R. Hiptmair
L. Scarabosio
C. Urzua Torres

Spring Term 2015

Numerical Methods for Partial Differential Equations

ETH Zürich
D-MATH

Homework Problem Sheet 8

Problem 8.1 Heat Conduction with Mass Term and Dirichlet BCs (Core problem)

This exercise deals with the variational formulation of a second-order elliptic boundary value problem (→ [NPDE, Section 2.9]), its approximate solution by means of Galerkin discretization based on linear Lagrangian finite elements on triangular meshes (→ [NPDE, Section 3.3]), and the implementation in DUNE and [NPDE, Section 3.6].In order to understand the treatment of Dirichlet boundary conditions, please recall the contents of [NPDE, Section 3.6.5] and study [NPDE, Code 3.6.114], [NPDE, Code 3.6.115], and [NPDE, Code 3.6.49].

The stationary heat equation with a linear reaction term (zero-order term) reads

$$-\Delta u(\boldsymbol{x}) + \underbrace{c(\boldsymbol{x})u(\boldsymbol{x})}_{\text{reaction term}} = f(\boldsymbol{x}) \quad \text{in } \Omega. \tag{8.1.1}$$

where the reaction coefficient $c(\boldsymbol{x})$ is uniformly positive and bounded on Ω, *cf.* [NPDE, Eq. (2.6.6)]. In addition to this we impose the homogeneous Dirichlet boundary condition $u = 0$ on $\partial\Omega$. The computational domain $\Omega \in \mathbb{R}^2$ is a pentagon as shown in Figure 8.1.

In this problem, we will extend the code from Problem 7.4 so that the full homogeneous Dirichlet problem for (8.1.1) is solved approximately using the 2D linear finite elements introduced in [NPDE, Section 3.3]. Template files for the new classes you will need to write are available in the lecture `svn` repository

`assignments_codes/assignment8/Problem1`

The idea is that you extend your own code, reason why it does not contain the files you already implemented in 7.4. Please do not forget to include them when you submit your work.

(8.1a) Make sure that you master the material of [NPDE, Section 3.3].

(8.1b) Study [NPDE, Section 2.6] and try to explain why the term $c(\boldsymbol{x})$ is called a "reaction term".

HINT: As you may remember from secondary school, high temperatures hasten most chemical reactions. Assume an endothermic reaction.

Solution: If we move the term to the right-hand side we get

$$-\Delta u(\boldsymbol{x}) = f(\boldsymbol{x}) - c(\boldsymbol{x})u(\boldsymbol{x}),$$

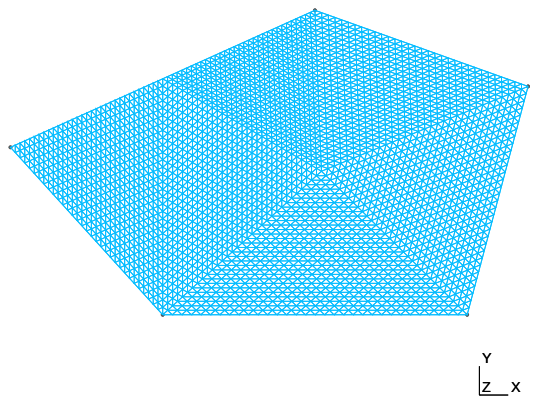


Figure 8.1: A triangulation of Ω.

so we can see that the reaction term acts as a heat source with opposite sign. That is, if $u < 0$, it acts as a heat *source*, and if $u > 0$ it acts as a heat *sink*. This is exactly what we expect to see if there is an endothermic reaction happening (a reaction that consumes heat).

(8.1c) Derive the variational formulation of the homogeneous Dirichlet problem for (8.1.1), that is, for the Dirichlet problem with all zero boundary conditions. Do not forget to specify the trial and test spaces.

HINT: As in [NPDE, Section 2.9] rely Green's first formula [NPDE, Thm. 2.5.9] and use Sobolev spaces.

Solution: We multiply with a function v, which is zero on $\partial\Omega$, and integrate:

$$\begin{aligned} -\int_\Omega v(\boldsymbol{x})\Delta u(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} + \int_\Omega c(\boldsymbol{x})u(\boldsymbol{x})v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} &= \int_\Omega f(\boldsymbol{x})v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x} \\ \int_\Omega [\operatorname{grad} u(\boldsymbol{x}) \cdot \operatorname{grad} v(\boldsymbol{x}) + c(\boldsymbol{x})u(\boldsymbol{x})v(\boldsymbol{x})]\,\mathrm{d}\boldsymbol{x} &= \int_\Omega f(\boldsymbol{x})v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}. \end{aligned} \tag{8.1.2}$$

The boundary term vanishes because $v = 0$ there. The variational formulation is then to find $u \in H_0^1(\Omega)$ (trial space) such that (8.1.2) holds for all $v \in H_0^1(\Omega)$ (test space).

(8.1d) Now assume that $c(\boldsymbol{x}) = \mathrm{const}$ and that we use linear finite elements on a triangular mesh ([NPDE, Section 3.3.1]) with a tent function basis ([NPDE, Section 3.3.3]). Compute the element (stiffness) matrix ([NPDE, Def. 3.6.35]) for a general triangle K analytically in terms of the angles, the area $|K|$, and edge lengths $|e_i|$.

HINT: Your computations for subproblems (7.4b) and (7.4c) could be useful.

Solution: In barycentric coordinates, the local shape functions are the barycentric coordinate functions λ_i, and from [NPDE, Lemma 3.6.61] we get

$$\mathsf{a}_K(\lambda_i, \lambda_j) = \begin{cases} \frac{c|K|}{12}, & i \neq j \\ \frac{c|K|}{6}, & i = j, \end{cases}$$

where $\mathsf{a}(\cdot,\cdot)$ is the bilinear form for the reaction part, i.e. $\mathsf{a}_K(u,v) = c\int_K u(\boldsymbol{x})v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}$.

So the local Galerkin matrix is (using [NPDE, Eq. (3.3.21)]).

$$\frac{1}{2}\begin{pmatrix} \cot\omega_3+\cot\omega_1 & -\cot\omega_3 & -\cot\omega_2 \\ -\cot\omega_3 & \cot\omega_3+\cot\omega_1 & -\cot\omega_1 \\ -\cot\omega_2 & -\cot\omega_1 & \cot\omega_2+\cot\omega_1 \end{pmatrix} + \frac{c|K|}{12}\begin{pmatrix} 2 & 1 & 1 \\ 1 & 2 & 1 \\ 1 & 1 & 2 \end{pmatrix}.$$

As you can see, the edge lengths weren't needed.

(8.1e) Write a class `LocalMass` templated with the type Function, whose constructor is

```
LocalMass(Function const& c) : c_(c) {};
```

and that provides the method

```
template <class Element, class Matrix>
void operator()(Element const& e, Matrix &local) const
```

to compute the element matrix associated to

$$\int_\Omega c(\boldsymbol{x})u(\boldsymbol{x})\,v(\boldsymbol{x})\,\mathrm{d}\boldsymbol{x}, \quad u,v\in L^2(\Omega)\ ,$$

and linear Lagrangian finite elements on triangular elements. This is similar to what you did for `AnalyticalLocalMass` in subproblem (7.4c), only now you must use

```
Dune::PkLocalFiniteElement<calc_t,calc_t, elem_dim, 1>
```

to obtain the local finite element basis (see [NPDE, Ex. 3.7.12] to recall their member functions), and `Dune::QuadratureRule<calc_t, elem_dim>` to integrate the entries, as shown in [NPDE, § 3.6.96]. Use a local quadrature rule of order 3, *cf.* [NPDE, Def. 3.6.87].

HINT: You can test your code by letting c be constant and comparing to the result from subproblem (8.1d)

Solution: See Listing 8.1 for the code.

Listing 8.1: Implementation for `LocalMass`

```
#ifndef LOCALMASS_HPP_
#define LOCALMASS_HPP_

#include <dune/localfunctions/lagrange/pk.hh>
#include <dune/geometry/quadraturerules.hh>
#include <dune/common/fmatrix.hh>

namespace NPDE15{

  template <class Function>
  class LocalMassC{
  public:
    using calc_t=double;
    using ElementMatrix = typename Dune::FieldMatrix<calc_t,3,3>;

```

```
    LocalMassC(Function const& c) : c_(c) {};

    template <class Element, class Matrix>
    void operator()(Element const& e, Matrix &locMassMat) const{
      const int world_dim = Element::dimension;
      const int elem_dim  = Element::mydimension;
      typedef typename Dune::QuadratureRule<calc_t, elem_dim> QuadRule_t;
      typedef typename Dune::QuadratureRules<calc_t, elem_dim> QuadRules;
      const QuadRule_t & quadRule = QuadRules::rule(e.type(), 3);
      Dune::PkLocalFiniteElement<calc_t,calc_t,elem_dim, 1> localFE;
      assert(localFE.type()==e.type());
      unsigned M = localFE.localBasis().size();
      locMassMat=0;
      auto const& egeom = e.geometry();

      for (auto qr : quadRule){
     // get quadrature point in the reference element
     auto const& qp_local_pos = qr.position();
     double  const& w = qr.weight();
     // evaluate shape function values (locally)
     std::vector<Dune::FieldVector<calc_t,1> > shapef_val;
     localFE.localBasis().evaluateFunction(qp_local_pos, shapef_val);
     // evaluate coefficient function (globally!)  at the current quadrature position
     calc_t coeff=c_(egeom.global(qp_local_pos));
     // determinant of transformation from reference element
     double jac_det = egeom.integrationElement(qp_local_pos);
     // add to local contribution matrix
     for (unsigned i=0;i<M;++i){
       for (unsigned j=0;j<M;++j)
         locMassMat[i][j]+= coeff*shapef_val[i]*shapef_val[j]*w*jac_det;
     }
      } // end for loop quadRule
    }
  private:
    Function const& c_;
  };

  // Template deduction helper for NPDE15::LocalMassC
  template <class Function>
  LocalMassC<Function> LocalMass(Function const& c){
    return LocalMassC<Function>(c);
  }

}
#endif
```

(8.1f) We aim to incorporate the treatment of Dirichlet boundary conditions into our finite element C++ code. For this reason, now you are required to implement a class `BoundaryDofs` which will take care to detect and mark the nodes corresponding to the Dirichlet boundary conditions. In order to achieve this purpose you have to complete the class

```
template <class DofHandler>
class BoundaryDofs{
  public:
   using IndexVector = std::vector<bool>;
   using calc_t=double;
   using GridView=typename DofHandler::GridView;
   enum{ K=1 };
   enum{ world_dim = GridView::dimension };

   BoundaryDofs(DofHandler const& dof_handler) :
      dofh(dof_handler), gv(dofh.gv){ };

   void operator()(IndexVector &BndVec) const;

  private:
    DofHandler const& dofh;
    GridView const& gv;
};
```

by writing the method `void operator()(IndexVector &BndVec) const`, so it detects the nodes at the boundary an initializes a component of `IndexVector BndVec` with `true`, if its index corresponds to a that of a global basis functions associated with a vertex on the boundary.

HINT: subproblems (7.3b) and (7.2b) may be helpful to recall how to work with the boundary. See [NPDE, Rem. 3.6.24] to recall the reference element conventions to obtain the local numbering of the intersection vertices.

Solution: See Listing 8.2 for the code.

Listing 8.2: Implementation for `BoundaryDofs`

```
#ifndef BOUNDARYDOFS_HPP_
#define BOUNDARYDOFS_HPP_

#include <vector>
#include <dune/common/fvector.hh>
#include <dune/geometry/referenceelements.hh>
#include <cassert>

namespace NPDE15{

  template <class DofHandler>
  class BoundaryDofs{
  public:
    using IndexVector = std::vector<bool>;
    using calc_t=double;
    using GridView=typename DofHandler::GridView;
```

```
    enum{ K=1 };
    enum{ world_dim = GridView::dimension }; // export dimension the grid lives in

    BoundaryDofs(DofHandler const& dof_handler) : dofh(dof_handler), gv(dofh.gv){ };

    void operator()(IndexVector &BndVec) const{
      BndVec = IndexVector(dofh.size(),false);
      // Loop over cells
      for (auto it=gv.template begin<0>();it!=gv.template end<0>();++it){
     auto const& e=*it;
     auto const& egeom = e.geometry();
     // Loop over intersections
     int is_counter = 0;
     for (auto iit=gv.ibegin(*it);iit!=gv.iend(*it);++iit){
       auto const& is=*iit;
       if (is.boundary()){ // check if it is in the boundary
         for (unsigned i=0;i<2;++i){
           // get vertex local id with respect to the entity
           unsigned local_i  = idVertexinElement(is_counter, i);
           unsigned global_i = dofh(e, local_i);
           BndVec[global_i] = true;
         }
       }  // end if boundary
       is_counter++;
     } // end for loop over intersections
      } // end for loop over cells

    }

    // Based on Dune's convention, use local number of vertix in the intersection,
    // to deduce it local number in the element
    int idVertexinElement(int idIntersection, int idVertexinIntersection)const{
      switch (idIntersection) {
      case 0: //edge 2
     switch(idVertexinIntersection){
     case 0: return 1;        break;
     case 1: return 2;        break;
     }
      case 1: //edge 1
     switch(idVertexinIntersection){
     case 0: return 0;        break;
     case 1: return 2;        break;
     }
      case 2: // edge 0
     switch(idVertexinIntersection){
     case 0: return 0;        break;
```

```
      case 1: return 1;          break;
      }
        default: { assert(false); }
        }
    }

    private:
      DofHandler const& dofh;
      GridView const& gv;
};

}

#endif
```

In [NPDE, Ex. 3.6.41], you learned that the `DofHandler` provides a method `active()` which tells whether a global shape function contributes to the finite element space. In subproblem (7.4a) you implemented this class with this method always returning `true`. Now we want to extend the class so it can handle inactive degrees of freedom. We do this by including the `FlagVector` `inactive_dofs` shown in the snippet below.

```
template <class GridView_t>
class DofHandler{
  public:
    using calc_t=double;
    using GridView=GridView_t;
    using index_t=typename GridView::IndexSet::IndexType;
    using FlagVector=std::vector<bool>;
    enum { K=1 };
    enum { world_dim=GridView::dimension };

    GridView const& gv;

    DofHandler(GridView const& gridview) :
    gv(gridview), set(gv.indexSet()), offset(set.size(world_dim)) {
      // all dofs are active by default
      inactive_dofs = FlagVector(size(), false); }

    void set_inactive(FlagVector const& inactive_dofs);

    bool active(index_t global_idx) const;

    template <class Element>
    index_t operator()(Element const& e, index_t dof) const;

    template <class Element>
    size_t size_loc(Element const &e) const;

    size_t size() const;
```

```
    GridView const& gv;

  private:
    typename GridView::IndexSet const& set;
    FlagVector inactive_dofs;
};
```

(8.1g) Update your code for `DofHandler` so it contains the new types and methods listed above. Then complete the method `active(index_t global_idx)` so it returns whether the dof corresponding to `global_idx` is active or not.

Solution: See Listing 8.3 for the code.

Listing 8.3: Implementation for `active()` in `DofHandler`

```
    bool active(index_t global_idx) const{
        return !inactive_dofs[global_idx];
    }
```

(8.1h) Implement the method

```
void set_inactive(FlagVector const& inactive_dofs)
```

to update the `FlagVector inactive_dofs`.

Solution: See Listing 8.4 for the code.

Listing 8.4: Implementation for `set_inactive()` in `DofHandler`

```
    void set_inactive(FlagVector const& inactive_dofs){
        assert(inactive_dofs.size()==size());
        this->inactive_dofs = inactive_dofs;
    }
```

(8.1i) Taking the cue from [NPDE, Ex. 3.6.113], complete your implementation of the class `VectorAssembler` by writing the method

```
template <class RHSVector , class Vector >
void set_inactive( RHSVector &Phi, const Vector & fixedVals) const
```

Solution:

(8.1j) Update the `main.cc` from subproblem (7.4f) to treat the homogeneous Dirichlet boundary conditions. Consider $f = 1$. This means it should

- Read a `.msh`-file and build the grid from it.
- Initialize the DofHandler.
- Create `lambda` functions for the Dirichet data and the right hand side,
- Use BoundaryDofs to mark the boundary nodes.
- Use the DofHandler to set them inactive.

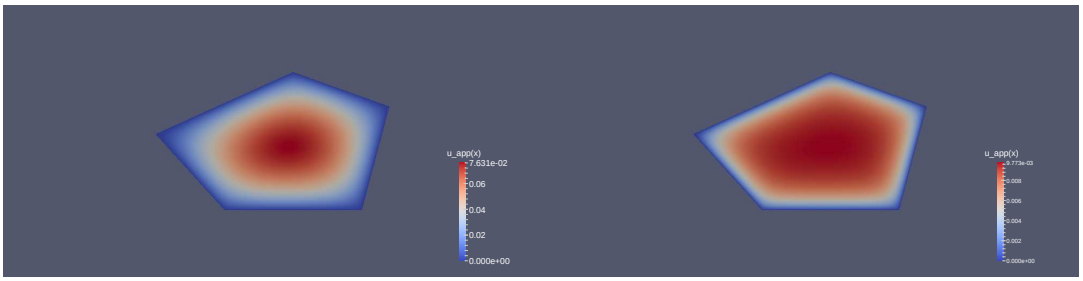
- Assemble the r.h.s. and use `set_inactive` for the values of the Dirichlet BC.
- Assemble the system matrix and use `set_inactive()` for the values of the Dirichlet BC.
- Solve and output the obtained solution to `Vtk` format.

Solution: See Listing 8.5 for the code.

Listing 8.5: Implementation for `main.cc`

```
const int world_dim = 2;
using calc_t = double;
using Matrix = Eigen::SparseMatrix<calc_t, Eigen::RowMajor>;
using Vector = Eigen::VectorXd;
using IndexVector = std::vector<bool>;
using GridType = Dune::ALUSimplexGrid<2, 2>;
using GridView = GridType::LeafGridView;
using Coordinate  = Dune::FieldVector<calc_t, world_dim>;
using DofHandler  = NPDE15::DofHandler<GridView>;

int main(int argc, char *argv[]){
  try{
    // load the grid from file
     std::string fileName;
    if(argc < 2){
      std::cout << "Enter msh file name: ";
      std::cin >> fileName;
    }
    else{
      fileName = argv[1];
    }

    // Declare and create mesh using the Gmsh file
    Dune::GridFactory<GridType> gridFactory;
    Dune::GmshReader<GridType>::read(gridFactory, fileName.c_str(),
       false, true);
    std::unique_ptr<GridType> workingGrid(gridFactory.createGrid());
    workingGrid->loadBalance();

    // Get the Gridview
    GridView gv = workingGrid->leafGridView();

    // Initialize dof-handler
    DofHandler dofh(gv);

    unsigned N = dofh.size();
    std::cout << "Solving for N =" << N << " unknowns.\n";

    // Load vector
    auto f = [](Coordinate const& x){return 1.0; };
```

```
    // Reaction function
    auto c = [](Coordinate const& x){return 1.0; };

    // Get boundary nodes
    IndexVector dirichlet_dofs(N);
    NPDE15::BoundaryDofs<DofHandler> get_bnd_dofs(dofh);
    get_bnd_dofs(dirichlet_dofs);
    dofh.set_inactive(dirichlet_dofs);

    ////////////////////////////////////////////////////////////
    // CREATE AND SOLVE SYSTEM USING ANALYTIC IMPLEMENTATION //
    ////////////////////////////////////////////////////////////
    // assemble rhs and set dirichlet dofs to dirichlet data
    Vector Phi(N); Phi.setZero();
    NPDE15::VectorAssembler<DofHandler> vecAssembler(dofh);
    vecAssembler(Phi, NPDE15::LLocalFunction(f));
    // Homogeneous dirichlet data
    Vector G(N); G.setZero();
    vecAssembler.set_inactive(Phi, G);

    // assemble the system matrix
    std::vector<Eigen::Triplet<calc_t>> triplets;
    NPDE15::MatrixAssembler<DofHandler> matAssembler(dofh);
    matAssembler(triplets, NPDE15::AnalyticalLocalLaplace());
    matAssembler(triplets, NPDE15::LocalMass(c));
    // Removes colums entries corresponding to inactive d.o.fs from
    // triplet container and sets corresponding diagonal entries to
       1.
    matAssembler.set_inactive(triplets);

    Matrix A(N, N);
    A.setFromTriplets(triplets.begin(), triplets.end());
    A.makeCompressed();

    // solution vector U
    Vector U(N); U.setZero();

    // solve the system
    U = Phi/A; // short-hand, see Pardiso.hpp for more information

    ////////////////////////////////////////////////////////////
    // WRITE SOLUTIONS TO VTK FILE (TO BE SEEN USING PARAVIEW) //
    ////////////////////////////////////////////////////////////
    std::cout << "\n\nWriting solution to vtk file... ";
    Dune::VTKWriter<GridView> vtkwriter(gv);
    std::stringstream outputName;
    outputName << "solution";
    vtkwriter.addVertexData(U, "u_app(x)");
    vtkwriter.write(outputName.str().c_str());
```



(a) $c(\boldsymbol{x}) = 1$ (b) $c(\boldsymbol{x}) = 100$

Figure 8.2: Plots for subproblem (8.1k).

```
    std::cout << "Done.\n";

  }
  // catch exceptions
  catch (Dune::Exception &e){
    std::cerr << "Dune reported error: " << e << std::endl;
  }
  catch (...)
    std::cerr << "Unknown exception thrown!" << std::endl;
  }
  return 0;
}
```

(8.1k) Solve and use Paraview to plot the solution of (8.1.1) for $c \equiv 1$ and $c \equiv 100$.

HINT: You may also use `Filters -> Warp By Scalar` to visualize it in 3D.

Solution:

Problem 8.2 Non-Linearly Coupled Elliptic Boundary Value Problems

This a complex problem related to a simulation task from engineering practice. It involves two linear second-order boundary value problems. Yet, their particular coupling renders the entire problem non-linear. Fortunately, the coupling is severed through a fixed point iteration that is employed as iterative solver so that we need solve only linear boundary problems. We accomplish this using linear Lagrangian finite elements. The new aspects coming into play are

1. mixed boundary conditions,
2. a coefficient that depends on a finite element solution,
3. Robin boundary conditions on parts of the domain boundary,
4. a right hand side source function that involves a finite element solution.

We consider the two coupled 2nd-order elliptic boundary value problems

$$\begin{aligned} -\operatorname{div}(\sigma(T(\mathbf{x}))\operatorname{grad} u(\mathbf{x})) &= 0 \quad \text{in } \Omega, \\ u(\mathbf{x}) &= 1 \quad \text{on } \Gamma_1, \\ u(\mathbf{x}) &= 0 \quad \text{on } \Gamma_0, \\ \operatorname{grad} u(\mathbf{x}) \cdot \mathbf{n}(\mathbf{x}) &= 0 \quad \text{on } \Gamma_N, \end{aligned} \tag{8.2.1}$$

and

$$\begin{aligned} -\operatorname{div}(\operatorname{grad} T(\mathbf{x})) &= \sigma(T)\|\operatorname{grad} u(\mathbf{x})\|^2 \quad \text{in } \Omega, \\ T(\mathbf{x}) &= 0 \quad \text{on } \Gamma_1 \cup \Gamma_0, \\ -\operatorname{grad} T(\mathbf{x}) \cdot \mathbf{n}(\mathbf{x}) &= T \quad \text{on } \Gamma_N. \end{aligned} \tag{8.2.2}$$

on the domain shown in Figure 8.3. Here $\Gamma_0 \cup \Gamma_1 \cup \Gamma_N$ is supposed to be a partition of the boundary. This means both (8.2.1) and (8.2.2) represent linear second-order elliptic boundary value problems with , see [NPDE, Ex. 2.7.8] and [NPDE, Ex. 2.5.18].

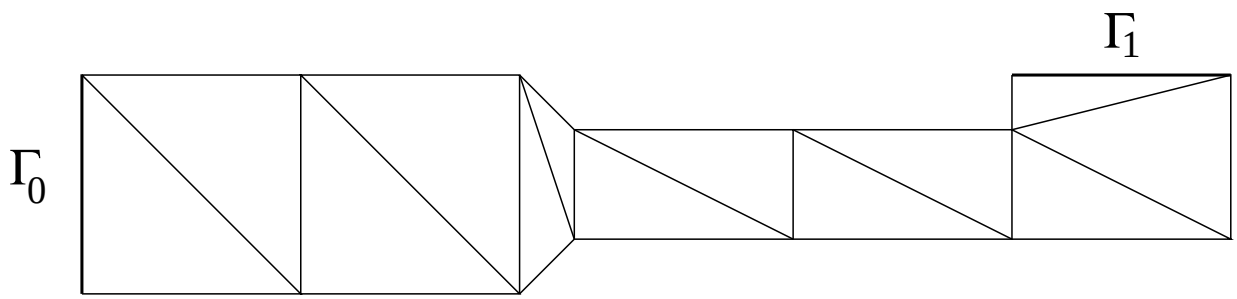


Figure 8.3: The domain Ω, with initial triangulation.

Concretely, Γ_1 is the horizontal edge in the top right of the domain, Γ_0 is the vertical edge to the left, and Γ_N is the remainder of the boundary. The function σ plotted in Figure 8.4 is defined as

$$\sigma(T) = 3 - \arctan(T).$$

The coupled BVPs (8.2.1) and (8.2.2) model the temperature $T = T(\boldsymbol{x})$ in a wire that carries an electric current. The current is driven by an electrostatic potential u, which is imposed at the contacts Γ_0 and Γ_1. The remaining boundary part Γ_N is electrically insulated so that there is no electric current flowing through it. The coefficient σ is to be read as electrical conductivity. The BVP (8.2.2) is the well known stationary heat equation, see [NPDE, Section 2.6] with prescribed temperature at the contacts and simple convective cooling boundary conditions at Γ_N, see [NPDE, Ex. 2.7.5]. The source term corresponds to the heat generated by the electric current through Ohmic losses.

We will use linear finite elements on a triangular mesh, see [NPDE, Section 3.3]. Template files for the new classes you will need to write are available in the lecture `svn` repository

`assignments_codes/assignment8/Problem2`

The idea is that you extend your own code, reason why it does not contain the files you already implemented in Problem 8.1 and 7.4. Please do not forget to include them when you submit your work.

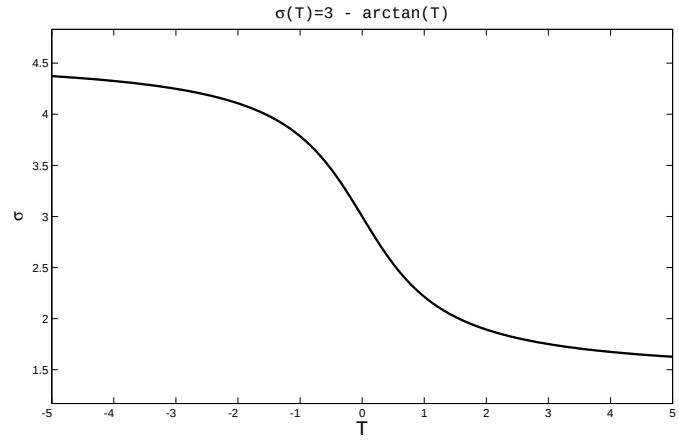


Figure 8.4: Variation of σ wrt T

(8.2a) Find the variational formulations of both (8.2.1) and (8.2.2).

HINT: Consider the T on the right-hand side of (8.2.2) to be fixed. Pay attention to the right function spaces matching the essential boundary conditions, see [NPDE, Section 2.9].

Solution: We denote our function spaces as

$$V = \left\{ u \in H^1(\Omega) \,\middle|\, u|_{\Gamma_1} = 1,\ u|_{\Gamma_0} = 0 \right\},$$
$$V_0 = \left\{ v \in H^1(\Omega) \,\middle|\, v|_{\Gamma_1 \cup \Gamma_0} = 0 \right\}.$$

Then the variational problem is to find $u \in V$ and $T \in V_0$ so that

$$\mathsf{a}_u(u, v) := \int_\Omega \sigma(T(\mathbf{x}))\, \mathrm{grad}\, u(\mathbf{x}) \cdot \mathrm{grad}\, v(\mathbf{x})\, \mathrm{d}\mathbf{x} = 0 =: \ell_u(v),$$
$$\mathsf{a}_T(T, R) := \int_\Omega \mathrm{grad}\, T(\mathbf{x}) \cdot \mathrm{grad}\, R(\mathbf{x})\, \mathrm{d}\mathbf{x} + \int_{\Gamma_N} R(\mathbf{x}) T(\mathbf{x})\, \mathrm{d}\mathbf{x} =$$
$$\int_\Omega \sigma(T(\mathbf{x})) \|\mathrm{grad}\, u(\mathbf{x})\|^2 R(\mathbf{x})\, \mathrm{d}\mathbf{x} =: \ell_T(R),$$

for all $v, R \in V_0$.

(8.2b) First we focus on (8.2.1). Observe we have Dirichlet Boundary Conditions in just one part of the boundary. We begin by extending the class `BoundaryDofs` so it mark the boundary dofs in which we are interested.

Implement the method

```
template <class Predicate>
    void operator()(IndexVector &BndVec, Predicate const& Pred)
        const
```

which detects the degrees of freedom in the boundary and fills their corresponding index `BndVec` with `true` if Pred is `true` for them.

HINT: The Predicate will provide `bool operator()(Coordinate x)`.

Solution: See Listing 8.6 for the code.

Listing 8.6: Implementation of constructor for `operator()`

```
template <class Predicate>
void operator()(IndexVector &BndVec, Predicate const& Pred) const{
  BndVec = IndexVector(dofh.size(),false);
  // Loop over cells
  for (auto it=gv.template begin<0>();it!=gv.template end<0>();++it){
    auto const& e=*it;
    auto const& egeom = e.geometry();
    // Loop over intersections
    for (auto iit=gv.ibegin(*it);iit!=gv.iend(*it);++iit){
      auto const& is=*iit;
      if (is.boundary()){ // check if it is in the boundary
     // reference element of entity
     auto &ref_elem = Dune::ReferenceElements<calc_t,world_dim>::general(e.type());
     // get number of vertices on intersection
     unsigned num_vertices = ref_elem.size(is.indexInInside(),1,world_dim);
     for (unsigned i=0;i<num_vertices;++i){
       // get vertex id with respect to the entity (instead of w.r.t. intersection)
       unsigned local_i = ref_elem.subEntity(is.indexInInside(), 1, i, world_dim);
       auto const& position = egeom.corner(local_i);
       if (Pred(position)){
         // set global index of i-th node of entity e to true if Pred returned true
         unsigned global_i=dofh(e, local_i);
         BndVec[global_i]=true;
       }
     }
      }

    } // end for loop over intersections
  } // end for loop over cells
}
```

(8.2c) Use

`Dune::PkLocalFiniteElement<calc_t,calc_t, elem_dim, 1>`

and

`Dune::QuadratureRule<calc_t, elem_dim>`

to complete the implementation of the class `LocalLaplacefromVector`

```
template <class Vector, class DofHandler, class Function>
  class LocalLaplacefV{
  public:
    using calc_t=double;
     using ElementMatrix = typename Dune::FieldMatrix<calc_t,3,3>;

    LocalLaplacefV(Vector const& T, DofHandler const& dofh,
       Function const& q) : T_(T), dofh_(dofh), q_(q) {};

    template <class Element>
    void operator()(Element const& e, ElementMatrix &local) const;
private:
    Vector const& T_;
    Function const& q_;
   DofHandler const& dofh_;
  };
```

by implementing the method `void operator()` to compute the element matrix associated to

$$\int_\Omega q(\boldsymbol{x})\,\mathbf{grad}\,u\cdot\mathbf{grad}\,v\,\mathrm{d}\boldsymbol{x},\quad u,v\in H^1(\Omega),$$

and linear Lagrangian finite elements, where $q(x)$ is an arbitrary function depending on a finite element function $T(x)=\sum_i T_i b_i(x)$, and `T` is its corresponding coefficient vector.

HINT: The class `LocalFunctionfromVector` contains builds the local vector for a right hand side of the form $\int_\Omega \alpha(x)v(\mathbf{x})\,\mathrm{d}\mathbf{x}$, where $\alpha(x)$ is a function depending on two finite element functions $T(x)=\sum_i T_i b_i(x)$ and $U(x)=\sum_i U_i\,\mathbf{grad}\,b_i(x)$. It may help you to fullfill this task.

Solution: See Listing 8.7 for the code.

Listing 8.7: Implementation for `LocalLaplacefromVector`

```
#ifndef LOCALLAPLACEFROMVECTOR_HPP_
#define LOCALLAPLACEFROMVECTOR_HPP_

#include <dune/localfunctions/lagrange/pk.hh>
#include <dune/geometry/quadraturerules.hh>
#include <dune/common/fmatrix.hh>

namespace NPDE15{

  template <class Vector, class DofHandler, class Function>
  class LocalLaplacefV{
  public:
    using calc_t=double;
     using ElementMatrix = typename Dune::FieldMatrix<calc_t,3,3>;

    LocalLaplacefV(Vector const& T, DofHandler const& dofh,
       Function const& q) : T_(T), dofh_(dofh), q_(q) {};

```

```
    template <class Element>
    void operator()(Element const& e, ElementMatrix &local) const{
      const int world_dim = Element::dimension;
      const int elem_dim  = Element::mydimension;
      typedef typename Dune::QuadratureRule<calc_t, elem_dim> QuadRule_t;
      typedef typename Dune::QuadratureRules<calc_t, elem_dim> QuadRules;
      const QuadRule_t & quadRule = QuadRules::rule(e.type(), 2);
      Dune::PkLocalFiniteElement<calc_t,calc_t,elem_dim, 1> localFE;
      local=0;

      auto const& egeom = e.geometry();
      for (auto qr : quadRule){
     auto const& local_pos=qr.position();
     // jacobian inverse transposed for transformation rule
     auto &jacInvTransp = egeom.jacobianInverseTransposed(local_pos);
     std::vector<Dune::FieldMatrix<calc_t,1,world_dim>> ref_gradients;
     // gradients on reference element evaluated at the quad points
     localFE.localBasis().evaluateJacobian(local_pos, ref_gradients);

     std::vector<Dune::FieldVector<calc_t,world_dim> > gradients(ref_gradients.size());
     // transform reference gradients to real element gradients:
     for (unsigned i=0;i<gradients.size();++i)
       jacInvTransp.mv(ref_gradients[i][0], gradients[i]);
     // determinant of transformation from reference element
     double jac_det = egeom.integrationElement(local_pos);
     // evaluate coefficient
     Dune::FieldVector<calc_t,3> T_locVal;
     for(int i=0; i<3; ++i) T_locVal[i] = T_[dofh_(e,i)];
     // evaluate shape function values
     std::vector<Dune::FieldVector<calc_t,1> > shapef_vals;
     localFE.localBasis().evaluateFunction(local_pos, shapef_vals);
     double coeff = q_(egeom.global(local_pos), shapef_vals, T_locVal);
     // add local contributions
     for (unsigned i=0;i<local.N();++i){
       for (unsigned j=0;j<local.M();++j){
         local[i][j]+=coeff*(gradients[i]*gradients[j])*qr.weight()*jac_det;
       }
     }

      }
    }
  private:
    Vector const& T_;
```

```
    Function const& q_;
    DofHandler const& dofh_;
  };

 template <class Vector, class DofHandler, class Function>
 LocalLaplacefV<Vector, DofHandler, Function>
    LocalLaplacefromVector(Vector const& T, DofHandler const& dofh,
    Function const& q){
   return LocalLaplacefV<Vector, DofHandler, Function>(T, dofh, q);
  }

}
#endif
```

(8.2d) Inside `solvers.hpp`, create a class `sigmaT` providing the following method

```
template<class Coordinate, class MatrixLocalBasis, class Vector>
double operator ()(Coordinate loc_x, MatrixLocalBasis lb_locVal,
   Vector T_locVal) const;
```

Here, `loc_x` will be a 2×1 vector containing the coordinates of a point *in the unit triangle*, `T_locVal` is a vector containing a linear finite element approximation of T in the usual form (nodal values), and `lb_locVal` are the local basis functions evaluated at `loc_x`. It should return the function values $\sigma \circ T$ evaluated at x.

Solution: See Listing 8.8 for the code.

Listing 8.8: Implementation for `sigmaT`

```
class SigmaT {
  public:
  SigmaT(void){}
  template<class Coordinate, class MatrixLocalBasis, class Vector>
    double operator ()(Coordinate loc_x, MatrixLocalBasis
       lb_locVal, Vector T_locVal) const{
    std::vector<Dune::FieldVector<double,1> >
       sigmaT_locCoeff(lb_locVal.size());
    // transform reference gradients to real element gradients:
    double T_lVal = 0.0;
    for(int k=0; k<3; ++k) T_lVal += T_locVal[k]*lb_locVal[k];
    return 3 - atan(T_lVal) ;
    }
};
```

(8.2e) In the file `solvers.hpp`, the following class is defined

```
template <class DofHandler>
class Solver_1
```

Implement the constructor

```
template <class DofHandler>
template <class DirichletLocator>
Solver_1<DofHandler>::Solver_1(DofHandler const& dof_handler,
    DirichletLocator const& location_dir)
```

so it uses the given arguments to set the boundary conditions. The inputs are the `dof_handler` as usual and `DirichletLocator location_dir`, which is a boolean function returning whether a point x belongs to the Dirichlet boundary or not.

Solution: See Listing 8.9 for the code.

Listing 8.9: Implementation of constructor for `solver_1`

```
template <class DofHandler>
template <class DirichletLocator>
Solver_1<DofHandler>::Solver_1(DofHandler const& dof_handler,
                               DirichletLocator const& location_dir)
  : dofh(dof_handler), gv(dofh.gv), N(dofh.size()) {
  NPDE15::LBoundaryNodes<DofHandler> get_bnd_dofs(dofh);
  // obtain index vectors corresponding to dirichlet dofs
  get_bnd_dofs(dirichlet_dofs, location_dir);
  dofh.set_inactive(dirichlet_dofs);
}
```

(8.2f) Implement the method

```
template <class DofHandler>
void Solver_1<DofHandler>::operator()(Vector const& T, Vector &U)
```

so it takes any `Vector T` and provides fills the `Vector U` with a linear finite element approximation of the solution (8.2.1). This means it should assemble the Global Matrix `A` corresponding to the bilinear form of (8.2.1) and the right hand side, and then solve the system.

HINT: The `main.cc` from subproblem (8.2a) and 7.4 may be useful.

Solution: See Listing 8.10 for the code.

Listing 8.10: Implementation of `operator()` for `solver_1`

```
template <class DofHandler>
void Solver_1<DofHandler>::operator()(Vector const& T, Vector &U){
  // assemble the system matrix
  std::vector<Eigen::Triplet<calc_t>> triplets;
  NPDE15::MatrixAssembler<DofHandler> matAssembler(dofh);
  SigmaT sigmaT;
  matAssembler(triplets, NPDE15::LocalLaplacefromVector(T, dofh,
      sigmaT));
  // Removes colums entries corresponding to inactive d.o.fs from
  // triplet container and sets corresponding diagonal entries to 1.
  matAssembler.set_inactive(triplets);

```

```
  A = MatrixType(N,N);
  A.setFromTriplets(triplets.begin(), triplets.end());
  A.makeCompressed();

  ///////////////
  // RHS //
  ///////////////
  // Load vector
  auto f = [](Coordinate const& x){
    return 0.0;
  };

  // Dirichlet data
  auto g = [](Coordinate const& x){
    if(x[0] == 0) return 0.0;
    else if(x[1] == 1 && x[0] >= 4) return 1.0;
  };
  // loop over cells
  Vector G(N); G.setZero();
  for (auto it=gv.template begin<0>();it!=gv.template
      end<0>();++it){
    auto const& e=*it;
    auto egeom = e.geometry();
    for (unsigned i=0;i<3;++i){
      unsigned loctoglob=dofh(e, i);
      G[loctoglob] = g(egeom.corner(i));
    }
  }

  // initialize vector assembler (for load vector)
  NPDE15::VectorAssembler<DofHandler> vecAssembler(dofh);

 // assemble load vector (with Phi = g at dirichlet dofs)
  Vector Phi(N); Phi.setZero();
  vecAssembler(Phi, NPDE15::LLocalFunction(f));
  vecAssembler.set_inactive(Phi, G);

  // solution vector u
  U = Vector(N); U.setZero();

  try{
    U = Phi/A; // solve the system with Pardiso
  }
  catch(...)
    std::cout << "\nError while solving the system of equations!\n";
  }

}
```

Now we focus on (8.2.2). Since we face a Robin boundary condition as in [NPDE, Ex. 2.7.5], we

need to extend our code to deal with another term in the bilinear form.

(8.2g) Complete the class `MatrixAssembler` by creating the method

```
template <class Triplets, class LocalAssembler, class IndexVector>
    void operator()(Triplets &triplets, LocalAssembler const&
        local_assembler, IndexVector const& bnd) const;
```

which calls the local assembler for each boundary intersection that belongs to bnd and distributes the local contributions to the global matrix the triplets.

Solution: See Listing 8.11 for the code.

Listing 8.11: Implementation of boundary `operator()`

```
    template <class Triplets, class LocalAssembler, class
        IndexVector>
    void operator()(Triplets &triplets, LocalAssembler const&
        local_assembler, IndexVector const& bnd) const{
      // loop over cells
      for (auto it=gv.template begin<0>();it!=gv.template
          end<0>();++it){
     auto const& e=*it;
     // loop over intersections
     for (auto iit=gv.ibegin(*it);iit!=gv.iend(*it);++iit){
       auto const& is=*iit;
       if (is.boundary()){
         typename LocalAssembler::ElementMatrix elementmatrix;
         // get local element matrix (integrate over intersection)
         local_assembler(is, elementmatrix);

         // distribute local matrix to global matrix using the dof
             handler
         for (unsigned i=0;i<2;++i){
           unsigned i_index=dofh(is, e, i);
           for (unsigned j=0;j<2;++j){
          unsigned j_index=dofh(is, e, j);
          // only add if indeces are part of the boundary being
              integrated
          if (bnd[i_index] && bnd[j_index])
            triplets.push_back({i_index,j_index,elementmatrix[i][j]});
           }
         }
       }
     }
      }
    }
```

(8.2h) Inside `solvers.hpp`, you will find the class `RHSf`, complete it by implementing the following method

```
template<class Coordinate, class MatrixGradBasis, class Vector>
```

```
double gradU(Coordinate loc_x, MatrixGradBasis grad_locVal,
    Vector U_locVal)const;
```

Where, `loc_x` will be a 2×1 vector containing the coordinates of a point *in the unit triangle*, `U_locVal` is a vector containing a linear finite element approximation of U in the usual form (nodal values), and `grad_locVal` are the local basis gradients evaluated at `loc_x`. The returned values should be the function $\|\text{grad}\, u(\mathbf{x})\|^2$ evaluated at x

Solution: See Listing 8.12 for the code.

Listing 8.12: Implementation of `gradU`

```
class RHSf {
  public:
    RHSf(void) {}
  template<class Coordinate, class MatrixLocalBasis, class
      MatrixGrad, class Vector>
    double operator ()(Coordinate loc_x, MatrixLocalBasis lb_locVal,
                       MatrixGrad lb_locGrad, Vector U_locVal, Vector
                          T_locVal) const{
      return sigmaT(loc_x, lb_locVal, T_locVal)* gradU(loc_x,
          lb_locGrad, U_locVal);
    }

    template<class Coordinate, class MatrixLocalBasis, class Vector>
    double gradU(Coordinate loc_x, MatrixLocalBasis lb_locVal,
        Vector U_locVal)const{
      double gradU_locVal2 = 0.0;
      Dune::FieldVector<double,2> gradU_locVal; gradU_locVal = 0.0;
      for(int k=0; k<3; ++k) {
     lb_locVal[k] *= U_locVal[k];
        gradU_locVal += lb_locVal[k];
      }
      return gradU_locVal*gradU_locVal;
    }
 private:
 SigmaT sigmaT;
};
```

(8.2i) In the file `solvers.hpp`, the following class is defined

```
template <class DofHandler>
class Solver_2
```

Implement the constructor

```
template <class DofHandler>
template <class DirichletLocator, class NeumannLocator>
Solver_2<DofHandler>::Solver_2(DofHandler const& dof_handler,
    DirichletLocator const& location_dir,
```

```
NeumannLocator
    const&
    location_n)
```

so it uses the given arguments to build the Global Matrix `A` corresponding to the bilinear form of (8.2.2). Where the first inputs are the same as for `Solver_1` and `NeumannLocator` `location_n` is a boolean function returning whether a point x belongs to the Neumann boundary or not.

HINT: Notice that the Global Matrix does not depend on U and T, therefore we build it just once. When assembling `A`, take into account that you will additionally incorporate the term corresponding to the Robin B.C.

Solution: See Listing 8.13 for the code.

Listing 8.13: Implementation of constructor for `solver_2`

```
template <class DofHandler>
template <class DirichletLocator, class NeumannLocator>
Solver_2<DofHandler>::Solver_2(DofHandler const& dof_handler,
    DirichletLocator const& location_dir,
                        NeumannLocator const& location_n)
:  dofh(dof_handler), gv(dofh.gv), N(dofh.size()) {
  NPDE15::LBoundaryNodes<DofHandler> get_bnd_dofs(dofh); //
      boundary-dofs object

  // obtain index vectors corresponding to dirichlet/neumann dofs
  get_bnd_dofs(dirichlet_dofs, location_dir);
  dofh.set_inactive(dirichlet_dofs);
  get_bnd_dofs(neumann_dofs, location_n);
}
```

(8.2j) Implement the method

```
template <class DofHandler>
void Solver_2<DofHandler>::operator()(Vector const& U, Vector &T)
```

so it takes any `Vector U` and provides fills the `Vector T` with a linear finite element approximation of the solution (8.2.2). This means it should assemble the right hand side and solve the system. HINT: The load vector function f for (8.2.2) should be the product of `sigmaT` and `gradU`. This is already defined in `RHSf`, which you can pass to `LocalFunction`.

Solution: See Listing 8.14 for the code.

Listing 8.14: Implementation of `operator()` for `solver_2`

```
template <class DofHandler>
void Solver_2<DofHandler>::operator()(Vector const&U, Vector &T){
  // build stiffness matrix
  std::vector<Eigen::Triplet<calc_t>> triplets;
  NPDE15::MatrixAssembler<DofHandler> matAssembler(dofh);
```

```
  matAssembler(triplets, NPDE15::AnalyticalLocalLaplace()); // assemble normal stiffness matrix
  matAssembler(triplets, NPDE15::LocalMass([](Coordinate const&){return 1.0;}), neumann_dofs);
  matAssembler.set_inactive(triplets);

  A = MatrixType(N,N);
  A.setFromTriplets(triplets.begin(), triplets.end());
  A.makeCompressed();

  // initialize vector assembler (for load vector)
  NPDE15::VectorAssembler<DofHandler> vecAssembler(dofh);
  Vector Phi = Vector(N); Phi.setZero();
  RHSf rhsf;
  vecAssembler(Phi, NPDE15::LocalFunctionfromVector(T, U, dofh, rhsf));
  Vector G(N); G.setZero();
  vecAssembler.set_inactive(Phi, G);

  try{
    T = Phi/A; // solve the system with Pardiso using stiffness matrix with identity entries at dirichlet nodes
  }
  catch(...)
    std::cout << "\nError while solving the system of equations!\n";
  }

}
```

(8.2k) Inside `solvers.hpp` implement the method

```
template <class GridView, class Function>
double H1snError(GridView gv, Vector const& Q ) const;
```

which returns the $H^1(\Omega)$ seminorm of a finite element function $q(x)$ with coefficient vector `Q`. HINT: [NPDE, Code 3.6.97] may be of help.

Solution: See Listing 8.15 for the code.

Listing 8.15: Implementation of `H1snError`

```
template <class DofHandler, class Vector>
double H1sNorm(DofHandler const& dofh, Vector const& Q) {
  std::vector<Eigen::Triplet<double>> triplets;
  NPDE15::MatrixAssembler<DofHandler> matAssembler(dofh);
  matAssembler(triplets, NPDE15::AnalyticalLocalLaplace());

  Eigen::SparseMatrix<double,Eigen::RowMajor> A(dofh.size(),dofh.size());
  A.setFromTriplets(triplets.begin(), triplets.end());
  A.makeCompressed();

  return sqrt(Q.transpose()*A*Q);
```

```
}

/* alternative
template <class DofHandler, class Vector>
double H1sNorm(DofHandler const& dofh, Vector const& Q) {
  using GridView = typename DofHandler::GridView;
  GridView const& gv = dofh.gv;
  using calc_t = double;
  const int world_dim = 2;
  Dune::PkLocalFiniteElement<calc_t,calc_t,world_dim,1> localFE;
  typedef typename Dune::QuadratureRule<calc_t, world_dim>
     QuadRule_t;
  typedef typename Dune::QuadratureRules<calc_t, world_dim>
     QuadRules;
  const QuadRule_t & quadRule = QuadRules::rule(localFE.type(), 10);

  calc_t H1sn=0.;
  for (auto it=gv.template begin<0>();it!=gv.template
     end<0>();++it){
    auto const& e=*it;
    auto egeom = e.geometry();
    assert(localFE.type()==e.type());
    for (auto qr : quadRule){
      auto const& local_pos=qr.position();
      // jacobian inverse transposed for transformation rule
      auto &jacInvTransp =
         egeom.jacobianInverseTransposed(local_pos);
      std::vector<Dune::FieldMatrix<calc_t,1,world_dim>>
         ref_gradients;
      // gradients on reference element evaluated at the quad points
      localFE.localBasis().evaluateJacobian(local_pos,
         ref_gradients);

      std::vector<Dune::FieldVector<calc_t,world_dim> >
         gradients(ref_gradients.size());
      // transform reference gradients to real element gradients:
      for (unsigned i=0;i<gradients.size();++i)
     jacInvTransp.mv(ref_gradients[i][0], gradients[i]);
      // determinant of transformation from reference element
      double jac_det = egeom.integrationElement(local_pos);

      Dune::FieldVector<calc_t,world_dim> valueQ;
      valueQ=0.0; // calculate interpolation at current quadrature
         point
      for (unsigned i=0;i<gradients.size();++i){
     unsigned globalidx=dofh(e, i);
     gradients[i] *= Q[globalidx];
     valueQ += gradients[i];
      }

```

```
    // and add weighted difference to the l2 error
    H1sn+=(valueQ*valueQ)*qr.weight()*jac_det;
    }
  }
  return sqrt(H1sn);
}
*/
```

(8.2l) The file `main.cc` contains code for loading the mesh and setting up the boundary conditions. We will now extend this code so that it solves (8.2.1) and (8.2.2) using an algorithm known as nonlinear Gauss-Seidel (see Algorithm 8.1).

Algorithm 8.1 Nonlinear Gauss-Seidel for (8.2.1) and (8.2.2)

Require: Termination condition ϵ.

$\mathbf{T}^{(0)} = 0$
for $i = 0, 1, \ldots$ **do**
  $\mathbf{U}^{(i)} = \texttt{solver\_1}(\mathbf{T}^{(i)})$
  $\mathbf{T}^{(i+1)} = \texttt{solver\_2}(\mathbf{U}^{(i)}, \mathbf{T}^{(i)})$
  if $\|\mathrm{grad}(\mathbf{T}^{(i+1)} - \mathbf{T}^{(i)})\| / \|\mathrm{grad}(\mathbf{T}^{(i+1)})\| \leq \epsilon$ **then**
    break
  end if
end for
return $\mathbf{U}^{(i)}, \mathbf{T}^{(i+1)}$

Implement this algorithm, and run `main.cc` with $\epsilon = 10^{-3}$. Plot the solutions for T and u. Reference solutions are given in Figure 8.5.

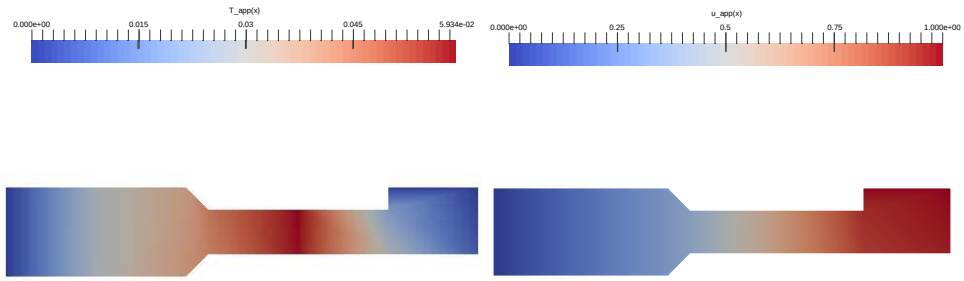
HINT: For the termination condition use the Galerkin matrix to compute the L^2-norm of the gradient of a finite element solution.

Solution: See Listing 8.16 for the code.

Listing 8.16: Implementation of `main.cc`

```
const int world_dim = 2;

using calc_t = double;
using IndexVector = std::vector<bool>;
using Matrix = Eigen::SparseMatrix<calc_t, Eigen::RowMajor>;
using Vector = Eigen::VectorXd;
using GridType = Dune::ALUSimplexGrid<2, 2>;
using GridView = GridType::LeafGridView;
using Coordinate  = Dune::FieldVector<calc_t, world_dim>;
using DofHandler  = NPDE15::LDofHandler<GridView>;
using Coords_t = Eigen::Matrix<calc_t, 3,2>;

int main(int argc, char *argv[]){
  try{
    // load the grid from file
    std::string fileName = "pipe.msh";
```

```
    // Declare and create mesh using the Gmsh file
    Dune::GridFactory<GridType> gridFactory;
    Dune::GmshReader<GridType>::read(gridFactory, fileName.c_str(),
        false, true);
    std::unique_ptr<GridType> workingGrid(gridFactory.createGrid());
    workingGrid->loadBalance();

    // Get the Gridview
    GridView gv = workingGrid->leafGridView();

    // Initialize dof-handler
    DofHandler dofh(gv);

    unsigned N = dofh.size();
    std::cout << "Solving for N =" << N << " unknowns.\n";

    // Function returning if x belong to dirichlet boundary or not
    auto loc_dirichlet = [](Coordinate const& x){
      return ((x[0]==0)+(x[0]>=4&&x[1]==1)); };

    // Declare solvers (passing loc dirichlet and the same for
        neumann)
    Solver_1<DofHandler> solver_1(dofh, loc_dirichlet);
    Solver_2<DofHandler> solver_2(dofh, loc_dirichlet,
        [](Coordinate const& x){return !((x[0]==0) +
            (x[0]>=4&&x[1]==1)); });

    // NON-LINEAR GAUSS SEIDEL
    double eps_ = 0.001; int Maxiter = 10;
    Vector U_app(N);
    Vector T_app(N); T_app.setZero();
    Vector T_old(N); T_old.setZero();
    int iter = 0;
    while(true){
      T_old = T_app;
      solver_1(T_app,U_app);
      solver_2(U_app, T_app);
      // Stopping criteria
       if (H1sNorm(dofh, T_app-T_old)/H1sNorm(dofh, T_app) <= eps_){
      std::cout << "stopped at iter " << iter << std::endl; break;}
       if (iter == Maxiter){
      std::cout << "max iter reached " << std::endl; break;
       }
       iter++ ;
    }

    ////////////////////////////////////////////////////////////
    // WRITE SOLUTIONS TO VTK FILE (TO BE SEEN USING PARAVIEW) //
    ////////////////////////////////////////////////////////////
```



(a) T (b) u

Figure 8.5: Plots for subproblem (8.2l).

```
    std::cout << "\n\nWriting solution to vtk file... ";
    Dune::VTKWriter<GridView> vtkwriter(gv);
    std::stringstream outputName;
    outputName << "solution";
    vtkwriter.addVertexData(U_app, "u_app(x)");
    vtkwriter.addVertexData(T_app, "T_app(x)");
    vtkwriter.write(outputName.str().c_str());
    std::cout << "Done.\n";

  }
  // catch exceptions
  catch (Dune::Exception &e){
    std::cerr << "Dune reported error: " << e << std::endl;
  }
  catch (...)
    std::cerr << "Unknown exception thrown!" << std::endl;
  }
  return 0;
}
```

Published on 15.04.2015.

To be submitted on 22.04.2015.

References

[NPDE] Lecture Slides for the course "Numerical Methods for Partial Differential Equations".SVN revision # 79326.

[NCSE] Lecture Slides for the course “Numerical Methods for CSE”.

Last modified on July 31, 2015