R. Hiptmair
L. Scarabosio
C. Urzua Torres

Spring Term 2015

Numerical Methods for Partial Differential Equations

ETH Zürich
D-MATH

Homework Problem Sheet 7

Introduction. This problem sheet is your first encounter with finite element implementation in C++ using a DUNE-style interface. It covers many aspects addressed in [NPDE, Section 3.6] and [NPDE, Section 3.6.3]. Doing the problems is essential for become familiar with DUNE.

Problem 7.1 Generating and Writing Mesh Data

This problem addresses how to generate a mesh with Gmsh, use DUNE to construct a Grid from a `.msh`-file and output it in `Vtk` format, so it can, for instance, be visualized using Paraview. As a preparation please study the material of [NPDE, Section 3.6.1].

(7.1a) First we start by generating a mesh with Gmsh. In order to achieve this task, you are required to:

- Build a rectangle with vertices: $\{(0,0), (2,0), (2,1), (0,1)\}$ using `Modules -> Geometry`.
- Generate a mesh with `Modules -> Mesh -> 2D`.
- Refine your mesh using `Modules -> Mesh -> Refine by splitting`. Do this a couple of times until you get around 32 elements.

  HINT: You can use `Tools -> Statistics -> Mesh` to see the number of elements in the current mesh.
- Finally save your mesh as a `rect.msh`.

Solution: Available in the `svn` repository as `rect2x1_32.msh`

(7.1b) Follow the steps stated in subproblem (7.1a), now to build a unit circle centered at (0,0). Refine until you get around 64 elements and save it as `circle.msh`.

Solution: Available in the `svn` repository as `circle_64.msh`

(7.1c) Write a `C++` code which reads a `.msh`-file file and uses the obtained data to initialize a triangular mesh. Remember that in this course we use `Dune::ALUSimplexGrid<2,2>` as grid implementation.

HINT: [NPDE, Rem. 3.6.11] might be of use.

Solution: See Listing 7.1 for the code.

Listing 7.1: Implementation for `main`

```
#include <stdexcept>
#include <cassert>
#include <cstdlib>
// Dune headers
#include "../config.h"
#include <dune/common/exceptions.hh>
#include <dune/grid/alugrid.hh>
#include <dune/grid/alugrid/2d/alugrid.hh>
#include <dune/grid/common/gridfactory.hh>
#include <dune/grid/io/file/gmshreader.hh>
#include <dune/grid/io/file/vtk/subsamplingvtkwriter.hh>

using namespace std;

int main(int argc, char *argv[]){
  try{
    const int world_dim=2;
    ////////////////////////////////////////////////////////////
    // READ TRIANGULAR MESH FROM GMSH FILE //
    ////////////////////////////////////////////////////////////
    // Get mesh file name
    const string FileName = argv[1];
    // Declare and create mesh using the Gmsh file
    using GridType = Dune::ALUSimplexGrid<2,2>;
    Dune::GridFactory<GridType> gridFactory;
    Dune::GmshReader<GridType>::read(gridFactory, FileName.c_str(),
        false, true);
    cout << "creating Grid from " << FileName << endl;
    GridType *workingGrid = gridFactory.createGrid();
    workingGrid->loadBalance();
    // Get the Gridview
    using GridView = GridType::LeafGridView;
    GridView gv    = workingGrid->leafGridView();

    ////////////////////////////////////////////////////////////
    // WRITE GRID TO VTK FILE (TO BE SEEN USING PARAVIEW) //
    ////////////////////////////////////////////////////////////
    cout << "Writing grid to vtk file... ";
    Dune::VTKWriter<GridView> vtkwriter(gv);
    stringstream name;
    name << "dune_grid";
    vtkwriter.write(name.str().c_str());
    cout << "Done." << endl;

  }
  // catch exceptions
  catch (Dune::Exception &e){
    cerr << "Dune reported error: " << e << endl;
  }
```

```
  catch (...)
    cerr << "Unknown exception thrown!" << endl;
  }
  return 0;
}
```

(7.1d) Write a C++ code which writes the Grid generated with Dune into a `.vtu`-file.

HINT: Use `Dune::VTKWriter<GridView>vtkwriter(const GridView & gv)`.

Solution: See Listing 7.1 for the code.

(7.1e) Write a `main` file to test your code from subproblem (7.1c)–(7.1d). Then run it using the meshes you generated in subproblem (7.1a)–(7.1b). Finally open the output files using Paraview and store the plots as PDF files.

HINT: You may as well write the last 3 subtasks together. A template file `main.cc` is available in the lecture `svn` repository in the folder

`assignments_codes/assignment7/Problem1`

Problem 7.2 Area and Perimeter Using DUNE (Core problem)

In this problem we practice how to import a mesh from a `.msh`-file and explore DUNE mesh data structures covered in [NPDE, Section 3.6.2]. Particularly, using the notation introduced in the lecture material, we deal with the following DUNE data types

- `GridView::template Codim<k>::Iterator` (→ [NPDE, Code 3.6.17])
- `GridView::template Codim<k>::Entity` (→ [NPDE, Code 3.6.23], Link)
- `GridView::template Codim<k>::EntityPointer` (→ [NPDE, Code 3.6.23], Link)
- `GridView::Intersection` (→ [NPDE, Code 3.6.30], Link)
- `GridView::IntersectionIterator` (→ [NPDE, Code 3.6.30])
- `Entity::Geometry` (→ [NPDE, Code 3.6.27], Link)

Please study the related explanations given in the course material unless you remember the details.

(7.2a) Write a (templated, DUNE-based) C++-function

```
double getDomainArea(GridView const& gv)
```

which takes as input the `GridView gv`, uses it to iterate over the grid elements, adds each element's area to the domain's area, and returns the total value.

HINT: Remember that in DUNE elements are entities of co-dimension 0. Thus, we use

```
GridView::template Codim<0>::Iterator
```

to traverse the elements of our DUNE grid as in [NPDE, Ex. 3.6.16].

HINT: In [NPDE, § 3.6.25], we learned that DUNE Entities provide access to a `Geometry` structure through the method `geometry()`. Moreover, this structure contains the method `volume()`, which returns the volume/length of the geometric entity.

Solution: See Listing 7.2 for the code.

Listing 7.2: Implementation for `getDomainArea`

```
double getDomainArea( const GridView & gv){
  double domain_area = 0.0;

  // Traverse the cells using gridview's EntityIterator of co-dimension 0
  for (auto it = gv.template begin<0>(); it != gv.template end<0>(); ++it){
    // Convert the current EntityPointer<0> to an Entity<0>
    const element_t & e = *it;
    // Obtain element's geometry (to obtain its area)
    const elem_geom_t & egeometry = e.geometry();
    // add current element's area to domain area
    domain_area +=  egeometry.volume();
  } // end traversing grid cells

  return domain_area;
}
```

(7.2b) Implement a (templated, DUNE-based) `C++` function

```
double getDomainPerimeter( GridView const& gv)
```

which takes the `GridView gv` and returns the domain's perimeter.

HINT: You should follow a similar strategy to the one proposed in subproblem (7.2a), only now we are interested in the length of the edges located in the boundary.

HINT: As explained in [NPDE, § 3.6.28], an intersection object is equipped with the methods `geometry()` and `boundary()`. Since the latter returns true when the edge is on the boundary of the grid, we will loop over Element's intersections instead of edges.

Solution: See Listing 7.3 for the code.

Listing 7.3: Implementation for `getDomainPerimeter`

```
double getDomainPerimeter( const GridView & gv){
  double domain_perimeter = 0.0;
  // Traverse the cells using gridview's EntityIterator of co-dimension 0
  for (auto it = gv.template begin<0>(); it != gv.template end<0>(); ++it){
    // Convert the current EntityPointer<0> to an Entity<0>
    const element_t & e = *it;
    // Iterate over elements intersections
    using intersect_t = GridView::Intersection;
```

```
    for (auto iit = gv.ibegin(e); iit != gv.iend(e); ++iit){
      // get current Intersection from IntersectionIterator
      const intersect_t & isect = *iit;
      // check boundary flags
      if(isect.boundary()){ // if the edge is in the boundary, add its length
     typedef typename GridView::template Codim<1>::Entity::Geometry edgeom_t;
     edgeom_t const& igeometry = isect.geometry();
     domain_perimeter += igeometry.volume();
      }
      else{ /* do nothing */ }
    } // end traversing element's intersections
  } // end traversing grid cells
  return domain_perimeter;
}
```

Listing 7.4: Implementation of `main.cc` for

```
#include <stdexcept>
#include <cassert>
#include <cstdlib>
// Dune headers
#include "../config.h"
#include <dune/common/exceptions.hh>
#include <dune/grid/alugrid.hh>
#include <dune/grid/alugrid/2d/alugrid.hh>
#include <dune/grid/common/gridfactory.hh>
#include <dune/grid/io/file/gmshreader.hh>
#include <dune/grid/io/file/vtk/subsamplingvtkwriter.hh>

using namespace std;
// Rename Dune types (for easy reading)
typedef typename Dune::ALUSimplexGrid<2,2> GridType;
typedef typename GridType::LeafGridView GridView;
typedef typename GridView::template Codim<0>::Entity element_t;
typedef typename element_t::Geometry elem_geom_t;

double getDomainArea( const GridView & gv);
double getDomainPerimeter( const GridView & gv);

int main(int argc, char *argv[]){
  try{
    ////////////////////////////////////////////////
    // READ TRIANGULAR MESH FROM GMSH FILE //
    ////////////////////////////////////////////////
    // Get mesh file name
    const std::string 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);
    cout << "creating Grid from " << FileName << endl;
    GridType *workingGrid = gridFactory.createGrid();
    workingGrid->loadBalance();

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

    // Get and print Domain's area
    cout << "Domain's total area : " << getDomainArea(gv) << endl ;

    // Get and print Domain's perimeter
    cout << "Domain's total perimeter : " << getDomainPerimeter(gv)
        << endl ;
  }
  // catch exceptions
  catch (Dune::Exception &e){
    cerr << "Dune reported error: " << e << endl;
  }
  catch (...)
    cerr << "Unknown exception thrown!" << endl;
  }
  return 0;
}
```

(7.2c) Test the methods implemented in subproblem (7.2a) and subproblem (7.2b) using the `.msh`-files you built in Problem 7.1. This means that you have to write a driver program for your functions.

HINT: A template file `main.cc` is available in the lecture `svn` repository in the folder
`assignments_codes/assignment7/Problem2`

Problem 7.3 Numerical Quadrature of a Given Function Using DUNE Data Structures

This problem is devoted to numerically integrating generic functions using local quadrature rules in the DUNE framework. In this task we are not going to rely on the quadrature facilities provided by DUNE (→ [NPDE, Def. 3.6.66]), but we will implement a simple quadrature rule directly.

In order to complete this task, you should be able to import a mesh from a `.msh`-file an use DUNE's mesh data, as in Problem 7.2.

(7.3a) Implement a `C++` function

```
template <class Function>
  double trapRuleDomain(const GridView & gv, const Function & f)
```

which gets as input the `GridView gv` and the `Function f` as a `lambda` object, and uses the 2D trapezoidal rule to numerically integrate f over the domain [NPDE, Eq. (3.3.45)].

HINT: The method `corner(int i)` available in `Geometry` might be useful, see [NPDE,

§ 3.6.25].

Solution: See Listing 7.5 for the code.

Listing 7.5: Implementation for `getDomainArea`

```
template <class Function>
double trapRuleDomain(GridView const& gv, Function const& f){
    double total = 0.0;
    // Traverse the cells using EntityIterator of co-dimension 0
    for (auto it = gv.template begin<0>(); it != gv.template
       end<0>(); ++it){
     // Convert the current EntityPointer<0> to an Entity<0>
     element_t const& e = *it;
     // get element's geometry
     elem_geom_t const& egeom = e.geometry();
     double f_int = 0.0;
     for (int i = 0; i<egeom.corners(); i++){
         // get quadrature point in the reference element
         coord_t const& global_pos = egeom.corner(i);
            auto f_val = f( global_pos ); //evaluate function
         // add contribution
         f_int+= 1.0/egeom.corners()*f_val*egeom.volume();
     }
     total += f_int;
    } // end traversing grid cells
    std::cout << "total " << total << std::endl;
    return total;
}
```

(7.3b) Write a C++ method

```
template <class Function>
double trapRuleBoundary(const GridView & gv, const Function & f)
```

which gets as input the `GridView gv` and the `Function f` as a `lambda` object, and uses the 1D trapezoidal rule [NPDE, Eq. (1.5.72)] to numerically integrate the function `f` over the boundary of the domain.

HINT: Use `GridView::IntersectionIterator` and the methods `geometry()` and `boundary()` provided in `GridView::Intersection`, see [NPDE, § 3.6.28].

Solution: See Listing 7.6 for the code.

Listing 7.6: Implementation for `getDomainPerimeter`

```
template <class Function>
double trapRuleBoundary(GridView const& gv, Function const& f){
    double total = 0.0;
    // Traverse the cells using EntityIterator of co-dimension 0
    for (auto it = gv.template begin<0>(); it != gv.template
       end<0>(); ++it){
     // Convert the current EntityPointer<0> to an Entity<0>
     element_t const& e = *it;
          // Iterate over elements intersections
```

```
    using intersect_t = GridView::Intersection;
    for (auto iit = gv.ibegin(e); iit != gv.iend(e); ++iit){
        // get current Intersection from IntersectionIterator
        intersect_t const& isect = *iit;
        // check boundary flags
        if(isect.boundary()){
            // if the edge is in the boundary, add its contribution
            typedef typename GridView::template Codim<1>::Entity::Geometry edgeom_t;
            edgeom_t const& igeom = isect.geometry();
            double f_int = 0.0;
            for (int i = 0; i<igeom.corners(); i++){
                // get node coordinates
                coord_t const& global_pos = igeom.corner(i);
                auto f_val = f( global_pos );
                // add contribution
                f_int+= 1.0/2*f_val*igeom.volume();
            }
            total += f_int;
        }
        else{ /* do nothing */ }
    } // end traversing element's intersections
  } // end traversing grid cells
  std::cout << "total " << total << std::endl;
  return total;
}
```

Listing 7.7: Implementation of `main.cc` for

```
#include <stdexcept>
#include <cassert>
#include <cstdlib>
#include <iostream>
#include <cmath>
// Dune includes
#include "../config.h"
#include <dune/common/exceptions.hh>
#include <dune/grid/alugrid.hh>
#include <dune/grid/alugrid/2d/alugrid.hh>
#include <dune/grid/common/gridfactory.hh>
#include <dune/grid/alugrid/2d/entity.hh>
#include <dune/grid/io/file/vtk/subsamplingvtkwriter.hh>
#include <dune/grid/io/file/gmshreader.hh>

const int world_dim = 2;
using calc_t=double;
// Rename Dune types (for easy reading)
using coord_t = Dune::FieldVector<calc_t, world_dim>;
using GridType = Dune::ALUSimplexGrid<2,2>;
using GridView = GridType::LeafGridView;
```

```
using element_t = GridView::template Codim<0>::Entity;
using elem_geom_t = element_t::Geometry;

template <class Function>
double trapRuleDomain(GridView const& gv, Function const& f);

template <class Function>
double trapRuleBoundary(GridView const& gv, Function const& f);

int main(int argc, char *argv[]){
    try{
     ////////////////////////////////////////////////
     // READ TRIANGULAR MESH //
     ////////////////////////////////////////////////
     // Get mesh file name
     const std::string 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::cout << "creating Grid from " << FileName << std::endl;
     GridType *workingGrid = gridFactory.createGrid();
     workingGrid->loadBalance();
     // Get the Gridview
     GridView gv    = workingGrid->leafGridView();

     ////////////////////////////////////////////////////////
     // PERFORM NUMERICAL QUADRATURE OF A GIVEN FUNCTION //
     ////////////////////////////////////////////////////////
     // Over the domain
     auto f = [](coord_t const& x){ return x[0]*x[1];};
     double int_f = trapRuleDomain(gv, f);
     // Over the boundary
     auto f2 = [](coord_t const& x){ return x[0];};
     double int_f2 = trapRuleBoundary(gv, f2);

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

(7.3c) Test the methods implemented in subproblem (7.3a) and subproblem (7.3b) using the `lambda` function

```
auto f = [] {return 1.0;}
```

and the `.msh`-files you built in Problem 7.1. Are they close to your results in subproblem (7.2c)?

HINT: A template file `main.cc` is available in the lecture `svn` repository in the folder

`assignments_codes/assignment7/Problem3`

Problem 7.4 Linear Finite Element Implementation for 2D Reaction-diffusion using DUNE (Core problem)

In Problem 6.1 (last homework sheet), we considered the following Neumann problem on the unit square $\Omega = [0,1]^2$ with homogeneous Neumann data and reaction term (cf. [NPDE, Eq. (3.1.4)]):

$$u \in H^1(\Omega): \quad \underbrace{\int_\Omega \mathbf{grad}\, u \cdot \mathbf{grad}\, v + u\, v \,\mathrm{d}\boldsymbol{x}}_{:=\mathsf{a}(u,v)} = \underbrace{\int_\Omega f v \,\mathrm{d}\boldsymbol{x}}_{:=\ell(v)} \quad \forall v \in H^1(\Omega), \tag{7.4.1}$$

and developed an efficient MATLAB code to discretize (7.4.1) on a triangular mesh using *linear finite elements*. Now we will follow a similar approach to do this with a C++ code using a DUNE-style interface.

As usual we will rely on `Dune::ALUSimplexGrid<2,2>` as grid implementation and access the grid data structures through the `GridView` (*cf.* [NPDE, § 3.6.15]).

Template files for the different classes you will need to write are available in the lecture `svn` repository in the folder

`assignments_codes/assignment7/Problem4`

together with the implementation of the class `MatrixAssembler` shown in [NPDE, Ex. 3.6.48].

(7.4a) As we learned in [NPDE, Section 3.3.5], cell oriented assembly entails knowing the global numbering of the basis functions associated with the vertices of each triangle. To this end, we introduced the data structure of the DofHandler which provides a local → global index mapping, as detailed in [NPDE, § 3.6.37], see also [NPDE, Code 3.6.17] and the link to the online DUNE documentation.

Taking the cue of [NPDE, Code 3.6.42] complete the following class

```
template <class GridView_t>
class DofHandler{
public:
  using calc_t   = double;
  using GridView = GridView_t;
  using index_t  = typename GridView::IndexSet::IndexType;
  using Element  = typename GridView::template Codim<0>::Entity;
  enum { wd=GridView::dimension };
  enum { K=1 }; //linear finite element
  DofHandler(GridView const& gridview):
  gv(gridview),set(gv.indexSet()),offset(set.size(wd)) {};
  // Operator providing the local → global index mapping
  template <class Element>
  index_t operator()(Element const& e, index_t local_idx) const;
  bool active(index_t gidx) const { return true; } ;
  size_t size_loc(Element const &e) const{ return 3;};
  size_t size() const { return set.size(wd); }
```

```
  GridView const& gv;
private:
  typename GridView::IndexSet const& set;
};
```

by implementing the method

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

Taking an Element and a local index of a given dof, and returning its global index.

HINT: Use `GridView::IndexSet`.

Solution: See Listing 7.8 for the code.

Listing 7.8: Implementation for `DofHandler`

```
#ifndef LDOFHANDLER_HPP
#define LDOFHANDLER_HPP

#include <dune/grid/common/indexidset.hh>
#include <dune/geometry/referenceelements.hh>
#include <cassert>

namespace NPDE15{
  template <class GridView_t>
  class LDofHandler{
  public:
    using calc_t=double;
    using GridView=GridView_t;
    using index_t=typename GridView::IndexSet::IndexType;
    using Element = typename GridView::template Codim<0>::Entity;
    enum { world_dim=GridView::dimension };
    enum { K=1 };

    LDofHandler(GridView const& gridview) : gv(gridview),
       set(gv.indexSet()) {};

    template <class Element>
    index_t operator()(Element const& e, index_t dof) const{
      index_t corners=e.geometry().corners(); // number of corners
      assert(dof<corners);
      return set.subIndex(e, dof, world_dim);
    }

    bool active(index_t gidx) const{ return true;}
    size_t size_loc(Element const &e) const { return 3; }
    size_t size() const{ return set.size(world_dim); }
    GridView const& gv;

  private:
    typename GridView::IndexSet const& set;
```

```
  };
}
#endif
```

(7.4b) Recall from [NPDE, Ex. 3.6.48], that to build the global system matrix with our cell-oriented strategy, we introduced the class `MatrixAssembler`. This structure is equipped with the method

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

which for each element calls the LocalAssembler to get the element matrix and distributes its contribution to the triplets.

Implement a class `AnalyticalLocalLaplace` which provides a method

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

to compute the element matrix associated to the bilinear form

$$\mathsf{a}_1(u,v) = \int_\Omega \mathbf{grad}\, u \cdot \mathbf{grad}\, v \,\mathrm{d}\boldsymbol{x}, \quad u,v \in H^1(\Omega),$$

and linear Lagrangian finite elements.

HINT: The solution of subproblems (6.1a) and (6.1b) might be useful. The implementation in MATLAB was already discussed in [NPDE, Section 3.3.5] and is given in [NPDE, Code 3.3.25]. This function has to be translated into C++. In order to perform linear algebra operations, you may rely on the facilities of the Eigen library, see [NPDE, Rem. 3.6.45].

Solution: See Listing 7.9 for the code.

Listing 7.9: Implementation for `AnalyticalLocalLaplace`

```
#ifndef ANALYTICALLOCALLAPLACE_HPP_
#define ANALYTICALLOCALLAPLACE_HPP_

#include <dune/common/fmatrix.hh>

namespace NPDE15{
  class AnalyticalLocalLaplace{
  public:
    using calc_t=double;
    using ElementMatrix = typename Dune::FieldMatrix<calc_t,3,3>;

    AnalyticalLocalLaplace() {};

    template <class Element>
    void operator()(Element const& e, ElementMatrix &local) const{
      local=0;
      // get element's geometry and area
      auto const& egeom = e.geometry();
      auto elem_area = egeom.volume();
      // Compute gradients using analytical formula (rotated)
```

```
      // Omit factor 1/(2*area) as it will be computed at the end
      Dune::FieldMatrix<calc_t,3,2> grad;
      grad[0] = (egeom.corner(1)- egeom.corner(2));
      grad[1] = (egeom.corner(2)- egeom.corner(0));
      grad[2] = (egeom.corner(0)- egeom.corner(1));
      // Finally compute the matrix
      for (unsigned i=0;i<local.N();++i){
     for (unsigned j=0;j<local.M();++j){
       local[i][j]= grad[i]*grad[j]/(4.*elem_area);
     }
      } // end for loop
    }

  };
}
#endif

/*
// Alternative with Eigen
#ifndef ANALYTICALLOCALLAPLACE_HPP_
#define ANALYTICALLOCALLAPLACE_HPP_

#include <Eigen/Dense>

namespace NPDE15{

  class AnalyticalLocalLaplace{
  public:
    using calc_t=double;
    using ElementMatrix = Eigen::Matrix<calc_t,3,3>;

    AnalyticalLocalLaplace() {};

    template <class Element>
    void operator()(Element const& e, ElementMatrix &local) const{
      local.setZero();
      // Get element's geometry, and use it to obtain its area and
         vertices
      auto const& egeom = e.geometry();
      auto elem_area = egeom.volume();
      Eigen::Matrix<calc_t,3,2> Vertices;
      for(int i=0; i<egeom.corners(); ++i){
     auto vert = egeom.corner(i);
     Vertices.row(i) << vert[0], vert[1];
      }
      // Compute gradients using analytical formula
      Eigen::Vector3d auxOnes = Eigen::Vector3d::Ones();
      ElementMatrix auxMat; auxMat << auxOnes, Vertices;
      ElementMatrix auxgrad = (auxMat.inverse());
      // Fill local matrix
```

```
      local = elem_area * (auxgrad.block<2,3>(1,0)).transpose() * auxgrad.block<2,3>(1,0);
    }

  };
}
#endif
*/
```

(7.4c) Implement a class `AnalyticalLocalMass` which provides a method

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

to compute the element matrix associated to the bilinear form

$$\mathsf{a}_2(u,v) = \int_\Omega u\,v\,\mathrm{d}\boldsymbol{x}, \quad u,v \in L^2(\Omega)\ ,$$

and linear Lagrangian finite elements on triangular elements.

HINT: Compute the entries of the element matrix by analytic evaluation of the two-dimensional integrals. You already did this to solve subproblem (6.1c).

Solution: See Listing 7.10 for the code.

Listing 7.10: Implementation for `AnalyticalLocalMass`

```
#ifndef ANALYTICALLOCALMASS_HPP_
#define ANALYTICALLOCALMASS_HPP_

#include <dune/common/fmatrix.hh>

namespace NPDE15{
  class AnalyticalLocalMass{
  public:
    using calc_t=double;
    using ElementMatrix = typename Dune::FieldMatrix<calc_t,3,3>;

    AnalyticalLocalMass() {};

    template <class Element>
    void operator()(Element const& e, ElementMatrix &locMassMat) const{
      // Query element's geometry and it's area
      auto const& egeom = e.geometry();
      auto elem_area = egeom.volume();
      locMassMat = elem_area/12.;
      // Finally compute the matrix using analytic formula
      for (unsigned i=0;i<locMassMat.N();++i){
     locMassMat[i][i]= elem_area/6.;
      }
    }
```

```

  };
}
#endif

/*
// Alternative with Eigen
#ifndef ANALYTICALLOCALMASS_HPP_
#define ANALYTICALLOCALMASS_HPP_

#include <Eigen/Dense>

namespace NPDE15{

  class AnalyticalLocalMass{
  public:
    using calc_t=double;
    using ElementMatrix = Eigen::Matrix<calc_t,3,3>;

    AnalyticalLocalMass() {};

    template <class Element>
    void operator()(Element const& e, ElementMatrix &locMassMat) const{
      // Query element's geometry and it's area
      auto const& egeom = e.geometry();
      auto elem_area = egeom.volume();
      // Finally compute the matrix using analytic formula
      locMassMat.setConstant(elem_area/12.);
      locMassMat += (locMassMat.diagonal()).asDiagonal();
    }

  };
}
#endif
*/
```

(7.4d) Now we focus on the global assembly of the right han side vector using DUNE (see [NPDE, Eq. (3.6.51)]).

Following the structure shown in [NPDE, Eq. (3.6.52)], implement the class `VectorAssembler` to assemble the right-hand side vector with linear finite element.

HINT: Note this is the C++ version of subproblem (6.1g).

Solution: See Listing 7.11 for the code.

Listing 7.11: Implementation for `VectorAssembler`

```
#ifndef VECTORASSEMBLER_HPP
#define VECTORASSEMBLER_HPP

```

```
#include <vector>

namespace NPDE15{
  template <class DofHandler>
  class VectorAssembler{
  public:
    using calc_t=double;
    using GridView=typename DofHandler::GridView;
    enum { K=DofHandler::K };
    enum { world_dim=GridView::dimension };

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

    template <class RHSVector, class LocalAssembler>
    void operator()(RHSVector &Phi, LocalAssembler const&
       local_assembler) const{
      using ElementVector = typename LocalAssembler::ElementVector;
      // loop over cells
      for (auto it=gv.template begin<0>();it!=gv.template
         end<0>();++it){
     ElementVector LocalPhi;
     auto const& e=*it;
     // save local contribution into LocalPhi
     local_assembler(e, LocalPhi);
     // distribute to global vector
     for (unsigned loc_idx=0;loc_idx<LocalPhi.size();++loc_idx){
       //global index of local shape function
       unsigned glob_idx = dofh(e, loc_idx);
       if(dofh.active(glob_idx))
         Phi[glob_idx]+=LocalPhi[loc_idx];
     } // end for over local dofs
      } // end loop over cells
    }

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

#endif
```

(7.4e) Implement the class

```
template <class Function>
  class LocalFunction
```

which provides a method

```
template <class Element>
```

```
void operator()(Element const& e, ElementVector &local) const;
```

to compute the element vector associated to the linear form $\ell(v) = \int_\Omega f v \, \mathrm{d}\boldsymbol{x}$, and linear Lagrangian finite elements.

Notice that the function f should be passed to the constructor in procedural form. Therefore the entries of the element vectors can be computed only approximately by means of numerical quadrature, *cf.* [NPDE, § 3.3.44]. Use *2D trapezoidal quadrature rule*.

HINT: Note this is similar to what you had to implement in subproblem (6.1e).

HINT: [NPDE, Code 3.6.99] might be useful.

Solution: See Listing 7.12 for the code.

Listing 7.12: Implementation for `LocalFunction`

```
#ifndef LLOCALFUNCTION_HPP_
#define LLOCALFUNCTION_HPP_

#include <stdexcept>
#include <vector>
#include <dune/common/exceptions.hh>
#include <dune/common/fvector.hh>

namespace NPDE15{
  template <class Function>
  class LLocalFunctionC{
  public:
    using calc_t = double;
    using ElementVector = typename Dune::FieldVector<calc_t,3>;

    LLocalFunctionC(Function const& _f) : f(_f) {};

    template <class Element>
    void operator()(Element const& e, ElementVector &local) const;

  private:
    Function const& f;
  };

  // template deduction helper
  template <class Function>
  LLocalFunctionC<Function> LLocalFunction(Function const& f){
    return LLocalFunctionC<Function>(f);
  }

  template <class Function>
  template <class Element>
  void LLocalFunctionC<Function>::operator()(Element const& e,
      ElementVector &local) const{
    auto const& egeom = e.geometry();
    auto elem_area = egeom.volume();
    for (unsigned i=0;i<3;++i){
```

```
      // local contribution computed by "2D-trapezoidal rule
         inspired quadrature"
      local[i] = elem_area/3 * f(egeom.corner(i));
    }

  };
}
#endif
```

(7.4f) Finally, as we already have all the required pieces to assemble the system matrix and right hand side vector, we are ready to find an approximate solution of (7.4.1).

Write a `main.cc` that:

- Builds a grid from `.msh` file.
- Gets the grid view and uses it to initialize the DofHandler.
- Creates the system matrix using an `Eigen` Triplets in two stages:
  - Fills the triplets with the contribution of $\mathsf{a}_1(u, v)$.
  - Does the same now with $\mathsf{a}_2(u, v)$.
- Construct the system matrix `A` from the triplets.
- Creates the right hand side vector `Phi` using and `Eigen` vector and the function.

```
auto f = [](Coordinate const& x){ return (1.0 +
   2*M_PI*M_PI)*cos(M_PI*x[0])*cos(M_PI*x[1]); }
```

- Solves the system using `Pardiso`.
- Outputs the solution in `Vtk` format.

You should then test it using the mesh `square_1024.msh` provided in the repository.

HINT: Use the short-hand `u = Phi/A` to solve the system.

HINT: Use

```
template<class GridView>
template<class V >
Dune::VtkWriter<GridView>::addVertexData( const V & v, const
   std::string & name, int ncomps = 1)
```

to write the solution vector to a `Vtk` format file (→ Link).

Solution: See Listing 7.13 for the code.

Listing 7.13: Implementation for `main.cc`

```
#include <stdexcept>
#include <cassert>
#include <cstdlib>
#include <iostream>
#include <cmath>
```

```
#include <Eigen/Sparse>
#include <Eigen/Dense>
#include <Eigen/Cholesky>
// Dune Includes
#include "../config.h"
#include <dune/common/exceptions.hh>
#include <dune/grid/alugrid.hh>
#include <dune/grid/alugrid/2d/alugrid.hh>
#include <dune/grid/common/gridfactory.hh>
#include <dune/grid/io/file/vtk/subsamplingvtkwriter.hh>
#include <dune/grid/io/file/gmshreader.hh>
// include the NPDE15 headers used in this example
#include "npde15/Pardiso.hpp"
#include "npde15/global/MatrixAssembler.hpp"
// LFEM-headers
#include "LDofHandler.hpp"
#include "LLocalFunction.hpp"
#include "AnalyticalLocalLaplace.hpp"
#include "AnalyticalLocalMass.hpp"
#include "LVectorAssembler.hpp"
#include "InterpFunction.hpp"

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

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

    Dune::GridFactory<GridType> factory;
    Dune::GmshReader<GridType>::read(factory, FileName.c_str(),
        false, true);
    GridType *WorkingGrid = factory.createGrid();
    WorkingGrid->loadBalance();
```

```
    // get the leafgridview
    GridView gv = WorkingGrid->leafGridView();
    // initialize dof-handler
    DofHandler dofh(gv);
    unsigned N = dofh.size();
    cout << "Solving for N =" << N << " unknowns.\n";

    ////////////////////////////////////////////////////////////
    // CREATE AND SOLVE SYSTEM USING ANALYTIC IMPLEMENTATION //
    ////////////////////////////////////////////////////////////
    // load vector (with known solution)
    auto f = [](Coordinate const& x){
      return (1.0 + 2*M_PI*M_PI)*cos(M_PI*x[0])*cos(M_PI*x[1]);
    };

    // assemble rhs
    Vector Phi(N);
    Phi.setZero();
    NPDE15::VectorAssembler<DofHandler> make_vector(dofh);
    make_vector(Phi, NPDE15::LLocalFunction(f));

    // assemble the system matrix (laplacian with mass)

    std::vector<Eigen::Triplet<calc_t>> triplets;
    NPDE15::MatrixAssembler<DofHandler> make_matrix(dofh);
    make_matrix(triplets, NPDE15::AnalyticalLocalLaplace());
    make_matrix(triplets, NPDE15::AnalyticalLocalMass());

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

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

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

    ////////////////////////////////////////////////////////
    // CREATE ANALYTICAL EXACT SOLUTION FOR THE GIVEN RHS //
    ////////////////////////////////////////////////////////
    // solution
    auto u_sol=[](Coordinate const& x){
      return cos(M_PI*x[0])*cos(M_PI*x[1]);
    };

    Vector u_ex(N); u_ex.setZero();
    NPDE15::InterpFunction<DofHandler> interpolator(dofh);
    interpolator(u_ex, u_sol);

    ////////////////////////////////////////////////////////////
```

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

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

Published on 01.04.2015.
To be submitted on 15.04.2015.

References

[NPDE] Lecture Slides for the course "Numerical Methods for Partial Differential Equations".SVN revision # 74771.

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

Last modified on April 15, 2015