estimator $\|\nabla e_h\|_{L^2(\Omega)} \leq c\,\eta_h$ described in Section 2.2.2 we need to compute four residual terms. The cell residuals depend on the discretization parameter h_T and the L^2 norm over Ω_h of the data f or of $f + \Delta u_h$, if we deal with a quadrangular discretization. For the edge residuals we need to know the length of the edge h_E and the jumps over the current edge.
The boundary residuals are defined by the length of the outer edge and the L^2- norm on boundary cells S of f. The domain of integration S is usually unknown. However, in case of the unit circle we can approximate S appropriately. To get ρ_A we require the length of the outer edge and the evaluation of ∇u_h in the direction of the outer unit normal along the outer edge A.
Concerning the application of the extended error estimator introduced above for adaptive refinement we propose the following: in each step of the refinement we refine on those elements whose

local error η_T lies above the mean error $\frac{\sum_{T\in\Omega_h}\eta_T}{\#T}$. Furthermore, we choose an upper bound of 100.000 cells in order to reduce the run time.
To stress the relevance of the additive terms of the new error estimator we will compare in Section 5.1 the extended and the standard error estimator.

3 The Commented Program

3.1 Preliminaries

HiFlow^3 is designed for high performance computing on massively parallel machines. So it applies the Message Passing Interface (MPI) library specification for message-passing, see sections 3.4, 3.5.2, 3.5.6, 3.5.10, [2], [1] .
The Poisson tutorial needs following two input files:

- A parameter file: The parameter file is an xml-file, which contains all parameters needed to execute the program. It is read in by the program. It is not necessary to recompile the program, when parameters in the xml-file are changed. By default the Poisson adaptive tutorial reads in the parameter file poisson_adaptive.xml, see section 3.2, which contains the parameters of a first numerical example, see section 5. This file is stored in `/hiflow/examples/poisson_adaptive/`.

- Geometry data: The file containing the geometry is specified in the parameter file (poisson_adaptive.xml).
  In the numerical example in Section 5 we used **unit_square_inner_square.inp**. You can find different meshes in the folder `/hiflow/examples/data` .

HiFlow^3 does not generate meshes for the domain Ω. Meshes in *.inp and *.vtu format can be read in. It is possible to extend the reader for other formats. Furthermore it is possible to generate other geometries by using external programs (Mesh generators) or by hand.

3.2 Parameter File

The parameters required are initialized in the paramter file poisson_adaptive.xml.

```
,
<Param>
  <Mesh>
    <Filename2>unit_square_inner_square.inp</Filename2>
    <InitialRefLevel>1</InitialRefLevel>
    <FinalRefLevel>100</FinalRefLevel>
    <FeDegree>1</FeDegree>
    <Refinement>2</Refinement>
    <Estimator>1</Estimator>
  </Mesh>
  <LinearAlgebra>
    <NameMatrix>CoupledMatrix</NameMatrix>
    <NameVector>CoupledVector</NameVector>
    <Platform>CPU</Platform>
    <Implementation>Naive</Implementation>
    <MatrixFormat>CSR</MatrixFormat>
  </LinearAlgebra>
  <LinearSolver>
```

```
    <Name>CG</Name>
    <SizeBasis>50</SizeBasis>
    <Method>Preconditioning</Method>
    <MaxIterations>10000</MaxIterations>
    <AbsTolerance>1.0e-14</AbsTolerance>
    <RelTolerance>1.0e-8</RelTolerance>
    <DivTolerance>1.0e6</DivTolerance>
  </LinearSolver>
</Param>
```

3.3 Structure of the Tutorial

Following member functions (signed by •) are defined in the class `Poisson Adaptive`. The survey reflects the access relations between the functions, classes and structs.

- run()
- build_initial_mesh()
- prepare_system()
  - DirichletZero struct (poisson_adaptive.h)
- assemble_system()
  - LocalPoissonAssembler class (poisson_adaptive.h)
    - ExactSol-struct (poisson_adaptive.h)
- solve_system()
- compute_error()
  - L2ErrorIntegrator class (poisson_adaptive.h)
  - H1ErrorIntegrator class (poisson_adaptive.h)
- error_estimator()
- visualize()
- adapt_uniform()
- adapt()

You can find the source text of every member function in an extra section below.

3.4 Main Function

The main function starts the simulation of the Poisson problem (poisson_adaptive.cc).

```
,
// Program entry point

int main( int argc, char** argv )
{
    MPI_Init( &argc, &argv );
```

```
    // Set default parameter file
    std::string param_filename( PARAM_FILENAME );
    std::string path_mesh;
    // If set take parameter file specified on console
    if ( argc > 1 )
    {
        param_filename = std::string( argv[1] );
    }
    // If set take mesh following path specified on console
    if ( argc > 2 )
    {
        path_mesh = std::string( argv[2] );
    }
    try
    {
        // Create log files for INFO and DEBUG output
        std::ofstream info_log( "poisson_adaptive_info_log" );
        LogKeeper::get_log( "info" ).set_target( &info_log );
        std::ofstream debug_log( "poisson_adaptive_debug_log" );
        LogKeeper::get_log( "debug" ).set_target( &debug_log );

        // Create application object and run it
        PoissonAdaptive app( param_filename, path_mesh );
        app.run( );

    }
    catch ( std::exception& e )
    {
        std::cerr << "\nProgram␣ended␣with␣uncaught␣exception.\n";
        std::cerr << e.what( ) << "\n";
        return -1;
    }
    MPI_Finalize( );
    return 0;
}
```

3.5 Member Functions

3.5.1 run()

The member function run() is defined in the class `Poisson Adaptive` (poisson_adaptive.cc).

```
'
    // Main algorithm

    void run( )
    {
        // Construct / read in the initial mesh.
        build_initial_mesh( );
        // Main adaptation loop.
        while ( !is_done_ )
        {
            LOG_DEBUG( 1, "=====␣\nRefinement␣level:␣"
                       << refinement_level_
                       << "\n=====" );

            std::cout << "=====␣\nRefinement␣level:␣"
                      << refinement_level_
                      << "\n====="
```

```
                << std::endl;
        // Initialize space and linear algebra.
        prepare_system( );
        // Compute the stiffness matrix and right-hand side.
        assemble_system( );
        // Solve the linear system.
        solve_system( );
        // Compute the error to the exact solution.
        // Includes the A posteriori error estimator.
        compute_error( );
        error_estimator( );

        // Visualize the solution and the errors.
        visualize( );
        // Uniform or adaptive refinement
        int refinement_ = params_["Mesh"]["Refinement"].get<int>( 3 );
        if ( refinement_ == 1 )
        {
            adapt_uniform( );
        }
        else
        {
            adapt( );
        }
    }
}
```

3.5.2 build_initial_mesh()

This member function, defined in the class Poisson Adaptive, reads the initial mesh (poisson_adaptive.cc), and writes out the refined mesh of initial refinement level.

```
,
void PoissonAdaptive::build_initial_mesh( )
{
    // Read in the mesh.
    //The mesh is chosen according to the dimension of the problem.
    std::string mesh_name;

    mesh_name =
            params_["Mesh"]["Filename2"].get<std::string>( "unit_square.inp" );

    std::string mesh_filename;
    if ( path_mesh.empty( ) )
    {
        mesh_filename = std::string( DATADIR ) + mesh_name;
    }
    else
    {
        mesh_filename = path_mesh + mesh_name;
    }

    // circle with radius = 1
    Coordinate radius = 1.;
    Ellipsoid circle( radius, radius );
    master_mesh_ = read_mesh_from_file( mesh_filename,
                                        DIMENSION,
                                        DIMENSION,
```

```
                                        0 );

    adapt_boundary_to_function( master_mesh_, circle );

    // Refine the mesh until the initial refinement level is reached.
    const int initial_ref_lvl =
            params_["Mesh"]["InitialRefLevel"].get<int>( 3 );

    for ( int r = 0; r < initial_ref_lvl; ++r )
    {
        adapt_boundary_to_function( master_mesh_, circle );
        ++refinement_level_;
    }

    // Add subdomain and remote index information needed by the library
    std::vector<int> remote_index
            (
              master_mesh_->num_entities( master_mesh_->tdim( ) ),
              -1
              );

    AttributePtr remote_index_attr( new IntAttribute( remote_index ) );
    master_mesh_->add_attribute( "_remote_index_",
                                 DIMENSION,
                                 remote_index_attr );

    std::vector<int> subdomain
            ( master_mesh_->num_entities( master_mesh_->tdim( ) ),
              0
              );
    AttributePtr subdomain_attr( new IntAttribute( subdomain ) );
    master_mesh_->add_attribute( "_sub_domain_",
                                 DIMENSION,
                                 subdomain_attr );
}
```

3.5.3 prepare_system()

The member function prepare_system() initializes the space and the linear algebra (poisson_adaptive.cc). The polynomial degree of the Finite Element functions is set to the parameter "FeDegree" defined in the parameter file. The size and the non-zero pattern of the stiffness matrix matrix_ is initialized. The vectors for the right-hand side rhs_, and the solution are initialized and set to a zero vector of the correct size.

```
,
void PoissonAdaptive::prepare_system( )
{
    QuadratureSelection q_sel( 3 );
    global_asm_.set_quadrature_selection_function( q_sel );

    // Assign degrees to each element.
    const int fe_degree = params_["Mesh"]["FeDegree"].get<int>( 1 );
    std::vector< int > degrees( 1, fe_degree );

    // Initialize the VectorSpace object.
    space_.Clear( );
    space_.Init( degrees, *master_mesh_ );
```

```
// Setup couplings object.
couplings_.Clear( );
couplings_.Init( comm_, space_.dof( ) );

// Compute the matrix graph.
SparsityStructure sparsity;
global_asm_.compute_sparsity_structure( space_, sparsity );

couplings_.InitializeCouplings( sparsity.off_diagonal_rows,
                                sparsity.off_diagonal_cols );

// Setup linear algebra objects.
delete matrix_;
delete rhs_;
delete sol_;

CoupledMatrixFactory<Scalar> CoupMaFact;
matrix_ = CoupMaFact.Get
        (
          params_["LinearAlgebra"]["NameMatrix"].get<std::string>
          (
            "CoupledMatrix"
          )
          )->
        params
        ( params_["LinearAlgebra"]
          );
matrix_->Init( comm_, couplings_ );

CoupledVectorFactory<Scalar> CoupVecFact;
rhs_ = CoupVecFact.Get
        (
          params_["LinearAlgebra"]["NameVector"].get<std::string>
          (
            "CoupledVector"
          )
          )->
        params
        (
          params_["LinearAlgebra"]
          );
sol_ = CoupVecFact.Get
        (
          params_["LinearAlgebra"]["NameVector"].get<std::string>
          (
            "CoupledVector"
          )
          )->
        params
        (
          params_["LinearAlgebra"]
          );
rhs_->Init( comm_, couplings_ );
sol_->Init( comm_, couplings_ );

// Initialize structure of LA objects.
matrix_->InitStructure( vec2ptr( sparsity.diagonal_rows ),
                        vec2ptr( sparsity.diagonal_cols ),
                        sparsity.diagonal_rows.size( ),
```

```
                        vec2ptr( sparsity.off_diagonal_rows ),
                        vec2ptr( sparsity.off_diagonal_cols ),
                        sparsity.off_diagonal_rows.size( ) );

rhs_->InitStructure( );
sol_->InitStructure( );

// Zero all linear algebra objects.
matrix_->Zeros( );
rhs_->Zeros( );
sol_->Zeros( );

// Compute Dirichlet BC dofs and values using known exact solution.
dirichlet_dofs_.clear( );
dirichlet_values_.clear( );

DirichletZero zero;

// The function compute_dirichlet_dofs_and_values des not yet work for 1D.
if ( DIMENSION == 1 )
{
    dirichlet_values_.resize( 2, 0.0 );

    // Loop over all cells.
    for ( EntityIterator facet_it = master_mesh_->begin( DIMENSION - 1 ),
          facet_end = master_mesh_->end( DIMENSION - 1 );
          facet_it != facet_end;
          ++facet_it )
    {

        // Returns the number of neighbors for each cell to check
        // if it is on the facet.
        const EntityCount num_cell_neighbors = facet_it
                ->num_incident_entities
                ( DIMENSION );

        // If it lies on the facet, the corresponding DOF is a Dirichlet
        // DOF and is added to dirichlet_dofs_.
        if ( num_cell_neighbors == 1 )
        {
            std::vector<int> dof_number_;
            space_.dof( ).get_dofs_on_subentity
                    (
                      0,
                      facet_it->begin_incident( DIMENSION )->index( ),
                      0,
                      facet_it->index( ),
                      dof_number_
                      );
            dirichlet_dofs_.push_back( dof_number_[0] );
        }
    }
}
    //homogeneous Dirichlet boundaries
else
{
    compute_dirichlet_dofs_and_values( zero, space_, 0, dirichlet_dofs_,
                                       dirichlet_values_ );
}
```

```
}
```

struct DirichletZero

This struct in poisson_adaptive.h defines the homogeneous Dirichlet boundary condition, given in (1).

```
,
// Dirichlet boundary condition
  // Functor used to impose u = 0 on the boundary.
  struct DirichletZero {
    std::vector<double> evaluate(const mesh::Entity& face,
                                 const std::vector<Coord>& coords_on_face) const {
                                   // return array with Dirichlet values for dof:s on boundar
                                   if(face.get_material_number() == 11) {
                    return std::vector<double>(coords_on_face.size(), 0.0);
                                   }
                                   // Neumann boundary values
                                   return std::vector<double>(0, 0.0);
                                 }
  };
```

3.5.4 assemble_system()

The member function assemble_system() computes the stiffness matrix and right-hand side (poisson_adaptive.cc). The stiffness matrix, right-hand side vector and solution vector are modified to set correct Dirichlet values for the boundary DoFs.

```
,
void PoissonAdaptive::assemble_system( )
{
    // Assemble matrix and right-hand-side vector.
    LocalPoissonAssembler<ExactSol> local_asm;
    global_asm_.assemble_matrix( space_, local_asm, *matrix_ );
    global_asm_.assemble_vector( space_, local_asm, *rhs_ );

    if ( !dirichlet_dofs_.empty( ) )
    {
        // Correct Dirichlet dofs.
        matrix_->diagonalize_rows( vec2ptr( dirichlet_dofs_ ),
                                   dirichlet_dofs_.size( ), 1.0 );
        rhs_->SetValues( vec2ptr( dirichlet_dofs_ ), dirichlet_dofs_.size( ),
                         vec2ptr( dirichlet_values_ ) );
        sol_->SetValues( vec2ptr( dirichlet_dofs_ ), dirichlet_dofs_.size( ),
                         vec2ptr( dirichlet_values_ ) );
    }
}
```

LocalPoissonAssembler class

This class implements the stiffness matrix and right-hand side locally for each cell.

```
,
// Assembling of the linear system
// Functor used for the local assembly of the stiffness matrix and load vector.

template<class ExactSol>
class LocalPoissonAssembler : private AssemblyAssistant<DIMENSION, double>
{
public:
```

```
    void operator () ( const Element<double>& element,
                       const Quadrature<double>& quadrature,
                       LocalMatrix& lm )
    {
        AssemblyAssistant<DIMENSION, double>::initialize_for_element
                ( element, quadrature );

        // Local stiffness matrix
        const int num_q = num_quadrature_points ( );
        for ( int q = 0; q < num_q; ++q )
        {
            const double wq = w ( q );
            const int n_dofs = num_dofs ( 0 );
            for ( int i = 0; i < n_dofs; ++i )
            {
                for ( int j = 0; j < n_dofs; ++j )
                {
                    lm ( dof_index ( i, 0 ), dof_index ( j, 0 ) ) +=
                            wq
                            * dot ( grad_phi ( j, q ), grad_phi ( i, q ) )
                            * std::abs ( detJ ( q ) );
                }
            }
        }
    }

    void operator () ( const Element<double>& element,
                       const Quadrature<double>& quadrature,
                       LocalVector& lv )
    {
        AssemblyAssistant<DIMENSION, double>::initialize_for_element
                ( element, quadrature );

        const int num_q = num_quadrature_points ( );
        for ( int q = 0; q < num_q; ++q )
        {
            const double wq = w ( q );
            const int n_dofs = num_dofs ( 0 );
            for ( int i = 0; i < n_dofs; ++i )
            {
                lv[dof_index ( i, 0 )] += wq
                        * f ( x ( q ) ) * phi ( i, q )
                        * std::abs ( detJ ( q ) );
            }
        }
    }

};
 // Right hand side f

double f ( Vec<DIMENSION, double> pt )
{
    ExactSol sol;
    double rhs_sol;
    const double x = pt[0];
    const double y = ( DIMENSION > 1 ) ? pt[1] : 0;
    const double pi = M_PI;
    const double a = 1.;
    const double b = 1.;
```

```
    const double k = 10.0;
    rhs_sol = (
                8.
                * k
                * std::pow (
                            ( x * x ) / ( a * a )+( y * y ) / ( b * b ),
                            4. * k
                            )
                * (
                    a * a * a * a * ( 8. * k - 1 ) * y * y
                    + a * a * b * b * ( x * x + y * y )
                    + b * b * b * b * ( 8. * k - 1 ) * x * x
                    )
                )
            / (
                std::pow ( a * a * y * y + b * b * x*x, 2 )
                );

    return rhs_sol;
}
```

ExactSol struct

The struct ExactSol in poisson_adaptive.h implements the exact solution u given by (6) and the exact gradient ∇u.

```
,
// Exact solution

struct ExactSol
{

    double operator () ( const Vec<DIMENSION, double>& pt ) const
    {
        const double x = pt[0];
        const double y = ( DIMENSION > 1 ) ? pt[1] : 0;
        double solution;
        // a and b are the axes of the ellipse geometry
        // unit circle a = b = 1
        const double a = 1.;
        const double b = 1.;
        const double k = 10.0;
        solution = -std::pow ( ( ( x / a )*( x / a )+( y / b )*( y / b ) ),
                               4. * k )
                + 1.;

        return solution;
    }

    // Partial derivations of the exact solution

    Vec<DIMENSION, double> eval_grad ( const Vec<DIMENSION, double>& pt ) const
    {
        Vec<DIMENSION, double> grad;
        const double x = pt[0];
        const double y = ( DIMENSION > 1 ) ? pt[1] : 0;

        const double a = 1.;
        const double b = 1.;
        const double k = 10.0;
        grad[0] = -4. * k * std::pow ( ( x / a )*( x / a )+( y / b )*( y / b ),
```

```
                                        ( 4. * k - 1. ) )
                * ( 2. / ( a * a ) )
                * x;

        grad[1] = -4. * k * std::pow ( ( x / a )*( x / a )+( y / b )*( y / b ),
                                        ( 4. * k - 1. ) )
                * ( 2. / ( b * b ) )
                * y;
        return grad;
    }
};
```

3.5.5 solve_system()

The member function solve_system() solves the linear system (poisson_adaptive.cc). The solver is specified in the parameter file. The Poisson equation is symmetric positive definite, which means that the CG-method is a good choice, see 3.2.

```
,
void PoissonAdaptive::solve_system( )
{
    LinearSolver<LAD>* solver_;
    LinearSolverFactory<LAD> SolFact;
    solver_ = SolFact.Get
            (
              params_["LinearSolver"]["Name"].get<std::string>( "CG" )
              )->
            params
            (
              params_["LinearSolver"]
              );

#ifdef WITH_ILUPP
    PreconditionerIlupp<LAD> precond;
    precond.InitParameter( 0,
                           1010,
                           75,
                           0.6,
                           3.5,
                           0.005 );
#else
    PreconditionerBlockJacobiStand<LAD> precond;
    precond.InitParameter( );
    precond.Init_SSOR( 1.3 );
#endif

    precond.SetupOperator( *matrix_ );
    solver_->SetupPreconditioner( precond );
    solver_->SetupOperator( *matrix_ );
    solver_->Solve( *rhs_, sol_ );
    interpolate_constrained_vector( space_, *sol_ );
    delete solver_;
}
```

3.5.6 compute_error()

This member function in poisson_adaptive.cc computes the error between the approximated and the exact solution mentioned in Section 5 in the L^2 norm and H^1 seminorm.

```
,
void PoissonAdaptive::compute_error( )
{
    // Prepare sol_ for post processing
    sol_->UpdateCouplings( );

    // Compute square of the L2 error on each element, putting the
    // values into L2_err_.
    L2_err_.clear( );
    L2ErrorIntegrator<ExactSol> L2_int( *( sol_ ) );
    global_asm_.assemble_scalar( space_, L2_int, L2_err_ );

    // Create attribute with L2 error for output.
    AttributePtr L2_err_attr( new DoubleAttribute( L2_err_ ) );
    master_mesh_->add_attribute( "L2 error", DIMENSION, L2_err_attr );
    double total_L2_err = std::accumulate
            (
              L2_err_.begin( ),
              L2_err_.end( ),
              0.
              );
    double global_L2_err = 0.;
    MPI_Reduce( &total_L2_err,
                &global_L2_err,
                1,
                MPI_DOUBLE,
                MPI_SUM,
                MASTER_RANK,
                comm_ );
    LOG_INFO( "error", "Local L2 error on partition " << rank_ << " = "
              << std::sqrt( total_L2_err ) );

    // Compute square of the H1 error on each element, putting the
    // values into H1_err_.
    H1_err_.clear( );
    H1ErrorIntegrator<ExactSol> H1_int( *( sol_ ) );
    global_asm_.assemble_scalar( space_, H1_int, H1_err_ );

    // Create attribute with H1 error for output.
    AttributePtr H1_err_attr( new DoubleAttribute( H1_err_ ) );
    master_mesh_->add_attribute( "H1 error", DIMENSION, H1_err_attr );
    double total_H1_err = std::accumulate
            (
              H1_err_.begin( ),
              H1_err_.end( ),
              0.
              );
    double global_H1_err = 0.;
    MPI_Reduce( &total_H1_err,
                &global_H1_err,
                1,
                MPI_DOUBLE,
                MPI_SUM,
                MASTER_RANK,
```

```
                comm_ );
    LOG_INFO( "error", "Local␣H1␣error␣on␣partition␣" << rank_ << "␣=␣"
              << std::sqrt( total_H1_err ) );

    // Output on consol: number of elements and global H^1 error
    if ( rank_ == MASTER_RANK )
    {
        std::cout << "H1size:␣" << L2_err_.size( ) << "\n"
                << "H1error:␣" << std::sqrt( global_H1_err ) << "\n";
    }
}
```

L2ErrorIntegrator class

The class L2ErrorIntegrator implements the evaluation of the square of the L^2 norm of the error on each element (poisson_adaptive.h).

```
,
// Functor used for the local evaluation of the square of the L2-norm of the
// error on each element.

template<class ExactSol>
class L2ErrorIntegrator : private AssemblyAssistant<DIMENSION, double>
{
public:

    L2ErrorIntegrator ( CoupledVector<Scalar>& pp_sol )
    : pp_sol_ ( pp_sol )
    {
    }

    void operator () ( const Element<double>& element,
                       const Quadrature<double>& quadrature,
                       double& value )
    {
        AssemblyAssistant<DIMENSION, double>::initialize_for_element
                ( element, quadrature );

        // Evaluate the computed solution at all quadrature points.
        evaluate_fe_function ( pp_sol_, 0., approx_sol_ );

        const int num_q = num_quadrature_points ( );
        for ( int q = 0; q < num_q; ++q )
        {
            const double wq = w ( q );
            // Delta is the quadrature point wise error.
            const double delta = sol_ ( x ( q ) ) - approx_sol_[q];
            value += wq * delta * delta * std::abs ( detJ ( q ) );
        }
    }

private:
    // Coefficients of the computed solution
    const CoupledVector<Scalar>& pp_sol_;
    // functor to evaluate exact solution
    ExactSol sol_;
    // Vector with values of computed solution evaluated at
    // each quadrature point
    FunctionValues< double > approx_sol_;
};
```

H1ErrorIntegrator class

The class H1ErrorIntegrator implements the evaluation of the square of the H^1 seminorm of the error on each element (poisson_adaptive.h).

```
,
// Functor used for the local evaluation of the square of the H1-norm of the
// error on each element.

template<class ExactSol>
class H1ErrorIntegrator : private AssemblyAssistant<DIMENSION, double>
{
public:

    H1ErrorIntegrator ( CoupledVector<Scalar>& pp_sol )
    : pp_sol_ ( pp_sol )
    {
    }

    void operator () ( const Element<double>& element,
                       const Quadrature<double>& quadrature,
                       double& value )
    {
        AssemblyAssistant<DIMENSION, double>::initialize_for_element
                ( element, quadrature );
        // Evaluate the gradient of the computed solution at all
        // quadrature points.
        evaluate_fe_function_gradients ( pp_sol_, 0., approx_grad_u_ );

        const int num_q = num_quadrature_points ( );
        for ( int q = 0.; q < num_q; ++q )
        {
            const Vec<DIMENSION, double> grad_u = sol_.eval_grad ( x ( q ) );
            value += w ( q )
                    * (
                        dot ( grad_u, grad_u )
                        - 2. * dot ( grad_u, approx_grad_u_[q] )
                        + dot ( approx_grad_u_[q], approx_grad_u_[q] )
                        )
                    * std::abs ( detJ ( q ) );
        }
    }

private:
    // Coefficients of the computed solution
    const CoupledVector<Scalar>& pp_sol_;
    // Functor to evaluate exact solution
    ExactSol sol_;
    // Gradient of computed solution evaluated at each quadrature point
    FunctionValues< Vec<DIMENSION, double> > approx_grad_u_;
};
```

3.5.7 error_estimator()

The member function error_estimator() in poisson_adaptive.cc contains all computations and outputs of the terms of the error estimator in Section 2.2.2.

```
,
void PoissonAdaptive::error_estimator( )
```

```
{

    InterfaceList if_list = InterfaceList::create( master_mesh_ );
    int Number_Boundary = 0;
    for ( InterfaceList::const_iterator it = if_list.begin( ),
          end_it = if_list.end( );
          it != end_it;
          ++it )
    {

        int remote_index_master = -10;
        master_mesh_->get_attribute_value
                ( "_remote_index_",
                  master_mesh_->tdim( ),
                  it->master_index( ),
                  &remote_index_master
                  );

        const int num_slaves = it->num_slaves( );
        if ( remote_index_master == -1 )
        {
            if ( num_slaves == 0 )
            {
                Number_Boundary += 1;
            }
        }
    }

    // Create attribute with H1 error for output.
    AttributePtr H1_err_attr( new DoubleAttribute( H1_err_ ) );
    master_mesh_->add_attribute( "H1 error", DIMENSION, H1_err_attr );
    double total_H1_err = std::accumulate
            (
              H1_err_.begin( ),
              H1_err_.end( ),
              0.
              );
    double global_H1_err = 0.;
    MPI_Reduce( &total_H1_err,
                &global_H1_err,
                1,
                MPI_DOUBLE,
                MPI_SUM,
                MASTER_RANK,
                comm_ );

    // rho_T output on console
    rho_cell_.clear( );
    CellTermAssembler rho_cell_int( *sol_ );
    global_asm_.assemble_scalar( space_, rho_cell_int, rho_cell_ );
    // Create attribute with term on the cells for output.
    AttributePtr rho_cell_attr( new DoubleAttribute( rho_cell_ ) );
    master_mesh_->add_attribute( "rho_T", DIMENSION, rho_cell_attr );
    double total_rho_cell = std::accumulate
            (
              rho_cell_.begin( ),
              rho_cell_.end( ),
              0.
              );
```

```
double global_rho_cell = 0.;
MPI_Reduce( &total_rho_cell,
            &global_rho_cell,
            1,
            MPI_DOUBLE,
            MPI_SUM,
            MASTER_RANK,
            comm_
            );
LOG_INFO( "error", "Global␣rho_T␣on␣partition␣" << rank_ << "␣=␣"
          << std::sqrt( global_rho_cell ) );

// Output on consol: number of elements and global rho_T
if ( rank_ == MASTER_RANK )
{
    std::cout << "size␣rho_T:␣" << rho_cell_.size( ) << "\n"
            << "value␣rho_T:␣" << std::sqrt( total_rho_cell ) << "\n";
}

// rho_S output on console
rho_bcell_.clear( );
BoundCellTermAssembler rho_bcell_int( *sol_ );
jump_term_asm_.assemble_interface_scalar_cells
        (
          space_,
          rho_bcell_int,
          rho_bcell_
          );
// Create attribute with boundary cell term for output.
AttributePtr rho_bcell_attr( new DoubleAttribute( rho_bcell_ ) );
master_mesh_->add_attribute( "rho_S", DIMENSION, rho_bcell_attr );
double total_rho_bcell = std::accumulate
        (
          rho_bcell_.begin( ),
          rho_bcell_.end( ),
          0. );
double global_rho_bcell = 0.;
MPI_Reduce( &total_rho_bcell,
            &global_rho_bcell,
            1,
            MPI_DOUBLE,
            MPI_SUM,
            MASTER_RANK,
            comm_
            );
LOG_INFO( "error", "Global␣rho_S␣on␣partition␣" << rank_ << "␣=␣"
          << std::sqrt( global_rho_bcell ) );
// Output on consol: number of elements and global rho_S
if ( rank_ == MASTER_RANK )
{
    std::cout << "size␣rho_S:␣" << Number_Boundary << "\n"
            << "value␣rho_S:␣" << std::sqrt( total_rho_bcell ) << "\n";
}

// rho_E integration and output
rho_jump_.clear( );
JumpTermAssembler rho_jump_int( *sol_ );
jump_term_asm_.assemble_interface_scalar_cells
        (
```

```
          space_,
          rho_jump_int,
          rho_jump_
          );
// Create attribute with inner edge jump term for output.
AttributePtr rho_jump_attr( new DoubleAttribute( rho_jump_ ) );
master_mesh_->add_attribute( "rho_E", DIMENSION, rho_jump_attr );
double total_rho_jump = std::accumulate
        (
          rho_jump_.begin( ),
          rho_jump_.end( ),
          0.
          );
double global_rho_jump = 0.;
MPI_Reduce( &total_rho_jump,
            &global_rho_jump,
            1,
            MPI_DOUBLE,
            MPI_SUM,
            MASTER_RANK,
            comm_
            );
LOG_INFO( "error", "Global␣rho_E␣on␣partition␣" << rank_ << "␣=␣"
          << std::sqrt( total_rho_jump ) );
// Output on consol: number of elements and global rho_E
if ( rank_ == MASTER_RANK )
{
    std::cout << "size␣rho_E:␣" << if_list.size( ) - Number_Boundary << "\n"
            << "value␣rho_E:␣" << std::sqrt( global_rho_jump ) << "\n";
}

// rho_A integration and output
rho_boundary_.clear( );
BoundaryTermAssembler rho_boundary_int( *sol_ );
jump_term_asm_.assemble_interface_scalar_cells
        (
          space_,
          rho_boundary_int,
          rho_boundary_
          );
// Create attribute with inner edge jump term for output.
AttributePtr rho_boundary_attr( new DoubleAttribute( rho_boundary_ ) );
master_mesh_->add_attribute( "rho_A", DIMENSION, rho_boundary_attr );
double total_rho_boundary = std::accumulate
        (
          rho_boundary_.begin( ),
          rho_boundary_.end( ),
          0.
          );
double global_rho_boundary = 0.;
MPI_Reduce( &total_rho_boundary,
            &global_rho_boundary,
            1,
            MPI_DOUBLE,
            MPI_SUM,
            MASTER_RANK,
            comm_
            );
LOG_INFO( "error", "Global␣rho_A␣on␣partition␣" << rank_ << "␣=␣"
```

```
          << std::sqrt( total_rho_boundary ) );
// Output on consol: number of elements and global rho_E
if ( rank_ == MASTER_RANK )
{
    std::cout << "size␣rho_A:␣" << Number_Boundary << "\n"
            << "value␣rho_A:␣" << std::sqrt( global_rho_boundary ) << "\n";
}

// Standard estimator output on console
std_estimator_.clear( );
for ( int i = 0; i < L2_err_.size( ); ++i )
{
    std_estimator_.push_back( rho_cell_[i] + rho_jump_[i] );
}
// Create attribute with term on the cells for output.
AttributePtr std_estimator_attr( new DoubleAttribute( std_estimator_ ) );
master_mesh_->add_attribute
        ( "std_estimator",
          DIMENSION,
          std_estimator_attr );
double total_std_estimator = std::accumulate
        (
          std_estimator_.begin( ),
          std_estimator_.end( ),
          0.
          );
double global_std_estimator = 0.;
MPI_Reduce( &total_std_estimator,
            &global_std_estimator,
            1,
            MPI_DOUBLE,
            MPI_SUM,
            MASTER_RANK,
            comm_
            );
LOG_INFO( "error", "Global␣std_estimator␣on␣partition␣" << rank_ << "␣=␣"
          << std::sqrt( global_std_estimator ) );

// Output on console: number of elements and global rho_std
if ( rank_ == MASTER_RANK )
{
    std::cout << "size␣stdEst:␣" << std_estimator_.size( ) << "\n"
            << "value␣stdEst:␣" << std::sqrt( total_std_estimator ) << "\n";
}

// New error estimator output on console
new_estimator_.clear( );
for ( int i = 0; i < L2_err_.size( ); ++i )
{
    new_estimator_.push_back
            (
              rho_cell_[i]
              + rho_jump_[i]
              + rho_bcell_[i]
              + rho_boundary_[i]
              );
}
// Create attribute with term on the cells for output.
AttributePtr new_estimator_attr( new DoubleAttribute( new_estimator_ ) );
```

```
    master_mesh_->add_attribute
            ( "new_estimator",
              DIMENSION,
              new_estimator_attr );
    double total_new_estimator = std::accumulate
            (
              new_estimator_.begin( ),
              new_estimator_.end( ),
              0.
              );
    double global_new_estimator = 0.;
    MPI_Reduce( &total_new_estimator,
                &global_new_estimator,
                1,
                MPI_DOUBLE,
                MPI_SUM,
                MASTER_RANK,
                comm_ );
    LOG_INFO( "error", "Global new_estimator on partition " << rank_ << " = "
              << std::sqrt( total_new_estimator ) );

    // Output on console: number of elements and global rho_new
    if ( rank_ == MASTER_RANK )
    {
        std::cout << "size newEst: " << new_estimator_.size( )
                  << "\n"
                  << "value newEst: " << std::sqrt( global_new_estimator )
                  << "\n";
    }
    // Division new error estimator or std estimator / real H^1 error

    int estimator_ = params_["Mesh"]["Estimator"].get<int>( 3 );
    if ( estimator_ == 1 )
    {
        double relation = std::abs( sqrt( global_std_estimator )
                                    / sqrt( global_H1_err ) );
        if ( rank_ == MASTER_RANK )
        {
            std::cout << "relation: " << relation << "\n";
        }
    }
    else
    {
        double relation = std::abs( sqrt( global_new_estimator )
                                    / sqrt( global_H1_err ) );
        if ( rank_ == MASTER_RANK )
        {
            std::cout << "relation: " << relation << "\n";
        }
    }
}
```

CellTermAssembler class

The class CellTermAssembler implements the evaluation of the square of the H^1 seminorm of the cell term $\rho_T := h_T \, \|f + \Delta u_h\|_{L^2(T)}$ on each inner cell T (poisson_adaptive.h).

```
,
// Cell Term Assembler for inner cells

class CellTermAssembler : private AssemblyAssistant <DIMENSION, double>
```

```
{
public:

    CellTermAssembler ( CoupledVector<Scalar>& u_h )
    : u_h_ ( u_h )
    {
    }

    void operator () ( const Element<double>& element,
                       const Quadrature<double>& quadrature,
                       double& value )
    {

        AssemblyAssistant<DIMENSION, double>::initialize_for_element
                ( element, quadrature );

        evaluate_fe_function_hessians ( u_h_, 0, approx_hess_u_h );

        const int num_q = num_quadrature_points ( );

        double h_T = 0.;
        for ( int i = 0; i < num_q; ++i )
        {
            for ( int j = 0; j < num_q; ++j )
            {
                h_T = std::max ( h_T, norm ( x ( i ) - x ( j ) ) );
            }
        }
        for ( int q = 0.; q < num_q; ++q )
        {
            const double laplace_u_h = trace ( approx_hess_u_h[q] );
            value += h_T * h_T
                    * w ( q )
                    * ( f ( x ( q ) ) + laplace_u_h )
                    * ( f ( x ( q ) ) + laplace_u_h )
                    * std::abs ( detJ ( q ) );
        }
    }
    const CoupledVector<Scalar>& u_h_;
    FunctionValues< Mat<DIMENSION, DIMENSION, double> > approx_hess_u_h;
};
```

BoundCellTermAssembler class

The class BoundCellTermAssembler implements the evaluation of the square of the H^1 seminorm of the boundary cell term $\rho_S := h_A^2 \, \|f\|_{L^2(S)}$ on each boundary cell S (poisson_adaptive.h).

```
,
// Cell Term Assembler for the semicircular boundary cells

class BoundCellTermAssembler : private DGAssemblyAssistant <DIMENSION, double>
{
public:

    BoundCellTermAssembler ( CoupledVector<Scalar>& u_h )
    : u_h_ ( u_h )
    {
    }

    void operator () ( const Element<double>& left_elem,
                       const Element<double>& right_elem,
```

```
                        const Quadrature<double>& left_quad,
                        const Quadrature<double>& right_quad,
                        int left_facet_number,
                        int right_facet_number,
                        InterfaceSide left_if_side,
                        InterfaceSide right_if_side,
                        double& local_val )
    {

        const bool is_boundary =
                (
                  left_if_side == DGGlobalAssembler<double>::INTERFACE_BOUNDARY
                  || right_if_side == DGGlobalAssembler<double>::INTERFACE_BOUNDARY
                  );

        if ( !is_boundary )
        {
            local_val = 0.;
            return;
        }

        initialize_for_interface ( left_elem, right_elem,
                                   left_quad, right_quad,
                                   left_facet_number, right_facet_number,
                                   left_if_side, right_if_side );

        const int num_q = num_quadrature_points ( );
        const int num_dofs_trial = trial ( ).num_dofs ( 0 );

        double h_A = 0.;
        for ( int i = 0; i < num_q; ++i )
        {
            for ( int j = 0; j < num_q; ++j )
            {
                h_A = std::max ( h_A, norm ( x ( i ) - x ( j ) ) );
            }
        }

        const double alpha = std::acos ( ( 1. - ( h_A * h_A ) ) / 2. );
        const double dA = 0.5 * ( alpha - std::sin ( alpha ) );
        const double h_S = h_A / 2. * std::tan ( 0.25 * alpha );
        Vec<DIMENSION, double> q_first = x ( 0 );
        Vec<DIMENSION, double> q_last = x ( num_q - 1 );
        Vec<DIMENSION, double> lot_point;
        lot_point[0] = 0.5 * ( q_first[0] + q_last[0] );
        lot_point[1] = 0.5 * ( q_first[1] + q_last[1] );
        Vec<DIMENSION, double> middle_point;
        middle_point = lot_point + 0.5 * h_S * trial ( ).n ( num_q / 2 );

        local_val += h_A * h_A * h_A * h_A
                * std::abs ( f ( middle_point ) * f ( middle_point ) )
                * dA * dA;
    }
    const CoupledVector<Scalar>& u_h_;
};
```

JumpTermAssembler class

The class JumpTermAssembler implements the evaluation of the square of the H^1 seminorm of the jump term $\rho_E := \frac{1}{2}\, h_E^{\frac{1}{2}}\, \|[n_E \cdot \nabla u_h]\|_{L^2(E)}$ on each inner edge E (poisson_adaptive.h).

```
,
// Jump Term Assembler for the jumps over the inner edges

class JumpTermAssembler : private DGAssemblyAssistant <DIMENSION, double>
{
public:

    JumpTermAssembler ( CoupledVector<Scalar>& u_h )
    : u_h_ ( u_h )
    {
    }

    void operator () ( const Element<double>& left_elem,
                       const Element<double>& right_elem,
                       const Quadrature<double>& left_quad,
                       const Quadrature<double>& right_quad,
                       int left_facet_number,
                       int right_facet_number,
                       InterfaceSide left_if_side,
                       InterfaceSide right_if_side,
                       double& local_val )
    {
        const bool is_boundary =
                (
                  left_if_side == DGGlobalAssembler<double>::INTERFACE_BOUNDARY
                  || right_if_side == DGGlobalAssembler<double>::INTERFACE_BOUNDARY
                  );

        if ( is_boundary
             || left_elem.get_cell_index ( ) == right_elem.get_cell_index ( ) )
        {
            local_val = 0.;
            return;
        }

        initialize_for_interface ( left_elem, right_elem,
                                   left_quad, right_quad,
                                   left_facet_number, right_facet_number,
                                   left_if_side, right_if_side );
        left_grad_u_h.clear ( );
        right_grad_u_h.clear ( );
        trial ( ).evaluate_fe_function_gradients ( u_h_, 0, left_grad_u_h );
        test ( ).evaluate_fe_function_gradients ( u_h_, 0, right_grad_u_h );
        const int num_q = num_quadrature_points ( );
        const int num_dofs_trial = trial ( ).num_dofs ( 0 );
        const int num_dofs_test = test ( ).num_dofs ( 0 );
        double h_E = 0.;
        for ( int i = 0; i < num_q; ++i )
        {
            for ( int j = 0; j < num_q; ++j )
            {
                h_E = std::max ( h_E, norm ( x ( i ) - x ( j ) ) );
            }
        }

        // Loop over quadrature points on each edge
        for ( int q = 0.; q < num_q; ++q )
        {
            const double dS = std::abs ( ds ( q ) );
```

```
            local_val += 0.25
                    * h_E
                    * w ( q )
                    * std::abs (
                                dot ( trial ( ).n ( q ), left_grad_u_h[q] )
                                + dot ( test ( ).n ( q ), right_grad_u_h[q] )
                                )
                    * std::abs (
                                dot ( trial ( ).n ( q ), left_grad_u_h[q] )
                                + dot ( test ( ).n ( q ), right_grad_u_h[q] )
                                )
                    * dS;
        }
    }
    const CoupledVector<Scalar>& u_h_;
    FunctionValues< Vec<DIMENSION, double> > left_grad_u_h;
    FunctionValues< Vec<DIMENSION, double> > right_grad_u_h;
};
```

BoundaryTermAssembler class

The class BoundaryTermAssembler implements the evaluation of the square of the H^1 semi-norm of the jump over the outer edges $\rho_A := h_{T_A} \|n_A \cdot \nabla u_h\|_{L^2(A)}$ on each outer edge (poisson_adaptive.h).

```
,
// Jump Term Assembler for the jumps over the outer edges

class BoundaryTermAssembler : private DGAssemblyAssistant <DIMENSION, double>
{
public:

    BoundaryTermAssembler ( CoupledVector<Scalar>& u_h )
    : u_h_ ( u_h )
    {
    }

    void operator () ( const Element<double>& left_elem,
                       const Element<double>& right_elem,
                       const Quadrature<double>& left_quad,
                       const Quadrature<double>& right_quad,
                       int left_facet_number,
                       int right_facet_number,
                       InterfaceSide left_if_side,
                       InterfaceSide right_if_side,
                       double& local_val )
    {

        const bool is_boundary =
                (
                  left_if_side == DGGlobalAssembler<double>::INTERFACE_BOUNDARY
                  || right_if_side == DGGlobalAssembler<double>::INTERFACE_BOUNDARY
                  );

        // To sort the inner and outer edges
        if ( !is_boundary )
        {
            local_val = 0.;
            return;
        }
```

```
        initialize_for_interface ( left_elem, right_elem,
                                   left_quad, right_quad,
                                   left_facet_number, right_facet_number,
                                   left_if_side, right_if_side );
        left_grad_u_h.clear ( );
        trial ( ).evaluate_fe_function_gradients ( u_h_, 0, left_grad_u_h );
        const int num_q = num_quadrature_points ( );
        const int num_dofs_trial = trial ( ).num_dofs ( 0 );
        const double C = is_boundary ? 1 : 0;

        double h_A = 0.;
        for ( int i = 0; i < num_q; ++i )
        {
            for ( int j = 0; j < num_q; ++j )
            {
                h_A = std::max ( h_A, norm ( x ( i ) - x ( j ) ) );
            }
        }
        // Loop over quadrature points on the edge
        for ( int q = 0.; q < num_q; ++q )
        {
            const double dS = std::abs ( ds ( q ) );
            local_val += C
                    * h_A * h_A
                    * w ( q )
                    * std::abs ( dot ( trial ( ).n ( q ), left_grad_u_h[q] ) )
                    * std::abs ( dot ( trial ( ).n ( q ), left_grad_u_h[q] ) )
                    * dS;
        }
    }
    const CoupledVector<Scalar>& u_h_;
    FunctionValues< Vec<DIMENSION, double> > left_grad_u_h;
};
```

3.5.8 visualize()

The member function visualize() in poisson_adaptive.cc writes out data for visualization. Note that HiFlow^3 has no own visualising module , so far.

```
,
void PoissonAdaptive::visualize( )
{
    sol_->UpdateCouplings( );
    // Setup visualization object.
    const int num_intervals = params_["Mesh"]["FeDegree"].get<int>( 1 );
    ParallelCellVisualization<double> visu
            ( space_,
              num_intervals,
              comm_,
              MASTER_RANK );

    // Generate filename.
    std::stringstream name;
    name << "solution" << refinement_level_;

    std::vector<double> remote_index
            ( master_mesh_->num_entities( master_mesh_->tdim( ) ),
```

```
            0. );
    std::vector<double> sub_domain
            ( master_mesh_->num_entities( master_mesh_->tdim( ) ),
              0. );
    std::vector<double> material_number
            ( master_mesh_->num_entities( master_mesh_->tdim( ) ),
              0. );

    for ( mesh::EntityIterator it = master_mesh_->begin
          ( master_mesh_->tdim( ) );
          it != master_mesh_->end( master_mesh_->tdim( ) );
          ++it )
    {
        int temp1, temp2;
        master_mesh_->get_attribute_value( "_remote_index_",
                                           master_mesh_->tdim( ),
                                           it->index( ),
                                           &temp1 );
        master_mesh_->get_attribute_value( "_sub_domain_",
                                           master_mesh_->tdim( ),
                                           it->index( ),
                                           &temp2 );
        remote_index.at( it->index( ) ) = temp1;
        sub_domain.at( it->index( ) ) = temp2;
        material_number.at( it->index( ) ) = master_mesh_->get_material_number
                ( master_mesh_->tdim( ),
                  it->index( ) );
    }

    visu.visualize( EvalFeFunction<LAD>( space_, *( sol_ ) ), "u" );

    // visualize error measures
    visu.visualize_cell_data( L2_err_, "L2" );
    visu.visualize_cell_data( H1_err_, "H1" );
    visu.visualize_cell_data( rho_cell_, "rho_T" );
    visu.visualize_cell_data( rho_bcell_, "rho_S" );
    visu.visualize_cell_data( rho_jump_, "rho_E" );
    visu.visualize_cell_data( rho_boundary_, "rho_A" );
    visu.visualize_cell_data( std_estimator_, "std_estimator" );
    visu.visualize_cell_data( new_estimator_, "new_estimator" );
    visu.visualize_cell_data( remote_index, "_remote_index_" );
    visu.visualize_cell_data( sub_domain, "_sub_domain_" );
    visu.visualize_cell_data( material_number, "Material␣Id" );
    visu.write( name.str( ) );
}
```

3.5.9 adapt_uniform()

The member function adapt_uniform() modifies the space through uniform refinement (poisson_adaptive.cc).

```
,
void PoissonAdaptive::adapt_uniform( )
{
    // Refine mesh on master process. 1D parallel execution is not
    // yet implemented.
    if ( DIMENSION == 1 )
    {
```

```
        if ( rank_ == MASTER_RANK )
        {
            const int final_ref_level = params_["Mesh"]["FinalRefLevel"]
                    .get<int>( 6 );
            if ( refinement_level_ >= final_ref_level )
            {
                is_done_ = true;
            }
            else
            {
                master_mesh_ = master_mesh_->refine( );
                ++refinement_level_;
            }
        }
    }
    else
    {
        if ( rank_ == MASTER_RANK )
        {
            const int final_ref_level = params_["Mesh"]["FinalRefLevel"]
                    .get<int>( 6 );
            if ( refinement_level_ >= final_ref_level )
            {
                is_done_ = true;
            }
            else
            {
                master_mesh_ = master_mesh_->refine( );

                Coordinate radius = 1.;
                Ellipsoid circle( radius, radius );
                adapt_boundary_to_function( master_mesh_, circle );
                ++refinement_level_;
            }
        }
        // Broadcast information from master to slaves.
        MPI_Bcast( &refinement_level_, 1, MPI_INT, MASTER_RANK, comm_ );
        MPI_Bcast( &is_done_, 1, MPI_CHAR, MASTER_RANK, comm_ );

        if ( !is_done_ )
        {
            // Distribute the new mesh.
            MeshPtr local_mesh = partition_and_distribute
                    ( master_mesh_,
                      MASTER_RANK,
                      comm_ );
            assert( local_mesh != 0 );
            SharedVertexTable shared_verts;
            master_mesh_ = compute_ghost_cells
                    ( *local_mesh,
                      comm_,
                      shared_verts );
        }
    }
    if ( master_mesh_->num_entities( master_mesh_->tdim( ) ) > 100000 )
    {
        is_done_ = true;
    }
}
```

3.5.10 adapt()

The member function adapt() modifies the space through adaptive, local refinement (poisson_adaptive.cc).

```
,
void PoissonAdaptive::adapt( )
{
    // Refine mesh on master process. 1D parallel execution is not
    // yet implemented.
    if ( DIMENSION == 1 )
    {
        const int final_ref_level = params_["Mesh"]["FinalRefLevel"]
                .get<int>( 6 );
        if ( refinement_level_ >= final_ref_level )
        {
            is_done_ = true;
        }
        else
        {
            ++refinement_level_;
        }
    }
    else
    {
        const int final_ref_level = params_["Mesh"]["FinalRefLevel"]
                .get<int>( 6 );
        if ( refinement_level_ >= final_ref_level )
        {
            is_done_ = true;
        }
        else
        {
            is_done_ = true;
            int estimator_ = params_["Mesh"]["Estimator"].get<int>( 3 );
            if ( estimator_ == 1 )
            {
                std::vector<int> refinemnts( std_estimator_.size( ), -5 );
                std::vector<int> sort_err( std_estimator_.size( ), 0 );
                for ( int i = 0; i < std_estimator_.size( ); ++i )
                {
                    sort_err[i] = i;
                }
                sortingPermutation( std_estimator_, sort_err );
                // Refinement of all cells over mean value of the std estimator
                int cell_index = sort_err.size( ) - 1;
                double total_std_estimator = std::accumulate
                        (
                          std_estimator_.begin( ),
                          std_estimator_.end( ),
                          0. );
                while (
                        (
                          std_estimator_[sort_err[cell_index]]
                          >
                          ( total_std_estimator / sort_err.size( ) )
                          )
                        && ( cell_index >= 0 )
                        )
```

```
    {
        refinemnts[sort_err[cell_index]] = 0;
        is_done_ = false;
        cell_index--;
    }
    // Unit square -> Ellipsoid implementation a = b = radius = 1.
    Coordinate radius = 1.;
    Ellipsoid circle( radius, radius );

    master_mesh_ = master_mesh_->refine( refinemnts );
    adapt_boundary_to_function( master_mesh_, circle );
}
else
{
    std::vector<int> refinemnts( new_estimator_.size( ), -5 );
    std::vector<int> sort_err( new_estimator_.size( ), 0 );
    for ( int i = 0; i < new_estimator_.size( ); ++i )
    {
        sort_err[i] = i;
    }
    sortingPermutation( new_estimator_, sort_err );
    // Refinement of all cells over mean value of the new estimator
    int cell_index = sort_err.size( ) - 1;
    double total_new_estimator = std::accumulate
            (
              new_estimator_.begin( ),
              new_estimator_.end( ),
              0. );
    while (
            (
              new_estimator_[sort_err[cell_index]]
              >
              ( total_new_estimator / sort_err.size( ) )
              )
            && ( cell_index >= 0 )
            )
    {
        refinemnts[sort_err[cell_index]] = 0;
        is_done_ = false;
        cell_index--;
    }
    // Unit square -> Ellipsoid implementation a = b = radius = 1.
    Coordinate radius = 1.;
    Ellipsoid circle( radius, radius );

    master_mesh_ = master_mesh_->refine( refinemnts );
    adapt_boundary_to_function( master_mesh_, circle );
}

// Regularize mesh
const TDim td = master_mesh_->tdim( );
bool mesh_is_regular = false;
while ( !mesh_is_regular )
{
    mesh_is_regular = true;

    // Vector to hold the cells that need to be refined
    const EntityCount num_cells = master_mesh_->num_entities( td );
    std::vector<int> refinements;
```

```
refinements.resize( num_cells, -5 );

// Create interface list
InterfaceList if_list = InterfaceList::create( master_mesh_ );

// Loop over interfaces
for ( InterfaceList::const_iterator it = if_list.begin( ),
      end_it = if_list.end( );
      it != end_it; ++it )
{

    // Do hanging nodes exist?
    // Not more than 1 hanging node on each edge allowed.
    const int num_slaves = it->num_slaves( );
    if ( num_slaves > 2 )
    {
        int default_num_children = 1;
        if ( td == 3 )
        {
            Entity const& master_cell = master_mesh_
                    ->get_entity( master_mesh_->tdim( ),
                                  it->master_index( ) );
            if ( master_cell.cell_type( ).tag( )
                 == CellType::HEXAHEDRON )
                default_num_children = 4;
            else if ( master_cell.cell_type( ).tag( )
                      == CellType::TETRAHEDRON )
                default_num_children = 3;
            else
            {
                std::cout
                        << "This␣cell␣type␣is␣not␣treated␣..."
                        << "␣implement␣me."
                        << std::endl;
                interminable_assert( 0 );
            }
        }

        // Check regularity of this interface
        if ( num_slaves > default_num_children )
        {
            mesh_is_regular = false;

            // Add master cell to refinement list
            refinements[it->master_index( )] = 0;
        }
    }
}

// Refine if needed.
// If mesh is not regular -> build new mesh
if ( !mesh_is_regular )
{
    // Unit square -> Ellipsoid implementation
    // a = b = radius = 1.
    Coordinate radius = 1.;
    Ellipsoid circle( radius, radius );

    master_mesh_ = master_mesh_->refine( refinements );
```

```
                    adapt_boundary_to_function( master_mesh_, circle );
                }

            }
            ++refinement_level_;
        }

        if ( !is_done_ )
        {
            // Add subdomain and remote index information needed by the library.
            std::vector<int> remote_index
                    (
                      master_mesh_->num_entities( master_mesh_->tdim( ) ),
                      -1
                      );
            AttributePtr remote_index_attr( new IntAttribute( remote_index ) );
            master_mesh_->add_attribute
                    ( "_remote_index_",
                      DIMENSION,
                      remote_index_attr );

            std::vector<int> subdomain
                    (
                      master_mesh_->num_entities( master_mesh_->tdim( ) ),
                      0
                      );
            AttributePtr subdomain_attr( new IntAttribute( subdomain ) );
            master_mesh_->add_attribute
                    ( "_sub_domain_",
                      DIMENSION,
                      subdomain_attr );
        }
    }
    if ( master_mesh_->num_entities( master_mesh_->tdim( ) ) > 100000 )
    {
        is_done_ = true;
    }
}
```

4 Program Output

HiFlow^3 can be executed in a parallel or sequential mode which influences the generated output data. Note that the log files can be viewed by any editor. This example for the a-posteriori estimation just works sequentially.

4.1 Sequential Mode

Executing the program sequentially by **./poisson_adaptive** following output data is generated.

- Mesh/geometry data:
  - **poisson_adaptive_mesh.pvtu** Global mesh (parallel vtk-format).
  - **poisson_adaptive_mesh_0.vtu** Global mesh owned by process 0 (vtk-format) containing the mesh information.
- Solution data.

- **solutionX.vtu** Solution of the poisson problem for refinement level X=3, 4, 5, 6, 7 and 8 (vtk-format).

- Log files:
  - **poisson_adaptive_debug_log** is a list of errors helping to simplify the debugging process. This file keeps empty if the program runs without errors.
  - **poisson_adaptive_info_log** is a list of parameters and some helpful informations to control the program as for example information about the residual of the linear and non-linear solver used.

- Terminal output: The global errors in L^2 norm and H^1 seminorm are listed for the different refinement levels.

4.2 Visualizing the Solution

HiFlow^3 only generates output data, see Section 3.5.8, but does not visualize. The mesh/geometry data as well as the solution data can be visualized by any external program which can handle the vtk data format as e.g. the program paraview [3].

5 Numerical Example

There are many different possibilities concerning the adaption of our program to a given problem. We just have to know the exact solution and that is the crux. Alternatively it is not feasible to calculate the exact H^1-error. Also you are able to compare the standard error estimator with the constructed error estimator for domains with bent boundaries.
In the following example our domain is the unit circle. We will test the constructed error estimator on the unit circle by solving the Poisson equation with homogeneous Dirichlet boundary values with a function f, which has a steep gradient especially at the boundary. We consider the following Poisson equation for $k \in \mathbb{N}$:

$$-\Delta u = 64k^2 \cdot (x^2+y^2)^{4k-1} \qquad \text{in } \Omega = \{\mathbf{R}^2 | x^2+y^2 = 1\} \tag{5}$$

$$u = 0 \qquad \text{on } \partial\Omega \tag{6}$$

$$\Rightarrow u(x,y) = -(x^2+y^2)^{4k} + 1. \tag{7}$$

The bigger we choose k, the steeper the gradient gets on the boundary. We can see that the additive residuals ρ_A and ρ_S have higher values if the gradient is steep on the boundary. To confirm this we will compare the residuals ρ_A and ρ_S for $k = 3$ and $k = 10$ in (3).
One can see in Figure 3 that the residuals for $k = 3$ are lying under the residuals for $k = 10$ from a certain number of cells. The difference in ρ_S is obvious already on the coarser grids. Long-term we can see, that ρ_A for $k = 10$ lies over ρ_A for $k = 3$. This effect is probably related to the approximation of S.
We now fix $k = 10$ in the following considerations. The analytical solution is depicted in Figure 4. As we can see, there the error estimator focuses very strongly on the boundary, because on account of the steep gradient near the boundary the error dominates in this area.
In Figure 5 one can see, that especially at the beginning ρ_A and ρ_S have a high percentage of the estimated error. Compared to the initial H^1-error the estimated error is by far too high. In fact, it seems as if the additive terms negatively influence the error estimator. Positively though is that the efficiency convergences along the refinement process.

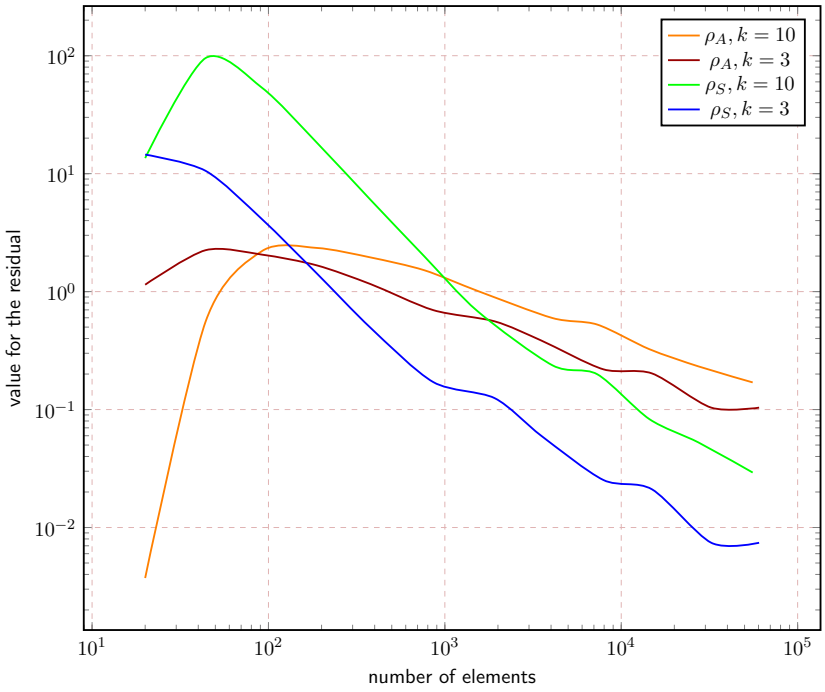

Figure 3: comparison of the additive terms for $k = 3$ and $k = 10$, f steep gradient on the boundary

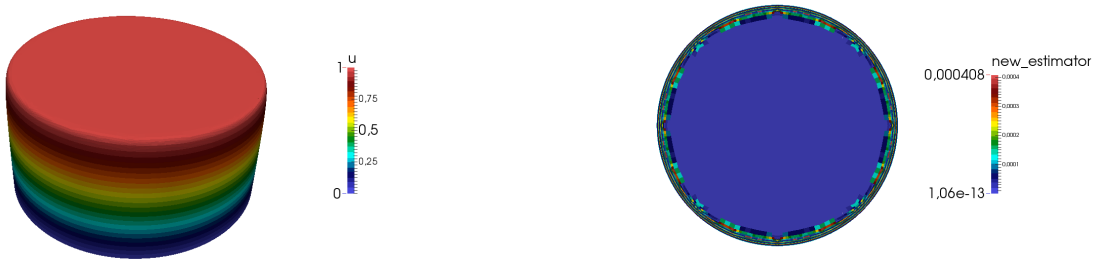

Figure 4: solution u on the unit circle for a right hand side with a steep gradient on the boundary, $k = 10$ (left), Plot in *Paraview* of the grid in the last step of refinement, refined with the extended error estimator, f steep gradient on the boundary, 55628 elements (right)

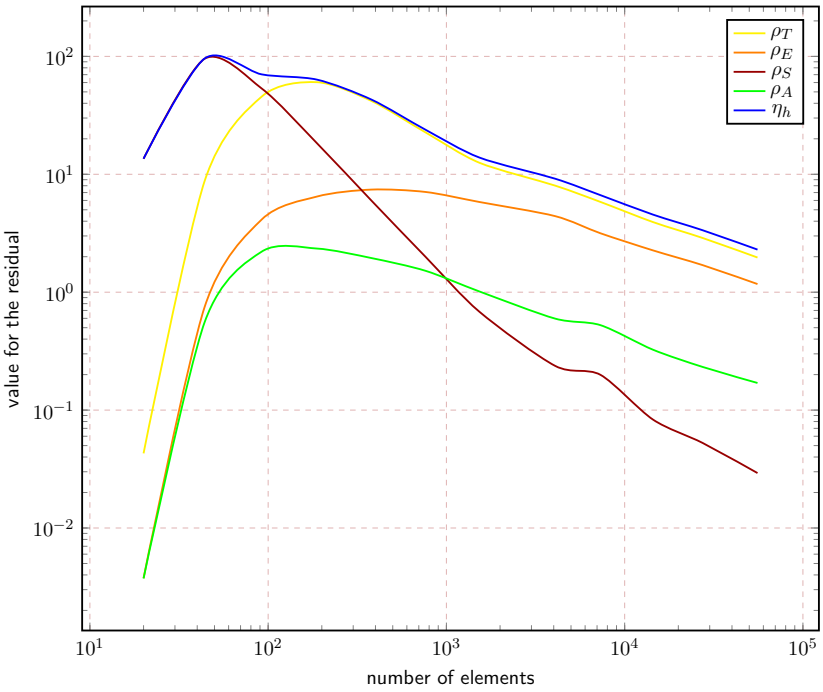

Figure 5: convergence of the residuals, f steep gradient on the boundary

5.1 Quality of the extended a-posteriori error estimator

We want to check the quality of the extended error estimator in comparison with the standard error estimator, depicted in Figure 6. The efficiency in Figure 6 on the right, preserved in a long run.
As a consequence the relevance of the additive terms depends on the problem.

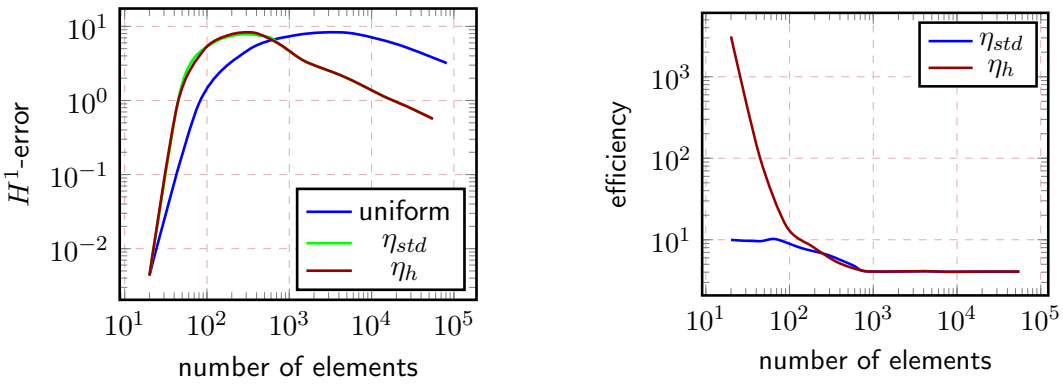

Figure 6: comparison of the H^1-error and the efficiency, f steep gradient on the boundary

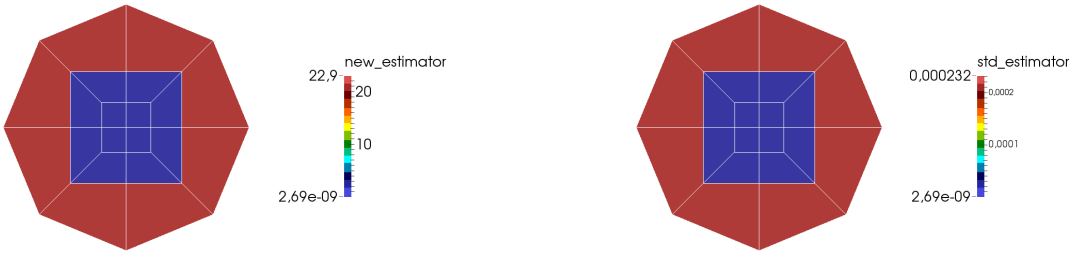

Figure 7: comparison of the extended and the standard error estimator in the first step of refinement, f steep gradient on the boundary, 20 elements

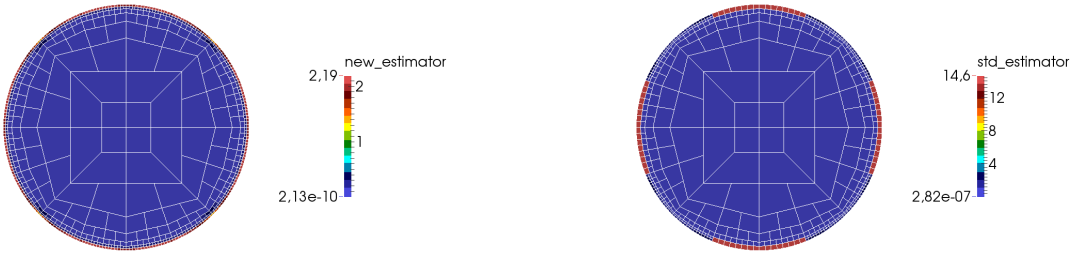

Figure 8: comparison of the extended and the standard error estimator in the sixth step of refinement, f steep gradient on the boundary, 764 and 572 elements, respectively

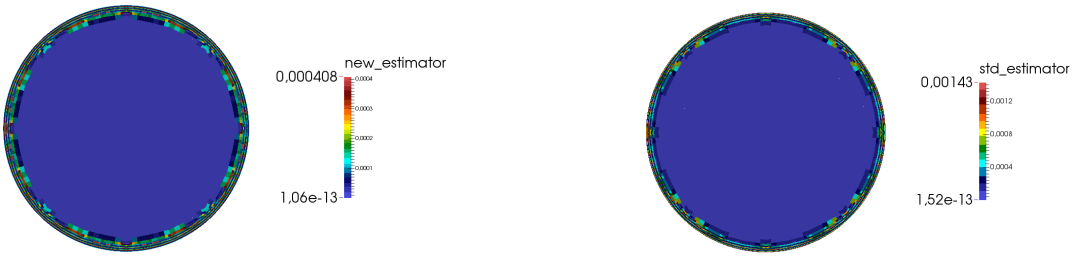

Figure 9: comparison of the extended and the standard error estimator in the last step of refinement, f steep gradient on the boundary, 55628 and 27092 elements, respectively

5.2 Evaluation

We could show that the error between the continuous and the discrete solution in essence does not increase, if we approximate a convex domain with bent boundaries with a polygonal domain and use linear Finite Elements, see [6, page 139].
The expected linear convergence of the constructed error estimator could be confirmed. Yet not answered is the question, if the extended error estimator really estimates the error better than the standard error estimator on domains with bent boundaries.

To extend the theory of a residual based error estimator for bent boundaries one would need to deal with the concave case in a next step. If $\Omega_h \not\subset \Omega$, then $V_h \not\subset V$ and consequently we do not have a conform ansatz. The analysis of this approximation turns out to be more complicated as in the conform case. However, one can show optimal rate of convergence in the latter case, too, see [7, page 89].

References

[1] http://www.mcs.anl.gov/research/projects/mpi.

[2] William D. Gropp. *MPI - Eine Einführung*. Oldenbourg, 2007.

[3] Amy Henderson Squillacote. *The ParaView guide: a parallel visualization application*. Kitware, [Clifton Park, NY], 2007.

[4] Katrin Mang. *A posteriori Fehlerschätzer und Adaption auf Gebieten mit gekrümmten Rändern.* Bachelor thesis, 12. August 2015.

[5] Thomas Richter. *Die Finite Elemente Methode für partielle Differentialgleichungen.* Lecture notes, Summer term 2014.

[6] Gerhard Dziuk. *Theorie und Numerik partieller Differentialgleichungen.* De Gruyter Studium, [Berlin/New York], 2010.

[7] Rolf Rannacher. *Numerische Mathematik 2 (Numerik partieller Differentialgleichungen).* Lecture notes, Winter term 2008.