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`.

(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`.

(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.

(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)`.

(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.

(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.

(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 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].

(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].

(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 $\to$ 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
```

```
10    DofHandler(GridView const& gridview):
11    gv(gridview),set(gv.indexSet()),offset(set.size(wd)) {};
12    // Operator providing the local → global index mapping
13    template <class Element>
14    index_t operator()(Element const& e, index_t local_idx)
        const;
15    bool active(index_t gidx) const { return true; } ;
16    size_t size_loc(Element const &e) const{ return 3;};
17    size_t size() const { return set.size(wd); }
18    GridView const& gv;
19  private:
20    typename GridView::IndexSet const& set;
21  };
```

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`.

(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].

(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).

(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).

(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.

(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).

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 # 74741.

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

Last modified on April 2, 2015