DUNE PDELab Tutorial 05
Adaptive Finite Element Method for a Nonlinear Poisson Equation

DUNE/PDELab Team

August 31, 2021

Contents

1 Introduction

In this tutorial we solve the nonlinear Poisson equation from tutorial 01 using adaptive grid refinement. The finite element solution on a given mesh is used to compute local error indicators that can be used to iteratively reduce the discretization error. It is assumed that the nonlinearity of the PDE is not too strong, so that it may be approximated by linearizing around the current finite element solution. This allows the calculation of local error estimates that quantify the discretization error on each element of the mesh.

Depends On

This tutorial depends on tutorial 01 which discusses the solution of the considered partial differential equation. It is assumed that you have worked through tutorial 01 before. Additionally, the error estimator has to assemble skeleton terms, which are explained in greater detail in tutorial 02. It may therefore be useful to have worked through that tutorial as well, but the actual method discussed there is not relevant here.

2 Problem Formulation

We consider the following nonlinear Poisson equation with Dirichlet and Neumann boundary conditions as introduced in tutorial 01:

$$-\Delta u + q(u) = f \qquad \text{in } \Omega, \tag{1a}$$

$$u = g \qquad \text{on } \Gamma_D \subseteq \partial\Omega, \tag{1b}$$

$$-\nabla u \cdot \nu = j \qquad \text{on } \Gamma_N = \partial\Omega \setminus \Gamma_D. \tag{1c}$$

$\Omega \subset \mathbb{R}^d$ is a domain, $q : \mathbb{R} \to \mathbb{R}$ is a given, possibly nonlinear function and $f : \Omega \to \mathbb{R}$ is the source term and ν denotes the unit outer normal to the domain.

The weak formulation of this problem is derived by multiplication with an appropriate test function and integrating by parts. This results in the abstract problem:

$$\text{Find } u \in U \text{ s.t.:} \quad r^{\text{NLP}}(u, v) = 0 \quad \forall v \in V, \tag{2}$$

with the continuous residual form

$$r^{\text{NLP}}(u, v) = \int_\Omega \nabla u \cdot \nabla v + (q(u) - f)v \, dx + \int_{\Gamma_N} jv \, ds$$

and the function spaces $U = \{v \in H^1(\Omega) : \text{“}v = g\text{” on } \Gamma_D\}$ and $V = \{v \in H^1(\Omega) : \text{“}v = 0\text{” on } \Gamma_D\}$. We assume that q is such that this problem has a unique solution.

The example application uses a domain Ω that is L-shaped and given by

$$\Omega = \left\{(x, y) \in \mathbb{R}^2 \colon |x| < 1, |y| < 1, (x < 0 \vee y > 0)\right\}.$$

We set $f = 0$ and $q(u) = 0$, and only consider Dirichlet boundary conditions. The resulting PDE in residual form is

$$r^{\text{NLP}}(u, v) = \int_\Omega \nabla u \cdot \nabla v \, dx,$$

and the function

$$u(r, \theta) = r^{2/3} \cdot \sin\left(\frac{2}{3}\theta\right)$$

in polar coordinates is one of its solutions on the given domain. We choose the restriction of this function to the boundary of Ω for the Dirichlet boundary condition.

3 Adaptive Grid Refinement

In the previous tutorials a fixed finite element space V_h was used, based on a fixed finite element mesh with an ordered set

$$\mathcal{X}_h = \{x_1, \ldots, x_N\} \tag{3}$$

of vertices and an ordered set

$$\mathcal{T}_h = \{T_1, \ldots, T_M\} \tag{4}$$

of elements. The task of choosing an appropriate mesh was not discussed in detail.

Any choice of finite element space will lead to discretization errors due to the finite-dimensional approximation of the true solution of the PDE, but the size of this error can be controlled by choosing an adequate mesh. Let $u_h \in V_h$ be the finite element solution and $\|\cdot\|$ an appropriate norm. The goal is then

$$\|u - u_h\| \leq \text{TOL}$$

with a given tolerance TOL, while keeping the complexity and size of V_h as low as possible.

The *a-priori* error estimates of formal proofs are not suitable for this task, since they contain a constant C that is typically unknown and don't provide information about the spatial distribution of the error. For this reason one is interested in *a-posteriori* error estimates of the form

$$\|u - u_h\| \leq \gamma(u_h) = \left(\sum_{T \in \mathcal{T}_h} \gamma_T^2(u_h)\right)^{1/2}, \tag{5}$$

where $\gamma_T(u_h)$ is a *local error estimator* for the element T. Such an error estimate should:

- have comparatively low cost of calculation to make an adaptive approach feasible
- accurately describe the spatial distribution and size of the discretization error

The local estimates $\gamma_T(u_h)$ can then be used in a strategy that tries to minimize $\gamma(u_h)$ for fixed $\dim(V_h)$ and thereby find a mesh with low discretization error $\|u - u_h\|$.

If the considered PDE is linear, i.e. $q(u)$ is an affine linear function, then it can be shown [2, 1] that such an estimate of the discretization error is given by

$$\gamma_T^2(u_h) = h_T^2 \|R_T(u_h)\|_{0,T}^2 + \sum_{F \in \mathcal{F}_h \cap \partial T} h_T \|R_F(u_h)\|_{0,F}^2$$

with the *element residuals*

$$R_T(u_h) = f + \Delta u_h - q(u_h)$$

and the *face residuals*

$$R_F(u_h) = \begin{cases} 2^{-1/2}[-(\nabla u_h) \cdot \nu] & F \in \mathcal{F}_h^i \\ -(\nabla u_h) \cdot \nu - j & F \in \mathcal{F}_h^N \end{cases}$$

Here h_T is the local mesh width of the element T, $\mathcal{F}_h$ the set of faces of the elements in $\mathcal{T}_h$, ∂T the boundary of element T, $\mathcal{F}_h^i \subset \mathcal{F}_h$ the set of interior faces, $\mathcal{F}_h^N \subset \mathcal{F}_h$ the set of Neumann boundary faces and ν the unit normal vector of the face F pointing from one of the elements T_F^- that belong to F to the other T_F^+. $\|\cdot\|_{0,T}$ is the L^2 norm on T, $\|\cdot\|_{0,F}$ that on F, and $[\cdot]$ the jump of the given expression across the face F in the direction of ν. Broadly speaking, the element residual R_T measures to which degree u_h solves the PDE on the element T, and the face residual R_F measures the consistency of the flux $-(\nabla u_h) \cdot \nu$ between the elements.

With these definitions one can show that

$$\|u - u_h\|_{1,\Omega} \leq C\gamma(u_h) = C \left(\sum_{T \in \mathcal{T}_h} \gamma_T^2(u_h) \right)^{1/2}$$

with $\|\cdot\|_{1,\Omega}$ the H^1 norm on Ω. The right hand side of this inequality does not depend on u, which means it can be evaluated and used to control the discretization error.

Several aspects have to be considered:

- The derivation of the error estimator given above does not require additional regularity beyond $u \in H^1(\Omega)$. This is critical, since error control is especially important in problems with low regularity.
- The constant C in the estimate is usually not known exactly, and this should be taken into account when defining the tolerance TOL.
- The error estimate may converge with a lower order than the actual discretization error which yields a very pessimistic stopping criterion. If the error estimate converges with the same order, then the estimator is called *efficient*. The lowest applicable constant C is then the *efficiency index*.

All of these considerations only hold if the PDE is linear. If the function $q(u)$ is nonlinear, then the estimate $\gamma(u_h)$ is no longer guaranteed to be an upper bound for the discretization error $\|u - u_h\|_{1,\Omega}$. However, it may still be used as an error indicator that is used to refine a given mesh. This can produce misleading results, and therefore the conclusions drawn from the estimate should be rather conservative. The error estimate will become more reliable the closer the finite element solution u_h is to the exact solution u, as long as the function $q(u)$ is regular enough to be linearized around the exact solution.

4 Realization in PDELab

The structure of the code is very similar to that of tutorial 01. It consists of the following files:

1) The ini-file `tutorial05.ini` holds parameters read by various parts of the code which control the execution.

2) The main file `tutorial05.cc` includes the necessary C++, DUNE and PDELab header files and contains the `main` function where the execution starts. The purpose of the `main` function is to instantiate DUNE grid objects and call the `driver` function.

3) File `driver.hh` instantiates the necessary PDELab classes for solving a nonlinear stationary problem and solves the problem. The file is similar to that of tutorial 01 but adds a loop for iterative refinement.

4) File `nonlinearpoissonfem.hh` contains the class `NonlinearPoissonFEM`, the local operator from tutorial 01.

5) File `nonlinearpoissonfemestimator.hh` contains an additional local operator `NonlinearPoissonFEMEstimator` that implements the local error indicators.

6) Finally, the tutorial provides a parameter class in file `problem.hh` and some mesh files.

4.1 Ini-File

The ini-file allows the user to set various parameters for the execution of the program:

```
[grid]
dim=2
manager=ug
```

The `grid` section allows choosing between `UGGrid` and `AluGrid`. The `dim` parameter could be used to select a different space dimension, but the tutorial only supplies a two-dimensional domain.

```
[grid.twod]
filename=ldomain.msh
```

The `grid.twod` section is used to specify the gmsh-file that is used for the coarse grid.

```
[fem]
degree=1
steps=10
uniformlevel=0
tol=0.05
fraction=0.8
```

The `fem` section provides the parameters for the finite element method. `degree` is the polynomial degree for the finite element space. The remaining parameters control the adaptive refinement: `steps` specifies the maximum number of iterations for the refinement loop, `uniformlevel` the number of iterations that should use global refinement in the beginning, `tol` denotes the error tolerance used as a stopping criterion, and `fraction` allows setting the number of elements that should be marked in each step.

```
[problem]
eta=0.0
```

The `problem` section provides parameters for the specific problem to be solved. The tutorial solves the Laplace equation with a known reference solution, and therefore $\eta = 0$ in the ini-file. However, the error estimator remains applicable if different values are chosen.

```
[output]
filename=adaptive_ug
subsampling=1
```

The `output` section controls the output of the solution to a `vtk`-file using DUNE's `SubsamplingVTKWriter`. The user can give the name of the output file and specify the number of subsampling intervals.

4.2 Function `main`

The `main` function in `tutorial05.cc` is very similar to the ones in `tutorial00.cc` and `tutorial01.cc`. It starts by getting a reference to the `Dune::MPIHelper` singleton and opens and reads in the ini-file. These parts are not repeated here.

Then there are two sections where `Dune::Grid` objects are instantiated and the `driver` function is called. Since the grid manager and the polynomial degree are template parameters of various classes but the user should be able to select these during run-time in the ini-file all the different cases are selected with `if`-statements within which the appropriate classes are instantiated. We present the section using `UGGrid` here, the section for `AluGrid` is nearly identical.

```
    // use UG grid if available and selected
    if (dim==2 && gridmanager=="ug")
      {
#if HAVE_UG
        typedef Dune::UGGrid<2> Grid;
        std::string filename = ptree.get("grid.twod.filename",
                                         "ldomain.msh");
        Dune::GridFactory<Grid> factory;
        Dune::GmshReader<Grid>::read(factory,filename,true,true);
        std::shared_ptr<Grid> gridp(factory.createGrid());
        if (degree==1) driver<Grid,1>(*gridp,ptree);
        if (degree==2) driver<Grid,2>(*gridp,ptree);
        if (degree==3) driver<Grid,3>(*gridp,ptree);
        if (degree==4) driver<Grid,4>(*gridp,ptree);
#else
```

```
        std::cout << "You␣selected␣ug␣as␣grid␣manager␣"
          << "but␣ug␣was␣not␣found␣during␣installation"
          << std::endl;
#endif
      }
```

This part of the code is executed if `UGGrid` has been chosen as grid manager and Dune has been compiled with support for UG. A grid is constructed from the gmsh-file that was defined in the ini-file, and afterwards the `driver` function is called with the correct polynomial degree for the finite element space. Note that the code of the implementation could easily be extended to 3D by supplying a matching mesh file and adjusting the lines above, since the rest of the implementation is independent of the space dimension.

4.3 Function `driver`

The function `driver` alternates between solving the problem on the current mesh and using the current solution for adaptive refinement. These two steps are repeated until the error estimate is below the prescribed tolerance or the maximum number of iterations is reached. The parts of the problem that are not affected by the changing mesh are instantiated outside of the loop.

```
  // make user functions
  RF eta = ptree.get("problem.eta",(RF)1.0);
  Problem<RF> problem(eta);
  auto glambda =
    [&](const auto& e, const auto& x){return problem.g(e,x);};
  auto g = Dune::PDELab::makeGridFunctionFromCallable(gv,glambda);
  auto blambda =
    [&](const auto& i, const auto& x){return problem.b(i,x);};
  auto b = Dune::PDELab::makeBoundaryConditionFromCallable(gv,blambda);
```

The parameter class `Problem` defines functions for the right hand side f, the source term q and the value of the Dirichlet boundary condition g. These functions are independent of the chosen discretization. This also holds for the lambda functions that encapsulate the Dirichlet boundary condition and its value.

```
  // Make grid function space
  typedef Dune::PDELab::PkLocalFiniteElementMap<GV,DF,RF,degree> FEM;
  FEM fem(gv);
  typedef Dune::PDELab::ConformingDirichletConstraints CON;
  typedef Dune::PDELab::ISTL::VectorBackend<> VBE;
  typedef Dune::PDELab::GridFunctionSpace<GV,FEM,CON,VBE> GFS;
  GFS gfs(gv,fem);
  gfs.name("Vh");
```

The grid function space `GFS` is a representation of the finite element space V_h. Most of its internals are independent of the actual mesh. The basis functions defined by the finite element map `FEM`, the constraints container `CON` and the vector backend `VBE` do not need to be modified when the mesh is refined. The only parts of the

grid function space that have to be updated are the local to global map and the constrained indices, which will be considered later in this function.

```
// Assemble constraints
typedef typename GFS::template ConstraintsContainer<RF>::Type CC;
CC cc;
Dune::PDELab::constraints(b,gfs,cc); // assemble constraints

// A coefficient vector
using Z = Dune::PDELab::Backend::Vector<GFS,RF>;
Z z(gfs); // initial value

// Fill the coefficient vector
Dune::PDELab::interpolate(g,gfs,z);
```

Then the constraints of the initial mesh and an initial guess for the finite element solution are assembled.

```
// adaptation loop
// only grid function space and coefficient vector
// live outside this loop
int steps = ptree.get("fem.steps",(int)3);
int uniformlevel = ptree.get("fem.uniformlevel",(int)2);
for (int i=0; i<steps; i++)
{
```

The central loop of the function starts by reading in the maximum number of refinement iterations and the number of steps that should use global refinement. This can be used to guarantee a maximum diameter of the elements of the resulting meshes. From this point on all statements are part of the loop that is repeated until the error estimate is small enough or the maximum number of iterations is reached.

```
  std::stringstream s;
  s << i;
  std::string iter;
  s >> iter;
  std::cout << "Iteration:␣" << iter
    << "\thighest␣level␣in␣grid:␣" << grid.maxLevel()
    << std::endl;
  std::cout << "constrained␣dofs=" << cc.size()
    << "␣of␣" << gfs.globalSize() << std::endl;
```

Information about the refinement level and dimension of the current finite element space is printed at the beginning of each iteration.

```
  // Make local operator
  typedef NonlinearPoissonFEM<Problem<RF>,FEM> LOP;
  LOP lop(problem);

  // Make a global operator
  typedef Dune::PDELab::ISTL::BCRSMatrixBackend<> MBE;
  MBE mbe((int)pow(1+2*degree,dim));
  typedef Dune::PDELab::GridOperator<
```

```
      GFS,GFS,  /* ansatz and test space */
      LOP,      /* local operator */
      MBE,      /* matrix backend */
      RF,RF,RF, /* domain, range, jacobian field type */
      CC,CC     /* constraints for ansatz and test space */
      > GO;
    GO go(gfs,cc,gfs,cc,lop,mbe);
```

Then the local operator and global operator of the nonlinear Poisson equation are created. The local operator is identical to the one of tutorial 01. It does not store information about the mesh and could therefore also be created outside of the loop. The global operator, however, has to take the discretization into account and therefore has to be updated after each iteration.

```
    // Select a linear solver backend
    typedef Dune::PDELab::ISTLBackend_SEQ_CG_AMG_SSOR<GO> LS;
    LS ls(100,0);

    // solve nonlinear problem
    Dune::PDELab::NewtonMethod<GO,LS> newton(go,ls);
    newton.setParameters(ptree.sub("newton"));
    newton.apply(z);
```

The next few lines construct a linear solver backend and an instance of the Newton solver and solve the nonlinear problem on the current mesh.

```
    // set up error estimator
    typedef Dune::PDELab::P0LocalFiniteElementMap<DF,RF,dim> P0FEM;
    P0FEM p0fem(Dune::GeometryTypes::simplex(dim));
    typedef Dune::PDELab::NoConstraints NCON;
    typedef Dune::PDELab::GridFunctionSpace<GV,P0FEM,NCON,VBE> P0GFS;
    P0GFS p0gfs(gv,p0fem);
    typedef NonlinearPoissonFEMEstimator<Problem<RF>,FEM> ESTLOP;
    ESTLOP estlop(problem);
    typedef Dune::PDELab::EmptyTransformation NCC;
    typedef Dune::PDELab::GridOperator<
      GFS,P0GFS,    /* one value per element */
      ESTLOP,       /* operator for error estimate */
      MBE,RF,RF,RF, /* same as before */
      NCC,NCC       /* no constraints */
      > ESTGO;
    ESTGO estgo(gfs,p0gfs,estlop,mbe);
```

These lines construct the error estimator and local error indicators. Each of these indicators is a single value γ_T that is associated with one of the elements T, and we use a finite element map `P0FEM` for piecewise constant functions to store these values. The function space does not need additional constraints, similar to the finite volume space of tutorial 02, and therefore the `NoConstraints` class is passed to the grid function space. Then a local operator for the error estimator is constructed. The parameter class of the PDE is passed as a template argument, since the estimator needs access to the problem definition. These components are then used to

construct a grid operator of type `ESTGO` which is a representation of the element-wise computation of the error estimate.

```
    // compute local error contribution and global error
    using Z0 = Dune::PDELab::Backend::Vector<P0GFS,RF>;
    Z0 z0(p0gfs,0.0);
    estgo.residual(z,z0);
    auto estimated_error = sqrt(z0.one_norm());
    std::cout << "Estimated␣error␣in␣step␣" << i
      << "␣is␣" << estimated_error << std::endl;
```

A vector `z0` for the local estimates is created, and the error estimate is computed as the residual of the estimation grid operator. This way, the infrastructure of the assembler for the residual form of the PDE can be reused for this task. The result is a vector containing the squared values of the local error estimates. The norm of the estimated error is then printed to provide feedback about the refinement process.

```
    // vtk output
    std::cout << "VTK␣output" << std::endl;
    Dune::SubsamplingVTKWriter<GV>
      vtkwriter(gv,Dune::refinementIntervals(ptree.get("output.subsampling",
    typedef Dune::PDELab::DiscreteGridFunction<GFS,Z> ZDGF;
    ZDGF zdgf(gfs,z);
    typedef Dune::PDELab::VTKGridFunctionAdapter<ZDGF> VTKF;
    vtkwriter.addVertexData(
        std::shared_ptr<VTKF>(new VTKF(zdgf,"fesol")));
    typedef Dune::PDELab::DiscreteGridFunction<P0GFS,Z0> Z0DGF;
    Z0DGF z0dgf(p0gfs,z0);
    typedef Dune::PDELab::VTKGridFunctionAdapter<Z0DGF> VTKF0;
    vtkwriter.addCellData(
        std::shared_ptr<VTKF0>(new VTKF0(z0dgf,"error2")));
    vtkwriter.write(ptree.get("output.filename",
          (std::string)"output")+iter, Dune::VTK::appendedraw);
```

A VTK output of the finite element solution and the squared error is created. This allows careful inspection of the estimated error and how it changes in the course of the adaptive refinement.

```
    // error control
    auto tol = ptree.get("fem.tol",(double)0.0);
    if (estimated_error<=tol) break;
    if (i==steps-1) break;
```

If the norm of the estimated error is below the desired tolerance, then the finite element solution can be accepted. Additional refinement is unnecessary and the loop is exited. This also happens if the last iteration is reached, since the modified mesh would not be used for further computations.

```
    // mark elements for refinement
    std::cout << "mark␣elements" << std::endl;
    auto fraction = ptree.get("fem.fraction",(double)0.5);
    RF eta_refine,eta_coarsen;
```

```
    Dune::PDELab::error_fraction(
        z0,fraction,0.0,eta_refine,eta_coarsen);
    if (fraction>=1.0 || i<uniformlevel) eta_refine=0.0;
    Dune::PDELab::mark_grid(grid,z0,eta_refine,0.0,2);
```

If the norm of the error estimate is still too large, then it has to be decided which of the elements should be refined and which should be kept. The fraction of elements that should be refined is read from the ini-file. The function `error_fraction` is called to translate this fraction of elements into a threshold `eta_refine` of the value of the local indicator that separates the elements that should be refined from those that should remain the same. It is also possible to request a fraction of elements that should be coarsened, but this value is zero here for simplicity. If the chosen fraction contains all elements or the loop is still in an iteration that should refine globally, then this threshold is set to zero. Afterwards the function `mark_grid` is called and marks all elements which have a value in `z0` that is larger than `eta_refine` to request their refinement.

```
    // do refinement
    std::cout << "adapt␣grid␣and␣solution" << std::endl;
    Dune::PDELab::adapt_grid(grid,gfs,z,2*(degree+1));
```

This is the central line that calls `adapt_grid` and modifies the mesh. In contrast to most other functions in PDELab, this function requires the grid as an argument, and not a `GridView`. Grid views are a read only concept, and `adapt_grid` has to modify its argument. The marks on the elements are passed to the grid manager, which decides how it can best achieve the requested refinement. Depending on the capabilities and restrictions of the grid manager, the resulting mesh may not be exactly as requested. Additional refinement may be necessary to keep the mesh conforming, for example. However, the level that is requested is a lower bound for the resulting level at each point in the domain, i.e. elements are only coarsened if requested and any marked element is refined. The function `adapt_grid` also receives the grid function space `gfs` and the solution `z` as arguments. The grid function space is updated to reflect the changes of the underlying mesh, and the solution is transferred to the new grid function space, by default through local L^2 projection onto the elements of the new mesh. The last argument specifies the order of the quadrature rule that is used in this process.

```
    // recompute constraints
    std::cout << "constraints␣and␣stuff" << std::endl;
    Dune::PDELab::constraints(b,gfs,cc);

    // write correct boundary conditions in new vector
    Z znew(gfs);
    Dune::PDELab::interpolate(g,gfs,znew);

    // copy Dirichlet boundary to interpolated solution
    Dune::PDELab::copy_constrained_dofs(cc,znew,z);
  }
```

The function `adapt_grid` has updated the grid function space and transferred the solution to the new mesh, but the Dirichlet boundary condition requires manual

intervention. The function `constraints` is called again, which updates the list of constrained degrees of freedom on the Dirichlet boundary. Then the corresponding values of the Dirichlet boundary condition are computed and copied onto the constrained degrees of freedom of `z`. This guarantees that the function `g` of Dirichlet boundary values is always represented in the most accurate way possible on the mesh. With these last steps the iteration is finished, and the next one begins.

4.4 Classes `Problem` and `NonlinearPoissonFEM`

These two classes are nearly identical to those of tutorial 01 and are therefore skipped. The only differences are the definitions of the right hand side and Dirichlet boundary condition.

```
//! right hand side
template<typename E, typename X>
Number f (const E& e, const X& x) const
{
  return 0.0;
}
```

The example application of the tutorial solves the Laplace equation for a known reference solution, and therefore we set $f = 0$.

```
//! Dirichlet extension
template<typename E, typename X>
Number g (const E& e, const X& xlocal) const
{
  auto x = e.geometry().global(xlocal);
  double theta = std::atan2(x[1],x[0]);
  if(theta < 0.0) theta += 2*M_PI;
  auto r = x.two_norm();
  return pow(r,2.0/3.0)*std::sin(theta*2.0/3.0);
}
```

The value of the Dirichlet boundary condition is the restriction of the reference solution to the boundary, which is given by

$$u(r, \theta) = r^{2/3} \cdot \sin\left(\frac{2}{3}\theta\right)$$

in polar coordinates.

4.5 Local Operator `NonlinearPoissonFEMEstimator`

The class `NonlinearPoissonFEM` implements the element-wise computations of the error estimate introduced in Section 3. The computation of the error estimate is then implemented as a grid operator that returns the square of the estimated error as its residual. The contributions R_T for the elements are provided by an `alpha_volume` term, while the contributions R_F for the faces are provided by `alpha_skeleton` and `alpha_boundary` terms.

The definition of class `NonlinearPoissonFEMEstimator` starts as follows:

```
 */
template<typename Param, typename FEM>
class NonlinearPoissonFEMEstimator
```

The class is parametrized by a parameter class and a finite element map, just as the `NonlinearPoissonFEM` class. The parameter class is the same in both cases and provides access to the problem definition. The finite element map is expected to provide element-wise constant functions that match the geometry of the mesh elements, since the calculated error estimate contains one value per element. The class does not contain methods for the Jacobian and matrix-free Jacobian evaluation, as only the residual has to be assembled for the error estimator. Therefore, only the `LocalOperatorDefaultFlags` are inherited.

Just as the class `NonlinearPoissonFEM`, the local operator contains three private data members, a cache for evaluation of the basis functions on the reference element:

```
{
  // a cache for local basis evaluations
  typedef typename FEM::Traits::FiniteElementType
    ::Traits::LocalBasisType LocalBasis;
```

a reference to the parameter object:

```
  Dune::PDELab::LocalBasisCache<LocalBasis> cache;
```

and an integer value controlling the order of the formulas used for numerical quadrature:

```
  Param& param; // parameter functions
```

The class has an additional private method `diameter`:

```

  // a function to compute the diameter of an entity
  template<class GEO>
  typename GEO::ctype diameter (const GEO& geo) const
  {
    typedef typename GEO::ctype DF;
    DF hmax = -1.0E00;
    for (int i=0; i<geo.corners(); i++)
      {
        auto xi = geo.corner(i);
        for (int j=i+1; j<geo.corners(); j++)
          {
            auto xj = geo.corner(j);
            xj -= xi;
            hmax = std::max(hmax,xj.two_norm());
          }
      }
    return hmax;
```

This function iterates over all pairs of corners of a given entity and defines the maximum distance among these pairs as its diameter. This information is required

to scale the element contributions and face contributions of the error estimate with the local mesh width.

The public part of the class again starts with the definition of the flags controlling the generic assembly process. The `doPatternVolume` and `doPatternSkeleton` flags are both `false` to indicate that the sparsity pattern of the Jacobian can be skipped for this local operator:

```
public:
  // pattern assembly flags
  enum { doPatternVolume = false };
```

The residual assembly flags indicate that in this local operator we will provide the methods `lambda_volume`, `lambda_boundary` and `alpha_volume`:

```

  // residual assembly flags
  enum { doAlphaVolume  = true };
  enum { doAlphaSkeleton  = true };
```

Next comes the constructor taking as an argument a reference to a parameter object and the optional increment of the quadrature order:

```

  //! constructor: pass parameter object
  NonlinearPoissonFEMEstimator (Param& param_, int incrementorder_=0)
    : param(param_), incrementorder(incrementorder_)
```

Method `alpha_volume`

This method assembles the error estimate contribution R_T for a single element T. Its interface is

```

  // volume integral depending on test and ansatz functions
  template<typename EG, typename LFSU, typename X,
    typename LFSV, typename R>
  void alpha_volume (const EG& eg, const LFSU& lfsu,
```

This method is similar to the `alpha_volume` method of `NonlinearPoissonFEM`, but assembles a different function. The method starts by extracting the floating point type to be used for computations:

```
  {
    // types & dimension
```

Then a quadrature rule is selected

```

    // select quadrature rule
    auto geo = eg.geometry();
    const int order =
      incrementorder+2*lfsu.finiteElement().localBasis().order();
    auto rule = Dune::PDELab::quadratureRule(geo,order);
```

a variable `sum` to collect the contributions is created, and the quadrature loop is started

```
    // loop over quadrature points
    RF sum(0.0);
    for (const auto& ip : rule)
      {
```

Within the quadrature loop the basis functions are evaluated

```
        // evaluate basis functions
        auto& phihat = cache.evaluateFunction(
```

and the value of u_h at the quadrature point is computed.

```
        // evaluate u
        RF u=0.0;
```

We neglect contributions containing the gradient of u_h, since they would require second-order derivatives of the basis functions. This means we do not need to evaluate the gradients of the basis functions.

The right hand side f and the nonlinear term $q(u_h)$ are evaluated:

```
        // evaluate reaction term
        auto q = param.q(u);

        // evaluate right hand side parameter function
        auto f = param.f(eg.entity(),ip.position());
```

and the local error estimate for this quadrature point is calculated:

```
        // integrate f^2
        RF factor =
          ip.weight() * geo.integrationElement(ip.position());
        sum += (f-q)*(f-q)*factor;
```

After summing up all local contributions, we are in the position to finally compute the estimate on the element by multiplying with h_T^2:

```
    // accumulate cell indicator
    auto h_T = diameter(eg.geometry());
```

Method `alpha_skeleton`

This method has a structure that is very similar to the skeleton terms of the finite volume method from tutorial 02. It implements the error estimate contributions R_F for the interior faces of the mesh and has the following interface:

```
  // skeleton integral depending on test and ansatz functions
  // each face is only visited ONCE!
```

```
template<typename IG, typename LFSU, typename X,
  typename LFSV, typename R>
void alpha_skeleton (const IG& ig,
    const LFSU& lfsu_i, const X& x_i, const LFSV& lfsv_i,
    const LFSU& lfsu_o, const X& x_o, const LFSV& lfsv_o,
```

As in tutorial 02, the arguments comprise an intersection, local trial function and local test space for both elements adjacent to the intersection and containers for the local residual contributions in both elements. The subscripts `_i` and `_o` correspond to "inside" and "outside". W.r.t. our notation above "inside" corresponds to "-" and "outside" corresponds to "+".

As already noted in the context of the finite volume method of tutorial 02, the `alpha_skeleton` method needs to assemble contributions for *both* elements next to the intersection.

It starts by extracting the geometries of the intersection relative to the two elements adjacent to the intersection

```
{
  // geometries in local coordinates of the elements
  auto insidegeo = ig.geometryInInside();
```

the two elements themselves

```
  // inside and outside cells
  auto cell_inside = ig.inside();
```

and their geometries

```
  // geometries from local to global in elements
  auto geo_i = cell_inside.geometry();
```

Then the dimension of the intersection is extracted and a quadrature rule is created:

```
  // dimensions
  const int dim = IG::Entity::dimension;

  // select quadrature rule
  auto globalgeo = ig.geometry();
  const int order =
    incrementorder+2*lfsu_i.finiteElement().localBasis().order();
```

The quadrature rule integrates over the intersection and collects contributions as the rule in `alpha_skeleton` did:

```
  // loop over quadrature points and integrate normal flux
  typedef decltype(makeZeroBasisFieldValue(lfsu_i)) RF;
  RF sum(0.0);
  for (const auto& ip : rule)
```

The coordinates on the intersection are translated to coordinates in the two elements to make the evaluation of u_h and its gradient possible, and the unit vector on the intersection pointing from T^- to T^+ is obtained.

```
    {
      // position of quadrature point in local coordinates of elements
      auto iplocal_i = insidegeo.global(ip.position());
      auto iplocal_o = outsidegeo.global(ip.position());

      // unit outer normal direction
```

The gradients of the basis functions are evaluated on the reference element, mapped onto the actual element, and used to calculate $(\nabla u_h) \cdot \nu$ for the element T_F^-. Afterwards the same steps are performed for T_F^+, which is not repeated here.

```
      // gradient in normal direction in self
      auto& gradphihat_i = cache.evaluateJacobian(
          iplocal_i,lfsu_i.finiteElement().localBasis());
      const auto S_i = geo_i.jacobianInverseTransposed(iplocal_i);
      RF gradun_i = 0.0;
      for (size_t i=0; i<lfsu_i.size(); i++)
        {
          Dune::FieldVector<RF,dim> v;
          S_i.mv(gradphihat_i[i][0],v);
          gradun_i += x_i(lfsu_i,i)*(v*n_F);
```

The face residual R_F is calculated as the difference between the two directional derivatives of u_h across the face, and its square is added to the sum.

```
      // integrate
      RF factor =
        ip.weight()*globalgeo.integrationElement(ip.position());
      RF jump = gradun_i-gradun_o;
```

After summing up all local contributions, the result is multiplied with $\frac{1}{2}h_T$ and added to the estimate on the element:

```
    // accumulate indicator
    auto h_T = diameter(globalgeo);
    r_i.accumulate(lfsv_i,0,0.5*h_T*sum);
```

Method `alpha_boundary`

The method `alpha_boundary` is largely identical with `alpha_skeleton` and therefore isn't repeated here. All calculations concerning T_F^+ are dropped, and instead the Neumann boundary condition value j is used in the calculation of the jump:

```
      // Neumann boundary condition value
```

```
        auto j = param.j(ig.intersection(),ip.position());

        // integrate
        RF factor =
          ip.weight()*globalgeo.integrationElement(ip.position());
        RF jump = gradun_i+j;
```

Additionally, the final accumulation scales with h_T, and not with $\frac{1}{2}h_T$ as the `alpha_skeleton` method.

```
    // accumulate indicator
    auto h_T = diameter(globalgeo);
```

4.6 Running the Example

Now we can run the tutorial and look at the results. The program can be run by typing

```
./tutorial05
```

on the command line. It then produces some output on the console and VTK files with the extension `.vtu`.

First, the program reports that it is run on one processor:

```
Parallel code run on 1 process(es)
```

Then the mesh file is read and some statistics about it are reported:

```
Reading 2d Gmsh grid...
version 2.2 Gmsh file detected
file contains 375 nodes
file contains 754 elements
number of real vertices = 375
number of boundary elements = 74
number of elements = 674
```

Now an instance of a DUNE grid is created, the initial setup is performed, and the refinement loop is started. The problem is solved on the initial mesh:

```
Iteration: 0    highest level in grid: 0
constrained dofs=74 of 375
  Initial defect:   2.9656e-02
  Newton iteration  1.  New defect:   2.9206e-11.  Reduction
    (this): 9.8481e-10.  Reduction (total):   9.8481e-10
```

Due to the missing reaction term q the problem is actually linear, and therefore it is solved after just one Newton iteration. The program runs the error estimator and reports an estimate for the error $\|u - u_h\|$ using the L^2 norm:

```
Estimated error in step 0 is 0.285625
```

Then the program states that it performs the remaining tasks of the first iteration:

```
VTK output
mark elements
adapt grid and solution
Updating entity set
constraints and stuff
```

The current solution u_h and the squared local error estimates are written out to a VTK file. Then the elements are marked for refinement based on the chosen fraction of elements and error estimate, the grid manager modifies the grid, and the solution u_h and the grid function space are updated to reflect these changes. This includes a reevaluation of the constraints and the values of the Dirichlet boundary condition. Then the solution vector can be used as an initial guess for the next iteration.

The loop runs for several iterations and produces similar output in each step. The reported error estimates are:

```
Estimated error in step 0 is 0.285625
Estimated error in step 1 is 0.209272
Estimated error in step 2 is 0.145672
Estimated error in step 3 is 0.09893
Estimated error in step 4 is 0.0672581
Estimated error in step 5 is 0.0446733
```

The program also reports the number of elements after each refinement:

```
constrained dofs=74 of 375
constrained dofs=78 of 419
constrained dofs=90 of 647
constrained dofs=117 of 1409
constrained dofs=171 of 2957
constrained dofs=244 of 6804
```

These numbers can be collected in a table containing the refinement level, the estimated error, the experimental order of convergence (EOC) for the error, the number of elements, and the number of elements a globally refined mesh would have:

level	$\gamma(u_h)$	$\#T$	$\#T_{\text{global}}$
0	0.286	375	375
1	0.209	419	1.5e3
2	0.146	647	6.0e3
3	0.099	1409	2.4e4
4	0.067	2957	9.6e4
5	0.045	6804	3.8e5

Figure 1 shows the final mesh and the corresponding solution u_h. The discretization error is almost completely restricted to the elements that touch the reentrant corner in the point $(0, 0)$ and therefore hard to visualize.

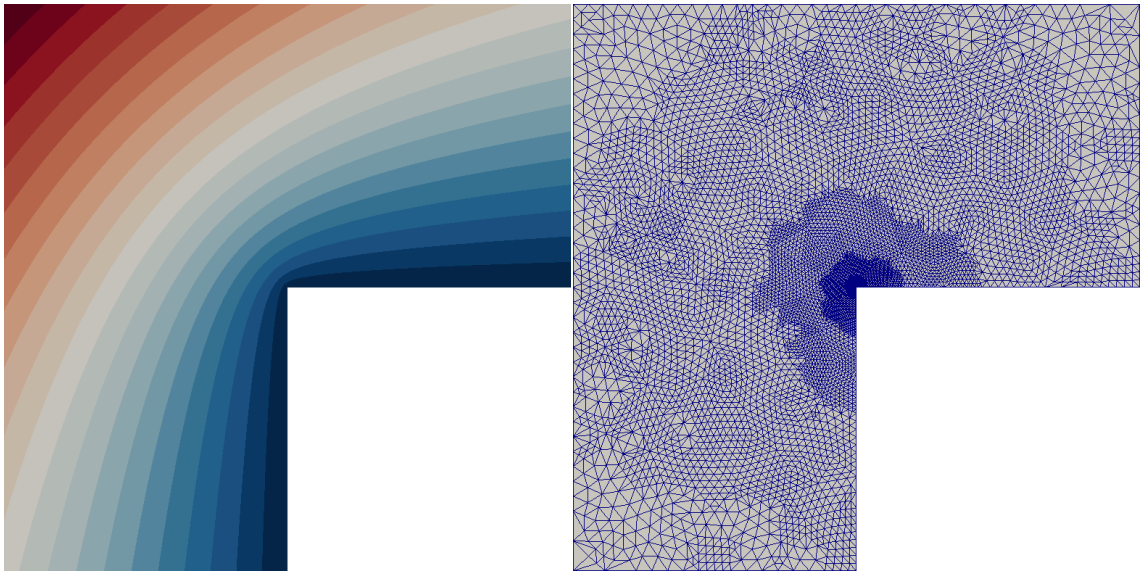

Figure 1: Solution on the finest mesh and underlying elements. The colormap for the solution uses discrete intervals to make the large gradients in the reentrant corner visible. Refinement has mainly occurred in this area, as can be seen on the right.

5 Outlook

The interested reader can proceed in different directions from here. The more obvious things are:

- Change the fraction of elements that is refined in each iteration and compare the resulting meshes, error estimates and convergence behavior.
- Switch to a different grid manager and examine the resulting meshes for similarities and differences.
- Test the reliability of the error estimate when a nonlinear function is used for q, or use a different right hand side f.

References

[1] M. Ainsworth and J. T. Oden. *A posteriori error estimation in finite element analysis*. Wiley, 2000.

[2] K. Eriksson, D. Estep, P. Hansbo, and C. Johnson. *Computational Differential Equations*. Cambridge University Press, 1996. `http://www.csc.kth.se/~jjan/private/cde.pdf`.