Tutorial

E. Ketelaer, S. Ronnås, A. Helfrich-Schkarbanenko, C. Straub

Dirichlet Boundary Value Problem for Poisson Equation

modified on November 16, 2017

Version 1.3

http://www.hiflow3.org/

Contents

Applying HiFlow^3 for solving the Dirichlet boundary value problem for 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_tutorial.cc, poisson_tutorial.h) and a parameter file for the first numerical example (poisson_tutorial.xml) 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 Developer

First build and compile HiFlow^3. Go to the directory `/build/example/poisson`, where the binary **poisson_tutorial** is stored. Type **./poisson_tutorial**, to execute the program in sequential mode. To execute in parallel mode with four processes, type **mpirun -np 4 ./poisson_tutorial**. In both cases, you need to make sure that the default parameterfile poisson_tutorial.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_tutorial** `/"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 only for the two dimensional case. However, it is easy to expand the theory to one or three dimensions.
Our aim is solving the Poisson problem with Dirichlet boundary condition in a domain $\Omega \subset \mathbb{R}^2$ with sufficiently smooth boundary $\partial\Omega$: Find $u \in C^2(\Omega) \cap C_0(\overline{\Omega})$ that satisfies

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

where $f \in C(\Omega)$ is given.

The theory of partial differential equations shows that the computation of classical solutions to (1) is difficult, see e.g. [7]. Especially the required assumptions on u, f and the general shape of Ω are causing these difficulties. So we need another form for representing solutions where we only need weaker assumptions on the regularity.

2.2 Weak Formulation

In the Hilbert space $L^2(\Omega)$ we can re-formulate equation (1) in a weaker sense. Let $u = u(x_1, x_2)$, $f = f(x_1, x_2)$ and $\phi = \phi(x_1, x_2)$. Find $u \in C_0^2(\Omega)$ such that for all test functions $\varphi \in C_0^\infty(\Omega)$ it holds:

$$\int_\Omega -\Delta u \, \varphi \, \mathrm{d}x = \int_\Omega f \varphi \mathrm{d}x \ .$$

Using Green's first identity and the Gaussian integral theorem we get

$$\int_\Omega \nabla u \cdot \nabla \varphi \, \mathrm{d}x - \int_{\partial\Omega} (\nabla u \cdot n) \varphi(x, y) \mathrm{d}\sigma = \int_\Omega f \varphi \mathrm{d}x \ , \tag{2}$$

where n is the (outer) normal vector on $\partial\Omega$. Since the test functions φ have compact support in Ω, and therefore $\varphi(x, y) = 0$ for $(x, y) \in \partial\Omega$, we get from (2) that

$$\int_\Omega \nabla u \cdot \nabla \varphi \, \mathrm{d}x = \int_\Omega f \varphi \mathrm{d}x \qquad \forall \varphi \in C_0^\infty(\Omega). \tag{3}$$

$C_0^\infty(\Omega)$ is dense in the Hilbert space $H_0^1(\Omega)$, or more precisely: Since Ω is bounded, for any $u \in H_0^1(\Omega)$ there exists a sequence of functions $u_m \in C_0^\infty(\Omega) \cap H_0^1(\Omega)$ such that

$$u_m \to u \textit{ in } H_0^1(\Omega). \tag{4}$$

(See [7, Section 5.3.2 Theorem 2] for further details and the proof). So by approximation, we derive that the identity (3) holds with the smooth function φ replaced by any $\varphi \in H_0^1(\Omega)$. Consequently, the resulting equation makes even sense, i.e. the integrals exist, if we loosen the assumptions on u to $u \in H_0^1(\Omega)$ and those on f to $f \in L^2(\Omega)$.
Such an u is called $weak$ solution of (1). The bilinear form

$$a : H_0^1(\Omega) \times H_0^1(\Omega) \to \mathbb{R}, \quad a(u, \varphi) = \int_\Omega \nabla u \cdot \nabla \varphi \, \mathrm{d}x$$

is continuous and elliptic. The linear form $f : H_0^1(\Omega) \to \mathbb{R}$, $f(\varphi) = \int_\Omega f \, \varphi \, \mathrm{d}x$ is continuous, see [3], Def. 2.4. Therefore, by Lax-Milgram lemma, the problem (3), i.e. the equation

$$a(u, \varphi) \ = \ f(\varphi) \quad \forall \varphi \in H_0^1(\Omega), \tag{5}$$

has one and only one solution $u \in H_0^1(\Omega)$, see [4, Chap 5.8] or [10] . The Cea lemma, see [8, Chap. 2.8, 2.5], represents the analogon of the Lax-Milgram theorem for finite dimensional spaces. That means, the coercivity of the bilinear form a transfers from $H_0^1(\Omega)$ onto finite dimensional subspace $H_h(\Omega) \subset H_0^1(\Omega)$. Hence, equation (5) can be solved by means of the finite element method. We use HiFlow^3 to find $u_h \in H_h(\Omega)$, which approximates the weak solution $u \in H_0^1(\Omega)$.

2.3 Classical versus Weak Formulation

2.3.1 Assumptions

If we take a closer look on the assumptions on u and f in the classical and in the weak formulation of the Poisson problem, we can see that in the weak formulation we "got rid" of the hard regularity assumptions on u and f. In the variational formulation we only need first derivatives of u and these only in the weak sense. Furthermore, f does not need to be continuous any more but only in $L^2(\Omega)$.
One might think that we now are looking for some completely different objects: one with regularity up to C^2 and one which only satisfies some identities in the integral sense. But they still are quite similar: If we assume that $f \in L^2(\Omega)$ and $\partial\Omega \in C^2$, then it holds that $u \in H^2(\Omega)$ [7, Chap 6.3, Theorem 4]. The imbedding theorem by Sobolev tells us that for $u \in H^2(\Omega)$ exists a representation of $u \in C^0(\Omega)$ [2, Chap 4]. We can even assume lesser regularization for the boundary $\partial\Omega$ to apply the imbedding theorem by Sobolev for elliptic operators, see e.g [4, Chap 9, Theorem 9.15] for further reading. Under certain circumstances weak and classical solution even coincide.

2.3.2 Classical and Weak Solution

From the derivation in section 2.2 it is clear that every classical (regular and $C^2(\Omega)$-smooth) solution of (1) also fulfills (5). So every classical solution is also a weak one.
The other way around let $u \in H_0^1(\Omega)$ be a weak solution fulfilling (5). Let furthermore $u \in C^2(\Omega) \cap C_0(\overline{\Omega})$ and additionally $f \in C(\Omega)$. Then it follows that

$$\int_\Omega f\varphi \mathrm{d}x = \int_\Omega \nabla u \cdot \nabla\varphi \,\mathrm{d}x = \int_\Omega -\Delta u\varphi \mathrm{d}x \qquad \forall\varphi \in C_0^\infty(\Omega)$$

and so u is a classical solution because all derivatives are classical ones.
So we have some hope to find classical solutions by finding weak solutions which are regular enough but "easier to find".

2.3.3 L^2-Norm, H^1-Norm and H^1-Seminorm

In this section one can see the definitions of the square of the L^2-norm, H^1-norm and H^1-seminorm [7], because we compute in section 3.5.5 the error of our approximation in the L^2-norm and H^1-seminorm.

$$\begin{aligned}
\|u\|_{L_2(\Omega)}^2 &:= \int_\Omega |u|^2 dx, \qquad L^2 - \textit{norm},\\
|u|_{H^1(\Omega)}^2 &:= \int_\Omega |\nabla u|^2 dx, \qquad H^1 - \textit{seminorm},\\
\|u\|_{H^1(\Omega)}^2 &:= \|u\|_{L_2(\Omega)}^2 + |u|_{H^1(\Omega)}^2, \qquad H^1 - \textit{norm}.
\end{aligned}$$

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, 3.5.5, 3.5.7, [5], [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 tutorial reads in the parameter file poisson_tutorial.xml, see section 3.2, which contains the parameters of the first numerical example, see section 5.1.This file is stored in `/hiflow/examples/poisson/`.
- Geometry data: The file containing the geometry is specified in the parameter file (poisson_tutorial.xml). In both numerical examples in section 5 we used **unit_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_tutorial.xml.

```
'
<Param>
  <Mesh>
    <Filename1>unit_line.inp</Filename1>
    <Filename2>unit_square.inp</Filename2>
    <Filename3>unit_cube.inp</Filename3>
    <InitialRefLevel>1</InitialRefLevel>
    <FinalRefLevel>8</FinalRefLevel>
    <FeDegree>1</FeDegree>
  </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>NoPreconditioning</Method>
    <MaxIterations>1000</MaxIterations>
    <AbsTolerance>1.0e-14</AbsTolerance>
    <RelTolerance>1.0e-8</RelTolerance>
    <DivTolerance>1.0e6</DivTolerance>
  </LinearSolver>
</Param>
```

3.3 Structure of the Poisson Tutorial

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

- run()

- build_initial_mesh()
- prepare_system()
  - DirichletZero-struct (poisson_tutorial.h)
- assemble_system()
  - LocalPoissonAssembler-class (poisson_tutorial.h)
    - ExactSol-struct (poisson_tutorial.h)
- solve_system()
- compute_error()
  - L2ErrorIntegrator-class (poisson_tutorial.h)
  - H1ErrorIntegrator-class (poisson_tutorial.h)
- visualize()
- 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_tutorial.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_tutorial_info_log");
        LogKeeper::get_log ( "info" ).set_target ( &( std::cout ) );
        //std::ofstream debug_log("poisson_tutorial_debug_log");
        LogKeeper::get_log ( "debug" ).set_target ( &( std::cout ) );
        std::ofstream error_log ( "poisson_tutorial_error_log" );
        LogKeeper::get_log ( "error" ).set_target ( &( std::cout ) );

        // Create application object and run it
        PoissonTutorial 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 PoissonTutorial (poisson_tutorial.cc).

```
,
// Main algorithm
    void run ( )
    {
        // Construct / read in the initial mesh.
        build_initial_mesh ( );
        // Main adaptation loop.
        while ( !is_done_ )
        {
            // 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.
            compute_error ( );
            // Visualize the solution and the errors.
            visualize ( );
            adapt ( );
        }
    }
```

3.5.2 build_initial_mesh()

This member function, defined in the class PoissonTutorial, reads the initial mesh (poisson_tutorial.cc), partitions and distributes mesh if indicated, and writes out the refined mesh of initial refinement level.

```
,
void PoissonTutorial::build_initial_mesh ( )
{
#ifdef USE_MESH_P4EST
    mesh::IMPL impl = mesh::IMPL_P4EST;
#else
    mesh::IMPL impl = mesh::IMPL_DBVIEW;
#endif

    // Read in the mesh on the master process. The mesh is chosen according to
    // the dimension of the problem.
    if ( rank_ == MASTER_RANK )
    {
        std::string mesh_name;

        switch ( DIMENSION )
        {
            case 1:
            {
                mesh_name =
                        params_["Mesh"]["Filename1"].get<std::string>
                            ( "unit_line.inp" );
                break;
            }
            case 2:
            {
                mesh_name =
                        params_["Mesh"]["Filename2"].get<std::string>
                            ( "unit_square.inp" );
                break;
            }
            case 3:
            {
                mesh_name =
                        params_["Mesh"]["Filename3"].get<std::string>
                            ( "unit_cube.inp" );
                break;
            }

            default: assert ( 0 );
        }
        std::string mesh_filename;
        if ( path_mesh.empty ( ) )
        {
            mesh_filename = std::string ( DATADIR ) + mesh_name;
        }
        else
        {
            mesh_filename = path_mesh + mesh_name;
        }

        std::vector<MasterSlave> period ( 0, MasterSlave ( 0., 0., 0., 0 ) );
        master_mesh_ = read_mesh_from_file ( mesh_filename, DIMENSION, DIMENSION,
                                             0, period, impl );

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

        if ( initial_ref_lvl > 0 )
        {
            master_mesh_ = master_mesh_->refine_uniform_seq ( initial_ref_lvl );
        }
        refinement_level_ = initial_ref_lvl;
    }

    // 1D parallel execution is not yet implemented.
    if ( DIMENSION == 1 )
    {
        mesh_ = master_mesh_;
    }
    else
    {
        MPI_Bcast ( &refinement_level_, 1, MPI_INT, MASTER_RANK, comm_ );
        SharedVertexTable shared_verts;

        int uniform_ref_steps;
        mesh_without_ghost_ = partition_and_distribute ( master_mesh_, MASTER_RANK,
                                                         comm_, &uniform_ref_steps,
                                                         impl );

        assert ( mesh_without_ghost_ != 0 );
        mesh_ = compute_ghost_cells ( *mesh_without_ghost_, comm_,
                                      shared_verts, impl );

        // Write out mesh of initial refinement level
        PVtkWriter writer ( comm_ );
        std::ostringstream name;
        name << "poisson_tutorial_mesh_" << refinement_level_ << ".pvtu";
        std::string output_file = name.str ( );
        writer.add_all_attributes ( *mesh_, true );
        writer.write ( output_file.c_str ( ), *mesh_ );
    }
}
```

struct DirichletZero

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

,

```
// 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 boundary face
        return std::vector<double>( coords_on_face.size ( ), 0.0 );
    }
};
```

3.5.3 assemble_system()

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

```
void PoissonTutorial::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 defined in poisson_tutorial.h implements the stiffness matrix and right-hand side locally for each cell.

```
// 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 );

        // compute local 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 ) );
            }
        }
    }

    double f ( Vec<DIMENSION, double> pt )
    {
        ExactSol sol;
        double rhs_sol;

        switch ( DIMENSION )
        {
            case 1:
            {
                rhs_sol = 4. * M_PI * M_PI * ( M * M ) * sol ( pt );
                break;
            }
            case 2:
            {
                rhs_sol = 4. * M_PI * M_PI * ( M * M + N * N ) * sol ( pt );
                break;
            }
            case 3:
            {
                rhs_sol = 4. * M_PI * M_PI * ( M * M + N * N + O * O ) *
                      sol ( pt );
                break;
            }
            default: assert ( 0 );
        }

        return rhs_sol;
    }
};
```

ExactSol struct

The struct ExactSol in poisson_tutorial.h implements the exact solution u given by (11) and the exact gradient ∇u.

,

```
struct ExactSol
{
```

```
double operator() ( const Vec<DIMENSION, double>& pt ) const
{
    const double x = pt[0];
    const double y = ( DIMENSION > 1 ) ? pt[1] : 0;
    const double z = ( DIMENSION > 2 ) ? pt[2] : 0;
    const double pi = M_PI;
    double solution;

    switch ( DIMENSION )
    {
        case 1:
        {
            solution = 10.0 * std::sin ( 2. * M * pi * x );
            break;
        }
        case 2:
        {
            solution = 10.0 * std::sin ( 2. * M * pi * x ) *
                              std::sin ( 2. * N * pi * y );
            break;
        }
        case 3:
        {
            solution = 10.0 * std::sin ( 2. * M * pi * x ) *
                              std::sin ( 2. * N * pi * y ) *
                              std::sin ( 2. * O * pi * z );
            break;
        }

        default: assert ( 0 );
    }
    return 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 z = ( DIMENSION > 2 ) ? pt[2] : 0;
    const double pi = M_PI;

    switch ( DIMENSION )
    {
        case 1:
        {
            grad[0] = 20. * M * pi * std::cos ( 2. * M * pi * x );
            break;
        }
        case 2:
        {
            grad[0] = 20. * M * pi * std::cos ( 2. * M * pi * x ) *
                      std::sin ( 2. * N * pi * y );
            grad[1] = 20. * N * pi * std::sin ( 2. * M * pi * x ) *
                      std::cos ( 2. * N * pi * y );
            break;
        }
        case 3:
        {
                grad[0] = 20. * M * pi * std::cos ( 2. * M * pi * x ) *
                        std::sin ( 2. * N * pi * y ) *
                        std::sin ( 2. * O * pi * z );
                grad[1] = 20. * N * pi * std::sin ( 2. * M * pi * x ) *
                        std::cos ( 2. * N * pi * y ) *
                        std::sin ( 2. * O * pi * z );
                grad[2] = 20. * O * pi * std::sin ( 2. * M * pi * x ) *
                        std::sin ( 2. * N * pi * y ) *
                        std::cos ( 2. * O * pi * z );
                break;
            }
            default: assert ( 0 );
        }

        return grad;
    }
};
```

3.5.4 solve_system()

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

```
void PoissonTutorial::solve_system ( )
{
    LinearSolver<LAD>* solver_;
    LinearSolverFactory<LAD> SolFact;
    solver_ = SolFact.Get (
            params_["LinearSolver"]["Name"].get<std::string>( "CG" ) )->
            params ( params_["LinearSolver"] );
    solver_->SetupOperator ( *matrix_ );
    solver_->Solve ( *rhs_, sol_ );
    delete solver_;
}
```

3.5.5 compute_error()

This member function in poisson_tutorial.cc computes the error between the approximated and the exact solution mentioned in section 5 in the L^2-norm and H^1-seminorm, see section 2.3.3.

```
void PoissonTutorial::compute_error ( )
{
    // prepare sol_ for post processing
    sol_->UpdateCouplings ( );

    L2_err_.clear ( );
    H1_err_.clear ( );

    // Compute square of the L2 error on each element, putting the
    // values into L2_err_.
    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_ ) );
    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_.

    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_ ) );
    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 ) );

    if ( rank_ == MASTER_RANK )
    {
        std::cout << "Global␣L2␣error␣=␣" << std::sqrt ( global_L2_err ) << "\n"
                  << "Global␣H1␣error␣=␣" << 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_tutorial.h).

```
,
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 );
            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_tutorial.h).

```
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_;
};
```

3.5.6 visualize()

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

```
void PoissonTutorial::visualize ( )
{
    // Setup visualization object.
    int num_intervals = 2;
    ParallelCellVisualization<double> visu ( space_, num_intervals, comm_,
                                             MASTER_RANK );

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

    std::vector<double> remote_index ( mesh_->num_entities ( mesh_->tdim ( ) ),
                                       0 );
    std::vector<double> sub_domain ( mesh_->num_entities ( mesh_->tdim ( ) ),
                                     0 );
    std::vector<double> material_number ( mesh_->num_entities ( mesh_->tdim ( ) ),
                                          0 );

    for ( mesh::EntityIterator it = mesh_->begin ( mesh_->tdim ( ) );
          it != mesh_->end ( mesh_->tdim ( ) );
          ++it )
    {
        int temp1, temp2;
        if ( DIMENSION > 1 )
        {
            mesh_->get_attribute_value ( "_remote_index_", mesh_->tdim ( ),
                                         it->index ( ),
                                         &temp1 );
            mesh_->get_attribute_value ( "_sub_domain_", mesh_->tdim ( ),
                                         it->index ( ),
                                         &temp2 );
            remote_index.at ( it->index ( ) ) = temp1;
            sub_domain.at ( it->index ( ) ) = temp2;
        }
        material_number.at ( it->index ( ) ) =
            mesh_->get_material_number ( 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 ( remote_index, "_remote_index_" );
    visu.visualize_cell_data ( sub_domain, "_sub_domain_" );
    visu.visualize_cell_data ( material_number, "Material␣Id" );
    visu.write ( name.str ( ) );
}
```

3.5.7 adapt()

The member function adapt() modifies the space through refinement (poisson_tutorial.cc).

```
void PoissonTutorial::adapt ( )
{
    // 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
            {
                mesh_ = mesh_->refine ( );
                ++refinement_level_;
            }
        }
    }
    else
    {
#ifdef USE_MESH_P4EST
        const int final_ref_level = params_["Mesh"]["FinalRefLevel"].
             get<int>( 6 );
        if ( refinement_level_ >= final_ref_level )
        {
            is_done_ = true;
        }
        else
        {
            mesh_without_ghost_ = mesh_without_ghost_->refine ( );
            ++refinement_level_;
        }
        if ( !is_done_ )
        {
            SharedVertexTable shared_verts;
            std::vector<MasterSlave> period ( 0, MasterSlave ( 0., 0., 0., 0 ) );
            mesh_ = compute_ghost_cells ( *mesh_without_ghost_, comm_,
                                          shared_verts, mesh::IMPL_P4EST );
        }
#else
#    ifdef WITH_PARMETIS
        const int final_ref_level = params_["Mesh"]["FinalRefLevel"].
              get<int>( 6 );
        if ( refinement_level_ >= final_ref_level )
        {
            is_done_ = true;
        }
        else
        {
            mesh_without_ghost_ = mesh_without_ghost_->refine ( );
            ++refinement_level_;
        }
        if ( !is_done_ )
        {
            // Repartition the new mesh.
            ParMetisGraphPartitioner parmetis_partitioner;
            MeshPtr local_mesh = repartition_mesh ( mesh_without_ghost_, comm_,
                                                    &parmetis_partitioner );
            assert ( local_mesh != 0 );
            mesh_.reset ( );
            SharedVertexTable shared_verts;
            mesh_ = compute_ghost_cells ( *local_mesh, comm_, shared_verts );
            local_mesh.reset ( );
        }
#    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 ( );
                ++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;
            mesh_ = compute_ghost_cells ( *local_mesh, comm_, shared_verts );
        }
#    endif
#endif
        PVtkWriter writer ( comm_ );
        std::ostringstream name;
        name << "poisson_tutorial_mesh_" << refinement_level_ << ".pvtu";
        std::string output_file = name.str ( );
        writer.add_all_attributes ( *mesh_, true );
        writer.write ( output_file.c_str ( ), *mesh_ );
    }
}
```

4 Program Output

HiFlow^3 can be executed in a parallel or sequential mode which influence the generated output data. Note that the log files can be viewed by any editor.

4.1 Parallel Mode

Executing the program in parallel, for example with four processes by **mpirun -np 4 ./poisson_tutorial** generates following output data.

- Mesh/geometry data:
  - **poisson_tutorial_mesh.pvtu** Global mesh of initial refinement level with default value = 3 (parallel vtk-format). It combines the local meshes of sequential vtk-format owned by the different processes to the global mesh.
  - **poisson_tutorial_mesh_X.vtu** local mesh of initial refinement level (default value = 3) owned by process X for X=0, 1, 2 and 3 (vtk-format).
- Solution data:
  - **solutionX.pvtu** Solution of the poisson problem for refinement level X=3, 4, 5, 6, 7 and 8 (parallel vtk-format). It combines the local solutions owned by the different processes to a global solution.
  - **solutionX_Y.vtu** Local solution of the poisson problem for refinement level X=3, 4, 5, 6, 7 and 8 of the degrees of freedoms which belong to cells owned by process Y, for Y=0, 1, 2 and 3 (vtk-format).
- Log files:
  - **poisson_tutorial_debug_log** Log file listing errors helping to simplify the debugging process. This file is empty if the program runs without errors.
  - **poisson_tutorial_info_log** Log file listing parameters and some helpful information 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 Sequential Mode

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

- Mesh/geometry data:
  - **poisson_tutorial_mesh.pvtu** Global mesh (parallel vtk-format).
  - **poisson_tutorial_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_tutorial_debug_log** is a list of errors helping to simplify the debugging process. This file keeps empty if the program runs without errors.

- **poisson_tutorial_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.

4.3 Visualizing the Solution

HiFlow^3 only generates output data, see section 3.5.6, 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 [6].

5 Examples

5.1 First Example

This example is implemented in one to three dimensions. All three cases are discussed in the following.

5.1.1 One Dimension

To be able to verify the results we choose in this example the right-hand side

$$f(x,y) \;=\; 4\pi^2 M^2 \sin(2\pi M x) \text{ with } M \in \mathbb{N}, \tag{6}$$

and the homogeneous Dirichlet data on the boundary of the unit line $\Omega := (0,1)$. The analytical solution is obviously

$$u(x,y) \;=\; \sin(2\pi M x). \tag{7}$$

Obviously, $u \in C^2(\Omega) \cap C_0(\overline{\Omega})$ holds since $\sin$ is an analytic function. Considering u and f we see that u is an eigenfunction of the operator $-\Delta u$ on Ω with the eigenvalue $4\pi^2 M^2$.
Hence, the error in the L^2-norm $\|u - u_h\|_{L^2(\Omega)}$ as well as the error in the H^1-seminorm $|u - u_h|_{H^1(\Omega)}$ between the finite element approximation and the exact solution can be computed, see section 3.5.5. Similar results can be shown for the two dimension and the three dimension case. Fig. 1 - 3 show the visualized solution for refinement level 4, 6 and 8 for $M = 4$.

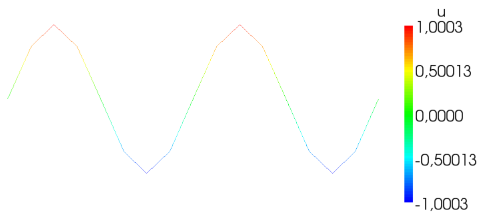


Figure 1: Global solution in 1D for refinement level 4 of the mesh.

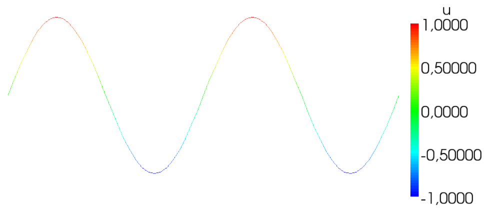


Figure 2: Global solution in 1D for refinement level 6 of the mesh.

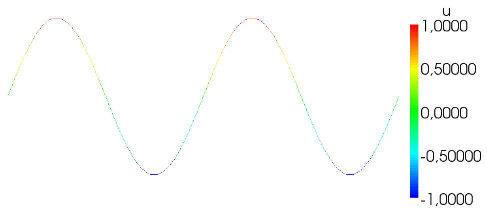


Figure 3: Global solution in 1D for refinement level 8 of the mesh.

5.1.2 Two Dimensions

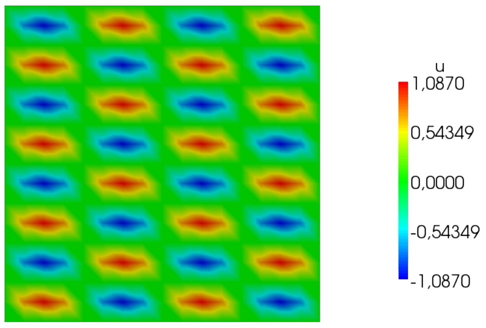
We expand the right-hand side to two dimensions, i.e.

$$f(x,y) \;=\; 4\pi^2(M^2+N^2)\sin(2\pi Mx)\sin(2\pi Ny) \text{ with } M,N\in\mathbb{N}, \tag{8}$$

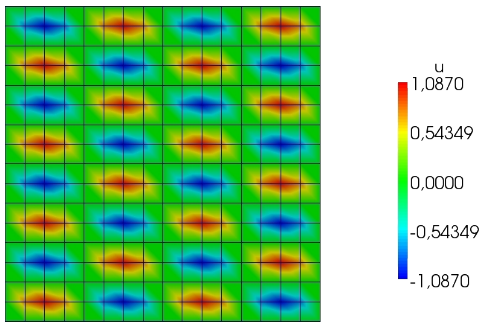
and choose again homogeneous Dirichlet data on the boundary. Ω is now the unit square $\Omega := (0,1)\times(0,1)$. By means of the separation methods we derive the analytical solution

$$u(x,y) \;=\; \sin(2\pi Mx)\sin(2\pi Ny). \tag{9}$$

Fig. 4 - 6 show the visualized solution for refinement level 4, 6 and 8 for $M=2$ and $N=4$. On the right side the corresponding mesh is included in the picture.



(a) Solution.



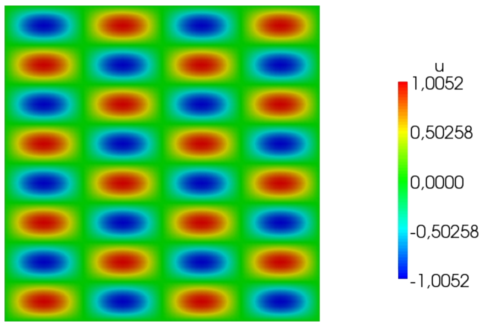
(b) Solution with mesh.

Figure 4: Global solution in 2D for refinement level 4 of the mesh.

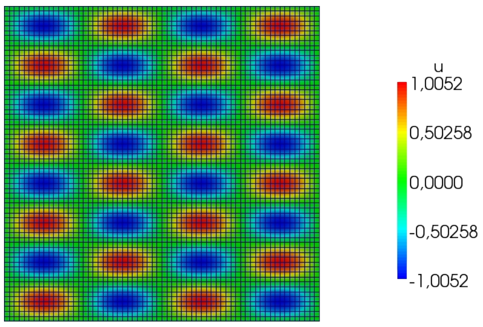
5.1.3 Three Dimensions

Now we do the same thing for three dimensions. The right-hand side changes to

$$f(x,y,z) \;=\; 4\pi^2(M^2+N^2+O^2)\sin(2\pi Mx)\sin(2\pi Ny)\sin(2\pi Oz) \text{ with } M,N,O\in\mathbb{N}, \tag{10}$$

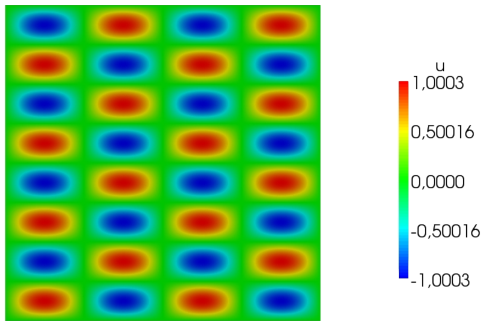


(a) Solution.

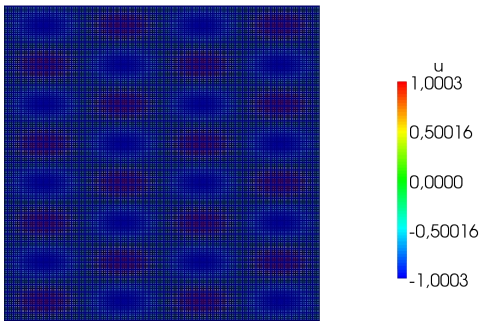


(b) Solution with mesh.

Figure 5: Global solution in 2D for refinement level 6 of the mesh.



(a) Solution.



(b) Solution with mesh.

Figure 6: Global solution in 2D for refinement level 8 of the mesh.

Ω is now the unit cube $\Omega := (0,1) \times (0,1) \times (0,1)$. Again, by means of the separation methods we derive the analytical solution

$$u(x,y) \quad = \quad \sin(2\pi Mx)\sin(2\pi Ny)\sin(2\pi Oz). \tag{11}$$

Fig. 7 and 8 show the visualized solution for refinement levels 3 and 5 for $M = 4$, $N = 2$ and $O = 1$, including the corresponding mesh.

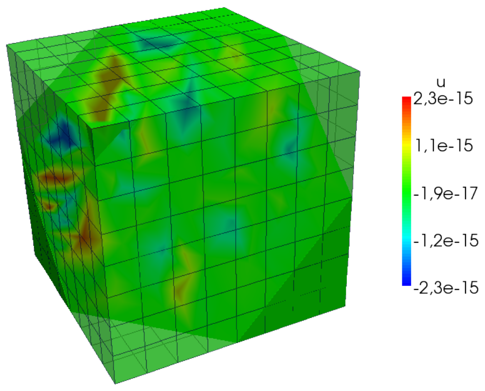


Figure 7: Global solution in 3D for refinement level 3 of the mesh.

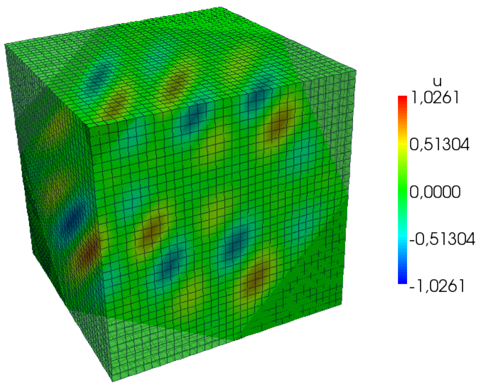


Figure 8: Global solution in 3D for refinement level 5 of the mesh.

5.2 Second Example

Now we take a look on a second example in two dimensions, where we change the right-hand side f to

$$f(x,y) = 32[x(1-x) + y(1-y)].$$

In this case, the exact solution to the Poisson problem (1) is given by

$$u(x,y) = 16x(1-x)y(1-y). \tag{12}$$

Obviously, u is eigenfunction of the Laplace operator on a square Ω. In contrast to the first example the solution computed on the fine grid with 1024 elements in (9) looks perfectly smooth. The reason for this observation lies in the nature of the solution: the solution u given by (12) is a polynomial of second order and does not oscillate as much as the solution in the first example. The solution in this example can be interpolated much better and more accurate by linear elements than the solution in the first example.

Our expectation is that increasing the polynomial degree of the ansatz functions affects the

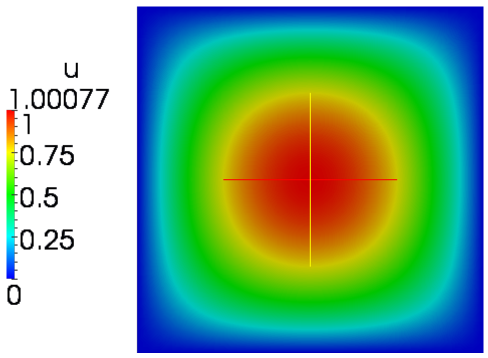


Figure 9: Global solution for refinement level 5 of the mesh for the second example.

quality of the approximation less in the second example as in the first one. The dependence upon the parameters polynomial degree p and level of refinement, measured by the quantity h, will be investigated in the next section.

6 Quality of the Approximation

An important aspect of the finite element method is that it approximates the space in which we are looking for the exact solution to the weak formulation of the partial differential equation, i.e. in our case $H_0^1(\Omega)$. Compared the the finite difference method where you approximate the occuring derivatives by difference quotients it is a completely different approach which affects practical aspects:

- We can now connect the approximating subspace to the discretization of the domain Ω and vice versa.
- As a consequence, the approximating subspace is customized to the domain so that we have the flexibility to handle complex domains.
- A good choice of the discretization of Ω can improve the quality of the approximation significantly.

For our case of conforming elements and piecewise polynomial basis functions this naturally leads to two parameters which we can vary to get a better approximation $u_h \in H_h(\Omega) \subset H_0^1(\Omega)$.

6.1 Convergence Theory

We want that the solution u_h converges in some sense to the exact solution of the infinite dimensional weak problem. In the case of the finite element method with conforming Lagrangian ansatz functions, there are two parameters that we can vary:

- The fineness of the grid which is measured by
$$h := \max_{K \in \mathcal{T}^h} diam K,$$
where $\mathcal{T}^h$ denotes the set of all elements K in the triangulation.
- The polynomial degree p of the Lagrangian ansatz functions.

We can expect that the errors in the approximation depend on these parameters, and that the quality of the approximation can somehow be measured in dependence of them. Cea lemma gives us an error estimate reduced to an interpolation estimate

$$\|u - u_h\|_X \leq C \inf_{v_h \in X_h} \|u - v_h\|_X \leq C\|u - I_h u\|_X,$$

where $X = H_0^1(\Omega)$ and $X^h \subset X$ is the conforming finite element space and I_h denotes the interpolation operator.
Furthermore, it can be shown that

$$\|u - I_h u\|_{L^2} \leq Ch^{p+1}\|u\|_{H^{p+1}}, \quad \|\nabla(u - I_h u)\|_{L^2} \leq Ch^p\|u\|_{H^{p+1}}, \tag{13}$$

see e.g. [3].

6.2 Dependence on h

First, we want to anticipate that in both examples the approximate solution converges with increasing refinement level to the exact solution. But the interesting question is about the differences how they converge.

6.2.1 First Example

For this experiment we fixed the polynomial degree to 1, i.e. we used linear elements, we worked in two dimensions and refined the grid up to six times. For the parameters we chose the case $M = N = 1$. The global error in the L^2-norm and in the H^1-norm is plotted in fig. 10. For the number of degrees of freedom we chose a logarithmic scale, because with each refinement step the number of degrees of freedom increases exponentially.

In fig. 10, one can clearly see the expected quadratic behaviour in the h dependence which leads

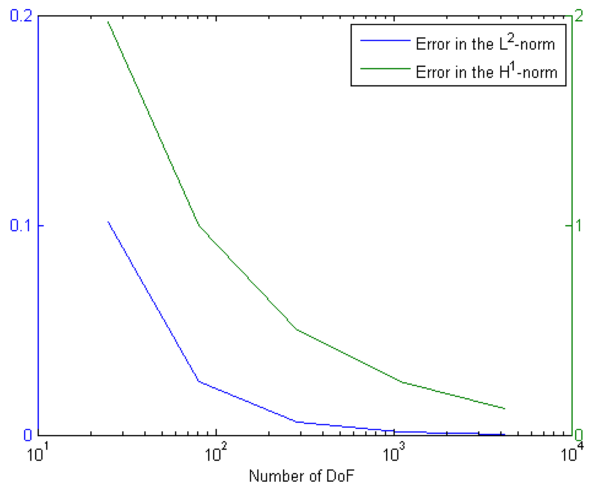


Figure 10: Global error in the L^2- and H^1-norm as function of the number of DoF for polynomial degree 1 for $M = N = 1$.

to a quick decrease of the error in the L^2-norm. Also in the plot of the error in the H^1-norm, the expected linear decrease with respect to h can be seen. This especially means that the error in the gradient dominates the error in the full H^1-norm.

6.2.2 Second Example

The plot of the global error in the L^2- and H^1-norm for the second example from section 5.2 can be seen in fig. 11. The plot configurations are the same as in the first example, i.e. we chose a logarithmic scale for the number of degrees of freedom. The observations in this example are quite the same as in the first. Especially the plot matches with our expectations about the decrease of the errors.

6.3 Dependence on the Polynomial Degree p

6.3.1 First Example

For the experiments within this section, we keep the mesh fixed and the polynomial degree is increased, see fig. 12. For this experiment we chose a constant refinement level of three, i.e. $h = \frac{1}{8}$. As we can see, the plots reflect the expected behaviour, because the error decreases exponentially.

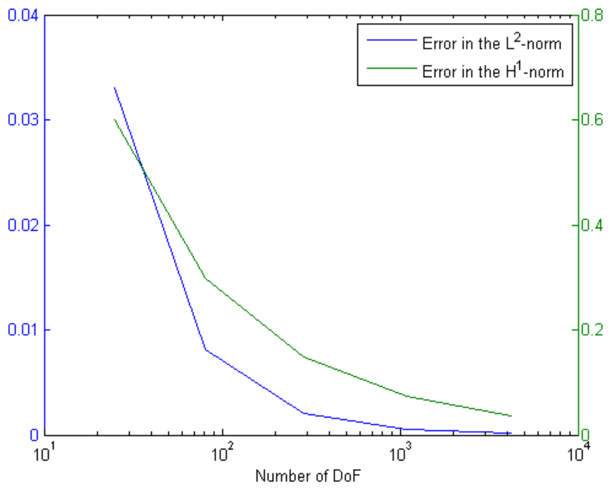


Figure 11: Global error in the L^2- and H^1-norm as function of the number of DoF for polynomial degree 1 for the second example.

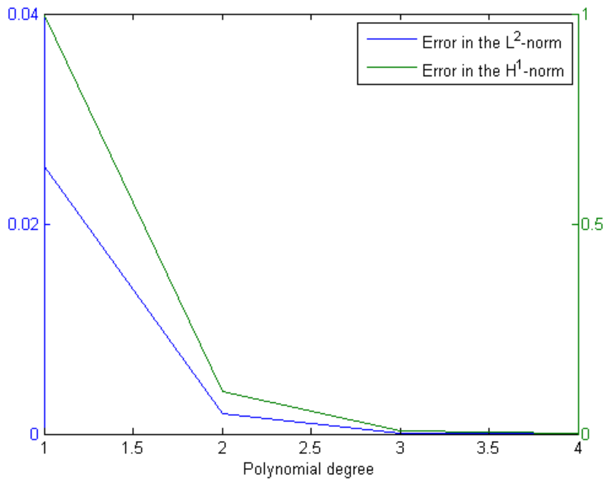


Figure 12: Global error in the L^2- and H^1-norm as function of the polynomial degrees for the first example for $M = N = 1$.

6.3.2 Second Example

Looking at fig. 13, we can see the comparison of the global error in the H^1-norm for the two different examples with fixed refinement level of three. Note that $M = N = 1$ holds for the first example. In fig. 13 one can see that the global error in the H^1-norm for the second example is for polynomial degree of two zero. This can be explained by the fact, that the right-hand side in the second example is a polynomial of degree two.

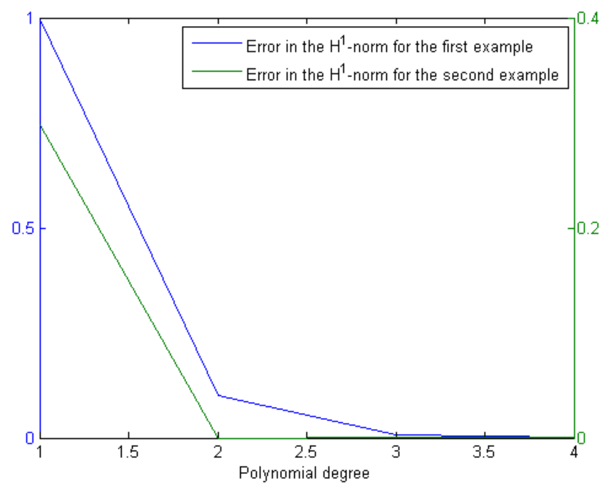


Figure 13: Global error in the H^1-norm as function of the polynomial degrees for the two different examples.

7 The Condition Number of the Stiffness Matrix

7.1 Dependence on the Level of Refinement and on the Polynomial Degree

It is very difficult to derive theoretical results on the dependence of the condition number of the stiffness matrix on the level of refinement and the polynomial degree. There are several reasons:

- The number of elements per row in the matrix depends on the geometry. It makes great differences if we choose triangles, rectangles etc. .
- Furhtermore, depending on the angles in one element, each degree fo freedom might have another number of adjacent neighbour nodes.
- Depending on the above two points, the number of adjacent neighbours additionally depends on the degree of the ansatz functions.

As a consequence, all three of these points determine the elements of the stiffness matrix so that it is more or less impossible to derive theoretical results except for special cases.
The results of some numerical experiments indicate that there is a dependence of the form

$$\kappa = O\left(\left(\frac{1}{h}\right)^{p+1}\right),$$

where κ denotes the condition numer of the stiffness matrix.

7.2 The CG-Method and the Condition Number

Let $A \in \mathbb{R}^{n\times n}$ be the stiffness matrix of the Poisson problem and $x_0 \in \mathbb{R}^n$ any initial vector to the CG-method. Then it is known from numerical analysis that the error in the k-th iteration is extimated by

$$\|x_k - x^*\|_A \leq 2\left(\frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1}\right)^k \|x_0 - x^*\|_A,$$

where κ denotes the condition number of the stiffness matrix, x^* the exact solution, x_k the k-th iterate in the CG-method and

$$\|x\|_A := \sqrt{x^T A x}$$

is the A-norm of $x \in \mathbb{R}^n$ (see e.g. [3, Chap IV Theorem 3.7]). Let $\varepsilon > 0$ denote the given tolerance to the CG-method. Then we want that

$$\|x_k - x^*\|_A \leq 2\left(\frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1}\right)^k \|x_0 - x^*\|_A \leq \varepsilon \|x_0 - x^*\|_A$$

holds and it follows

$$k \geq \frac{\log(\frac{\varepsilon}{2})}{\log(\frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1})}.$$

8 Parallelization

All content about laws on parallel performance can be found in [9].

8.1 Speedup and Efficiency

Let $T(P)$ the runtime of a parallel algorithm on P processors and let T_S the runtime of the best sequential implementation of the algorithm. Then the speedup $S(P)$ and the efficiency $E(P)$ are defined by

$$\begin{aligned} S(P) &:= \frac{T_S}{T(P)}, \\ E(P) &:= \frac{S(P)}{P}. \end{aligned} \tag{14}$$

The efficiency is a measure of the utilized capacity of the processors. In an ideal world we want $S(P) \approx P$ and $E(P) \approx 1$. In practise $T(1)$ is often used instead of T_S and it holds $T(1) \geq T_S$. So for some algorithms one can obtain $S(P) > P$ and $E(P) > 1$ due to cache effects, data locality etc. .
When T_S or $T(1)$ can not be determined because of the problem size the incremental speedup

$$S_I(P) := \frac{T(\frac{P}{2})}{T(P)} \quad (\leq 2)$$

is considered instead.

8.2 Amdahl's Law

When we parallelize an algorithm, we want to save runtime for the same problem size. But in practise most algorithms won't scale perfectly because they contain sequential parts which can not be parallelized. An estimation about the speedup that we can expect is given by Amdahl's law.
Let $T(1)$ the time for the sequential run of the program. Let f with $0 \leq f \leq 1$ the part of the program which is not parallelizable. Then it holds for the runtime $T(P)$ of the parallel execution on P processors

$$T(P) = f \cdot T(1) + \frac{1-f}{P} T(1). \tag{15}$$

Inserting (15) into (14) we obtain

$$S(P) = \frac{T(1)}{T(P)} = \frac{1}{f + \frac{1-f}{P}}. \tag{16}$$

Now, one can ask what happens if we increase the number of processors more and more? Do we get infinite speedup? Letting $P \to \infty$ in (16) we see that the speedup saturates with limit

$$\lim_{P\to\infty} S(P) = \frac{1}{f}.$$

So the maximal speedup is given by the sequential part of the program.
Amdahl's law is often beeing critisised because it does not consider the problem size N. If the sequential part of the program is independ on the problem size and only the parallel part depends on it then Amdahl's law is too pessimistic. Another approach that avoids this problem is the law by Gustafson.

8.3 Gustafson's Law

Let the number of processors P fixed. Let $f(N)$ the sequential part of the algorithm for the problem size N. Furthermore, let $T(P) = 1$ normed. Then it holds:

$$T(P) = f(N) + (1 - f(N)), \quad T(1) = f(N) + (1 - f(N)) \cdot P,$$
$$S(P) = \frac{T(1)}{T(P)} = f(N) + (1 - f(N)) \cdot P$$

where we used $T(P) = 1$ in the last equation. If it holds

$$\lim_{N\to\infty} f(N) = 0$$

then we obtain

$$\lim_{N\to\infty} s(N) = P$$

as desired. This is a realistic scenario for many problems.

References

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

[2] John J. F. Adams, Robert A. ; Fournier. *Sobolev spaces*. Pure and applied mathematics ; 140. Academic Press, Amsterdam, 2. ed., repr. edition, 2006.

[3] Dietrich Braess. *Finite Elemente : Theorie, schnelle Löser und Anwendungen in der Elastizitätstheorie*. Springer, Berlin, 4., überarb. und erw. aufl. edition, 2007.

[4] N.S.Trudinger D.Gilbarg. *Elliptic Partial Differential Equations of Second Order*. Springer, second edition, 1989.

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

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

[7] L.C.Evans. *Partial Differential Equations*. American Mathematical Society, second edition, 1998.

[8] L.R.Scott S.C.Brenner. *The Mathematical Theory of Finite Element Methods*. Springer, New-York, 1994.

[9] Jan-Philipp Weiss. *Paralleles Rechnen*. Lecture notes, Summer term 2011.

[10] W.McLean. *Strongly Elliptic Systems and Boundary Integral Equations*. Cambridge University Press, 2000.