DUNE PDELab Tutorial 01
Conforming Finite Element Method for a Nonlinear Poisson Equation

DUNE/PDELab Team

February 5, 2021

Contents

1 Introduction

In this tutorial we extend tutorial 00 in the following ways:

1) Solve a nonlinear stationary partial differential equation (PDE).

2) Use conforming finite element spaces of arbitrary order.

3) Use different types of (conforming) meshes (simplicial, cubed and mixed).

4) Use multiple types of boundary conditions.

Combined with the fact that the implementation works in any dimension (note: it is not claimed to be efficient in high dimension $d > 3$) this comprises already a relatively large space of different methods, so the example illustrates the flexibility of PDELab. Moreover, the finite element method developed in this tutorial will serve as a building block for instationary problems, adaptive mesh refinement and parallel solution in subsequent tutorials.

Depends On

This tutorial depends on tutorial 00 which discusses piecewise linear elements on simplicial elements. It is assumed that you have worked through tutorial 00 before.

2 Problem Formulation

Here we consider the following nonlinear Poisson equation with Dirichlet and Neumann boundary conditions:

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

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

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

$\Omega \subset \mathbb{R}^d$ 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:

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

with the continuous residual form

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

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.

3 Finite Element Method

The finite element method [7, 2, 5, 3, 1, 6, 4] replaces the function spaces U and V by finite dimensional approximations defined on a finite element mesh. Before describing exactly how these spaces are constructed let us explore the consequences of this.

3.1 Algebraic Problem

Any finite-dimensional function space is spanned by a basis. So, assume that

$$U_h = \text{span}\{\phi_1, \ldots, \phi_n\}, \quad V_h = \text{span}\{\psi_1, \ldots, \psi_m\}$$

are corresponding sets of basis functions for U_h and V_h. Expanding the solution $u_h = \sum_{j=1}^{n}(z)_j\phi_j$ in the basis and hereby introducing the coefficient vector $z \in \mathbb{R}^n$ we can reformulate the problem as

$$\begin{aligned}
&\text{Find } u_h \in U_h \text{ s.t.:} && r(u_h, v) = 0 && \forall v \in V_h \\
&\Leftrightarrow && r\left(\sum_{j=1}^{n}(z)_j\phi_j, \psi_i\right) = 0 && \forall i = 1, \ldots, m \\
&\Leftrightarrow && R(z) = 0, &&
\end{aligned}$$

where $R : \mathbb{R}^n \to \mathbb{R}^m$ given by $R_i(z) = r_h\left(\sum_{j=1}^{n}(z)_j\phi_j, \psi_i\right)$ is a nonlinear, vector-valued function.

The solution of the nonlinear algebraic equation $R(z) = 0$ is typically computed in an iterative fashion using e.g. a fixed-point iteration of the form

$$z^{(k+1)} = z^{(k)} - \lambda^k W(z^{(k)})R(z^{(k)}). \tag{3}$$

Here λ^k is a damping factor and $W(z^{(k)})$ is a preconditioner matrix, e.g. in Newton's method (see e.g. [1]) one has

$$W(z^{(k)}) = (J(z^{(k)}))^{-1} \quad \text{where } (J(z^{(k)}))_{i,j} = \frac{\partial R_i}{\partial z_j}(z^{(k)})$$

(we now assumed that $n = m$ and that the Jacobian $J(z^{(k)})$ is invertible). Newton's method requires the solution of the linear system $J(z^{(k)})w = R(z^{(k)})$ in each step which could be done using either direct or iterative methods. The implementation of Newton's method requires the following algorithmic building blocks:

i) residual evaluation $R(z)$,

ii) Jacobian evaluation $J(z)$ (or an approximation of it),

iii) matrix-free Jacobian application $J(z)w$ (or an approximation).

Only one of the methods ii) and iii) is required depending on the chosen solution procedure.

A Note on Matrix-free Evaluation

The matrix-free multiplication of the Jacobian $J(z)$ with a vector w is with the definitions above:

$$(J(z)w)_i = \sum_{j=1}^{n} (J(z))_{i,j}(w)_j = \sum_{j=1}^{n} \frac{\partial}{\partial z_j} r_h \left(\sum_{l=1}^{n} (z)_l \phi_l, \psi_i \right) (w)_j.$$

At this point one may exploit the local support of basis functions in order to compute only the partial derivatives that are nonzero.

In the linear case, for comparison, one has $r_h(u,v) = a_h(u,v) - l_h(v)$ where a_h is a bilinear form and l_h is a linear form. Then, the application of the Jacobian can be simplified as

$$\begin{aligned}
(J(z)w)_i &= \sum_{j=1}^{n} \frac{\partial}{\partial z_j} r_h \left(\sum_{l=1}^{n} (z)_l \phi_l, \psi_i \right) (w)_j \\
&= \sum_{j=1}^{n} \frac{\partial}{\partial z_j} \left(a_h \left(\sum_{l=1}^{n} (z)_l \phi_l, \psi_i \right) - l_h(\psi_i) \right) (w)_j \\
&= \sum_{j=1}^{n} \frac{\partial}{\partial z_j} \left(\sum_{l=1}^{n} (z)_l a_h(\phi_l, \psi_i) \right) (w)_j \\
&= \sum_{j=1}^{n} a_h(\phi_j, \psi_i)(w)_j = a_h \left(\sum_{j=1}^{n} (w)_j \phi_j, \psi_i \right) = (Aw)_i
\end{aligned}$$

where $(A)_{i,j} = a_h(\phi_j, \psi_i)$ is the stiffness matrix which is independent of z. For this reason there exist two different functions for matrix-free operator application, one for the linear case providing only the argument w and one for the nonlinear case providing two arguments z and w.

Note also that it is advantageous to separate in the implementation of the residual form the part that depends on trial and test functions, and which consequently contributes to the Jacobian, and the part that only depends on the test functions and which does not contribute to the Jacobian.

3.2 Finite Element Space

The detailed construction of the basis functions ϕ_j involves the finite element mesh. Wrapping up the notation from tutorial 00, a finite element mesh consists of

i) A set of vertices $\mathcal{X}_h = \{x_1, \ldots, x_N\}$ and elements $\mathcal{T}_h = \{T_1, \ldots, T_M\}$. Elements are closed and connected sets of points with non-intersecting interior partitioning the domain Ω.

ii) A partitioning of the vertex index set $\mathcal{I}_h = \{1, \ldots, N\}$ into indices of interior and boundary vertices

$$\mathcal{I}_h = \mathcal{I}_h^{int} \cup \mathcal{I}_h^{\partial\Omega}, \quad \mathcal{I}_h^{int} = \{i \in \mathcal{I}_h \,:\, x_i \in \Omega\}, \quad \mathcal{I}_h^{\partial\Omega} = \{i \in \mathcal{I}_h \,:\, x_i \in \partial\Omega\}.$$

iii) For every element $T \in \mathcal{T}_h$ a local-to-global map

$$g_T : \{0, \ldots, n_T - 1\} \to \mathcal{I}_h$$

associating a local number of a corner of element T with a global vertex number. n_T is the number of corners of element T.

iv) For every element $T \in \mathcal{T}_h$ an element transformation map

$$\mu_T : \hat{T} \to T$$

mapping the corresponding reference element to T. The element transformation map need not be affine but is assumed to be sufficiently differentiable with invertible Jacobian as well as consistent in the sense $\forall i \in \{0, \ldots, n_T - 1\} : \mu_T(\hat{x}_i) = x_{g_T(i)}$.

The conforming finite element space of degree k in dimension d on the mesh $\mathcal{T}_h$ is given by

$$V_h^{k,d}(\mathcal{T}_h) = \left\{ v \in C^0(\overline{\Omega}) \, : \, \forall T \in \mathcal{T}_h : v|_T = \mu_T \circ p_T \wedge p_T \in \mathbb{P}_T^{k,d} \right\} \tag{4}$$

with the appropriate space of multivariate polynomials of degree k in dimension d depending on the type of element T:

$$\mathbb{P}_T^{k,d} = \left\{ \begin{array}{ll} \left\{ p \, : \, p(x_1, \ldots, x_d) = \sum\limits_{0 \le \|\alpha\|_1 \le k} c_\alpha x_1^{\alpha_1} \cdot \ldots \cdot x_d^{\alpha_d} \right\} & \hat{T} = \hat{S} \text{ (simplex)}, \\ \left\{ p \, : \, p(x_1, \ldots, x_d) = \sum\limits_{0 \le \|\alpha\|_\infty \le k} c_\alpha x_1^{\alpha_1} \cdot \ldots \cdot x_d^{\alpha_d} \right\} & \hat{T} = \hat{C} \text{ (cube)}. \end{array} \right. \tag{5}$$

Note that in dimension 1 there is no difference between cube and simplex. In dimension 2 triangular and quadrilateral elements may be mixed. In dimension 3, however, tetrahedral and hexahedral elements may not be mixed without introducing additional elements such as prisms. The dimension of $\mathbb{P}_T^{k,d}$ is

$$n_{\hat{C}}^{k,d} = (k+1)^d$$

in the case of a cube reference element and

$$n_{\hat{S}}^{k,d} = \left\{ \begin{array}{ll} 1 & k = 0 \vee d = 0 \\ \sum_{i=0}^{k} n_{\hat{S}}^{i,d-1} & \text{else} \end{array} \right.$$

in the case of a simplex reference element.

Local Lagrange Basis

(4) defines the finite element space without reference to a basis. For the implementation in the computer a basis is needed. We now generalize the construction of the Lagrange basis functions to the general space $V_h^{k,d}(\mathcal{T}_h)$. To that end, the reference element $\hat{T}$ is equipped with Lagrange points

$$L_{\hat{T}} = \left\{ \hat{x}_0^{\hat{T}}, \ldots, \hat{x}_{n_{\hat{T}}^{k,d}-1}^{\hat{T}} \right\}$$

and Lagrange polynomials

$$P_{\hat{T}} = \left\{ p_0^{\hat{T}}, \ldots, p_{n_{\hat{T}}^{k,d}-1}^{\hat{T}} \right\}$$

such that

$$p_i^{\hat{T}}(\hat{x}_j^{\hat{T}}) = \delta_{i,j}.$$

Global continuity is then ensured by carefully placing $n_{\hat{T}}^{k,d-c}$ of these points on each subentity of codimension c in the reference element.

Global Lagrange Basis

The local-to-global map g_T is extended to map indices of local basis functions to indices of global basis functions of the finite element space. Let

$$g_T : \{0, \ldots, n_{\hat{T}}^{k,d} - 1\} \to \mathcal{I}\left(V_h^{k,d}(\mathcal{T}_h)\right) = \left\{0, \ldots, \dim V_h^{k,d}(\mathcal{T}_h) - 1\right\}$$

be this extension and

$$C(i) = \{(T, m) \in \mathcal{T}_h \times \mathbb{N} \,:\, g_T(m) = i\}$$

its inversion. Then

i) $C(i)$ is nonempty for all $i \in \mathcal{I}\left(V_h^{k,d}(\mathcal{T}_h)\right)$ and

ii) for all $(T, m), (T', m') \in C(i)$ it holds $\mu_T(x_m^{\hat{T}}) = \mu_{T'}(x_{m'}^{\hat{T}'})$.

The global Lagrange basis functions spanning $V_h^{k,d}(\mathcal{T}_h)$ are then defined by

$$\phi_i(x) = \begin{cases} p_m^{\hat{T}}(\mu_T^{-1}(x)) & x \in T \wedge (T, m) \in C(i) \\ 0 & \text{else} \end{cases}, \quad i \in \mathcal{I}\left(V_h^{k,d}(\mathcal{T}_h)\right).$$

Each global basis function ϕ_i corresponds to a Lagrange point x_i from the ordered set

$$\mathcal{X}_h^{k,d} = \left\{ x_i \in \overline{\Omega} \,:\, x_i = \mu_T(\hat{x}_m^{\hat{T}}) \wedge (T, m) \in C(i) \right\}.$$

Note that for degree $k = 1$ one has $\mathcal{I}\left(V_h^{1,d}(\mathcal{T}_h)\right) = \mathcal{I}_h$, i.e. one basis function is associated with each vertex of the mesh.

3.3 Incorporation of Dirichlet Boundary Conditions

The standard way to handle Dirichlet boundary conditions in the conforming finite element method is to incorporate them directly into the function space. To do that define the indices of Lagrange points on the Dirichlet boundary

$$\mathcal{I}^D\left(V_h^{k,d}(\mathcal{T}_h)\right) = \left\{ i \in \mathcal{I}\left(V_h^{k,d}(\mathcal{T}_h)\right) \,:\, x_i \in \mathcal{X}_h^{k,d} \wedge x_i \in \Gamma_D \right\}.$$

For the test space one defines the space of finite element functions that are zero on the Dirichlet boundary:

$$V_{h,0}^{k,d}(\mathcal{T}_h) = \left\{ v \in V_h^{k,d}(\mathcal{T}_h) \,:\, v(x_i) = 0 \quad \forall i \in \mathcal{I}^D\left(V_h^{k,d}(\mathcal{T}_h)\right) \right\}$$

For the trial space set

$$u_{h,g} = \sum_{i \in \mathcal{I}^D\left(V_h^{k,d}(\mathcal{T}_h)\right)} g(x_i)\phi_i$$

with the Dirichlet boundary condition function g. In fact one may want to use

$$u_{h,g} = \sum_{i \in \mathcal{I}\left(V_h^{k,d}(\mathcal{T}_h)\right)} u_g(x_i)\phi_i \qquad \text{with } u_g(x_i) = g(x_i) \text{ for all } i \in \mathcal{I}^D\left(V_h^{k,d}(\mathcal{T}_h)\right)$$

which incorporates the Dirichlet boundary conditions on Γ_D and provides an initial guess for the nonlinear iterative solver in the interior of the domain. Then the trial space is

$$U_h^{k,d}(\mathcal{T}_h) = \left\{ u \in V_h^{k,d}(\mathcal{T}_h) \,:\, u = u_{h,g} + w \wedge w \in V_{h,0}^{k,d}(\mathcal{T}_h) \right\}.$$

Finally, the finite element problem in its precise form reads

$$\text{Find } u \in U_h^{k,d}(\mathcal{T}_h) \text{ s.t.:} \quad r^{\text{NLP}}(u,v) = 0 \quad \forall v \in V_{h,0}^{k,d}(\mathcal{T}_h). \tag{6}$$

General Constraints

PDELab provides a more general approach to construct subspaces of finite element spaces. Given a finite-dimensional space $U_h = \operatorname{span}\{\phi_j \,:\, j \in J_h = \{1,\dots,n\}\}$ a subspace $\tilde{U}_h$ is constructed by

i) selecting a subset of indices $\tilde{J}_h \subset J_h$

ii) and setting $\tilde{U}_h = \operatorname{span}\left\{\tilde{\phi}_j \,:\, j \in \tilde{J}_h\right\}$, where the new basis functions have the form

$$\tilde{\phi}_j = \phi_j + \sum_{l \in J_h \setminus \tilde{J}_h} (B)_{j,l}\phi_l \quad \forall j \in \tilde{J}_h.$$

Thus, any subspace of U_h is characterized by $C = (\tilde{J}_h, B)$. This abstractions allows to represent Dirichlet conditions ($J_h \setminus \tilde{J}_h$ are the indices of the Dirichlet nodes and $B = 0$), hanging nodes ($J_h \setminus \tilde{J}_h$ are the indices of hanging nodes and B represents the interpolation conditions) or even rigid body modes.

3.4 Element-wise Computations

We now turn to how the residual can be evaluated in practice. The residual form (2) can be readily decomposed into elementwise contributions:

$$r^{\text{NLP}}(u,v) = \sum_{T \in \mathcal{T}_h} \alpha_T^V(u,v) + \sum_{T \in \mathcal{T}_h} \lambda_T^V(v) + \sum_{F \in \mathcal{F}_h^{\partial\Omega}} \lambda_F^B(v)$$

with

$$\alpha_T^V(u,v) = \int_T \nabla u \cdot \nabla v + q(u)v \, dx, \quad \lambda_T^V(v) = -\int_T fv \, dx, \quad \lambda_F^B(v) = \int_{F \cap \Gamma_N} jv \, ds.$$

Here $\mathcal{F}_h^{\partial\Omega}$ is the set of intersections of elements with the domain boundary $\partial\Omega$. The element-wise computations can be classified on the one hand as volume integrals (superscript V), boundary integrals (superscript B) and skeleton integrals (superscript S, to be shown later) and on the other hand as integrals depending on trial and test functions (α-terms) and integrals depending only on test functions (λ-terms). Here we need three of these six possible combinations.

The three terms can now be evaluated using the techniques introduced in tutorial 00 with the small extension that for general maps μ_T we have

$$\nabla w(\mu_T(\hat{x})) = J_{\mu_T}^{-1}(\hat{x})\hat{\nabla}\hat{w}(\hat{x})$$

with $J_{\mu_T}(\hat{x})$ the Jacobian of μ_T at point $\hat{x}$.

λ Volume Term

For any $(T, m) \in C(i)$ we obtain

$$\lambda_T^V(\phi_i) = -\int_T f\phi_i \, dx = -\int_{\hat{T}} f(\mu_T(\hat{x}))p_m^{\hat{T}}(\hat{x})|\mathrm{det}J_{\mu_T}(\hat{x})| \, d\hat{x}.$$

This integral on the reference element is then computed by employing numerical integration of appropriate order. The evaluation for all test functions with support on element T may be collected in a vector

$$(\mathcal{L}_T^V)_m = -\int_{\hat{T}} f(\mu_T(\hat{x}))p_m^{\hat{T}}(\hat{x})|\mathrm{det}J_{\mu_T}(\hat{x})| \, d\hat{x}.$$

λ Boundary Term

For $F \in \mathcal{F}_h^{\partial\Omega}$ with $F \cap \Gamma_N \neq \emptyset$ and $(T_F^-, m) \in C(i)$ we obtain

$$\lambda_T^B(\phi_i) = \int_F jv \, ds = \int_{\hat{F}} j(\mu_F(s))p_m^{\hat{T}}(\eta_F(s))\sqrt{|\mathrm{det}(J_{\mu_F}^T(s)J_{\mu_F}(s))|} \, ds$$

Because integration is over a face of codimension 1 now, two mappings are involved. The map μ_F maps the reference element $\hat{F}$ of F into global coordinates while the map η_F maps $\hat{F}$ into the reference element $\hat{T}$ of T. Also the integration element has to be redefined accordingly. Again, all contributions of the face F can be collected in a vector:

$$(\mathcal{L}_T^B)_m = \int_{\hat{F}} j(\mu_F(s))p_m^{\hat{T}}(\eta_F(s))\sqrt{|\mathrm{det}J_{\mu_T}^T(s)J_{\mu_T}(s)|} \, ds.$$

α Volume Term

For any $(T, m) \in C(i)$ we get

$$\begin{aligned}
\alpha_T^V(u_h, \phi_i) &= \int_T \nabla u \cdot \nabla\phi_i + q(u)\phi_i \, dx, = \int_T \sum_j (z)_j \left(\nabla\phi_j \cdot \nabla\phi_i\right) + q\left(\sum_j (z)_j\phi_j\right)\phi_i \, dx, \\
&= \int_{\hat{T}} \sum_n (z)_{g_T(n)}(J_{\mu_T}^{-1}(\hat{x})\hat{\nabla}p_n^{\hat{T}}(\hat{x})) \cdot (J_{\mu_T}^{-1}(\hat{x})\hat{\nabla}p_m^{\hat{T}}(\hat{x})) \\
&\qquad + q\left(\sum_n (z)_{g_T(n)}p_n^{\hat{T}}(\hat{x})\right)p_m^{\hat{T}}(\hat{x})|\mathrm{det}J_{\mu_T}(\hat{x})| \, d\hat{x}
\end{aligned}$$

Again contributions for all test functions can be collected in a vector

$$(\mathcal{R}_T^V(R_T z))_m = \sum_n (z)_{g_T(n)} \int_{\hat{T}} (J_{\mu_T}^{-1}(\hat{x})\hat{\nabla} p_n^{\hat{T}}(\hat{x})) \cdot (J_{\mu_T}^{-1}(\hat{x})\hat{\nabla} p_m^{\hat{T}}(\hat{x}))|\mathrm{det} J_{\mu_T}(\hat{x})|\, d\hat{x}$$
$$+ \int_{\hat{T}} q \left(\sum_n (z)_{g_T(n)} p_n^{\hat{T}}(\hat{x}) \right) p_m^{\hat{T}}(\hat{x})|\mathrm{det} J_{\mu_T}(\hat{x})|\, d\hat{x}$$

Putting it all together

Now with these definitions in place the evaluation of the algebraic residual is

$$R(z) = \sum_{T \in \mathcal{T}_h} R_T^T \mathcal{R}_T^V(R_T z) + \sum_{T \in \mathcal{T}_h} R_T^T \mathcal{L}_T^V + \sum_{F \in \mathcal{F}_h^{\partial\Omega} \cap \Gamma_N} R_T^T \mathcal{L}_F^B \tag{7}$$

The Jacobian of the residual is

$$(J(z))_{i,j} = \frac{\partial R_i}{\partial z_j}(z) = \sum_{(T,m,n):(T,m)\in C(i)\wedge(T,n)\in C(j)} \frac{\partial (\mathcal{R}_T^V)_m}{\partial z_n}(R_T z)$$

Note that:

a) Entries of the Jacobian can be computed element by element.

b) The derivative is independent of the λ-terms as they only depend on the test functions.

c) In the implementation below the Jacobian is computed numerically by finite differences. This can be achieved automatically by deriving from an additional base class.

4 Realization in PDELab

The structure of the code is very similar to that of tutorial 00. It consists of the following files:

1) The ini-file `tutorial01.ini` holds parameters read by various parts of the code which control the execution.

2) The main file `tutorial01.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 finally solves the problem.

4) File `nonlinearpoissonfem.hh` contains the class `NonlinearPoissonFEM` realizing a PDELab local operator implementing the conforming finite element method for arbitrary order and on arbitrary meshes.

5) File `problem.hh` contains a so-called parameter class which encapsulates the user-definable part of the PDE problem such as right hand side and boundary conditions.

6) Finally, the tutorial provides some mesh files.

4.1 Ini-File

The ini-file allows the user to set various parameters for the execution of the program. Here we skim briefly through the sections.

```
[grid]
dim=2         # set to 1 | 2 | 3
manager=yasp  # set to ug | alu | yasp
refinement=5  # be careful
```

The `grid` section allows to set the space dimension to 1, 2 or 3. In one dimension the grid manager `OneDGrid` is used. In dimension 2 or 3 `UGGrid` or `ALUGrid` using simplex grids or `YaspGrid` using cube grids can be selected. The refinement parameter can be used to refine the initial coarse mesh the specified number of times.

```
[grid.oned]
a=0.0
b=1.0
elements=2
```

The `grid.oned` subsection is active when the dimension of the grid is set to one. Then the domain is the interval from `a` to `b` and `elements` gives the number of elements used to subdivide the interval.

```
[grid.structured]
LX=1.0
LY=1.0
LZ=1.0
NX=2
NY=2
NZ=2
```

The `grid.structured` subsection is active when `yasp` is selected as a grid manager and allows to set the length of the domain in every direction (always starting at zero) and the number of elements to be used per direction.

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

The `grid.twod` subsection is active when `ug` or `alu` are selected as grid managers in two space dimensions. The gmsh-file with the given name is read as coarse grid.

```
[grid.threed]
filename=unitcube.msh
```

The `grid.threed` subsection is active when `ug` or `alu` are selected as grid managers in three space dimensions. The gmsh-file with the given name is read as coarse grid.

```
[fem]
degree=1 # oned: 1..4, ug|alu: 1..3, yasp: 1..2
```

The `fem` section provides the parameters for the finite element method. The only parameter in this tutorial is the polynomial degree for the finite element space. Note that different degrees are possible depending on the grid manager used.

```
[problem]
eta=2.0
```

The `problem` section provides parameters for the specific problem to be solved. The tutorial solves problem (1) with

$$q(u) = \eta u^2, \quad f(x) = -2d, \quad \Gamma_D = \partial\Omega, \quad g(x) = \|x\|^2.$$

The parameter η in the nonlinear term is read from the ini-file.

The behavior of the newton solver is also controlled by the ini-file.

```
[newton]
reassemble_treshhold = 0.0 # always reassemble J
verbosity = 3 # be verbose
reduction = 1e-10 # total reduction
min_linear_reduction = 1e-4 # min. red. in linear solve

[newton.terminate]
max_iterations = 25 # limit number of iterations

[newton.line_search]
line_search_max_iterations = 10 # limit linea search iterations
```

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.

```
[output]
filename=eta2
subsampling=1
```

4.2 Function `main`

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

Then there are several sections where `Dune::Grid` objects are instantiated and the `driver` function is called. Since the grid manager, the space dimension 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. As an example we just present the section for dimension 1 using `OneDGrid` here.

In one space dimension we start with reading the grid parameters from the ini-file:

```
    if (dim==1)
      {
        // read grid parameters from input file
        typedef Dune::OneDGrid::ctype DF;
        DF a = ptree.get<DF>("grid.oned.a");
        DF b = ptree.get<DF>("grid.oned.b");
        unsigned int N = ptree.get<int>("grid.oned.elements");
```

Then we create a `std::vector` with an equidistant subdivision of the given interval. Note that `OneDGrid` could handle arbitrary subdivisions.

```
        // create equidistant intervals
        std::vector<DF> intervals(N+1);
        for(unsigned int i=0; i<N+1; ++i)
          intervals[i] = a + DF(i)*(b-a)/DF(N);
```

Now an instance of `OneDGrid` can be created and refined uniformly:

```
        // Construct grid
        typedef Dune::OneDGrid Grid;
        Grid grid(intervals);
        grid.globalRefine(refinement);
```

Finally, for polynomial degrees one through four a finite element map of the appropriate order is created and the `driver` function is called. The `driver` function gets a grid view, the finite element map and the parameter tree as parameters and is covered in the next section:

```
        // call generic driver function
        typedef Dune::OneDGrid::LeafGridView GV;
        GV gv=grid.leafGridView();
        if (degree==1) {
          typedef Dune::PDELab::
            PkLocalFiniteElementMap<GV,DF,double,1> FEM;
          FEM fem(gv);
          driver(gv,fem,ptree);
        }
        if (degree==2) {
          typedef Dune::PDELab::
            PkLocalFiniteElementMap<GV,DF,double,2> FEM;
          FEM fem(gv);
          driver(gv,fem,ptree);
        }
        if (degree==3) {
          typedef Dune::PDELab::
            PkLocalFiniteElementMap<GV,DF,double,3> FEM;
          FEM fem(gv);
          driver(gv,fem,ptree);
        }
        if (degree==4) {
          typedef Dune::PDELab::
```

```
            PkLocalFiniteElementMap<GV,DF,double,4> FEM;
          FEM fem(gv);
          driver(gv,fem,ptree);
        }
```

4.3 Function `driver`

The function `driver` solves the problem on a given mesh with a particular finite element method given by its local basis functions on the reference element. It has the following interface:

```
template<typename GV, typename FEM>
void driver (const GV& gv, const FEM& fem,
             Dune::ParameterTree& ptree)
```

The `driver` function is very similar to the one in tutorial 00. Therefore we mainly focus on the differences. It starts by extracting the dimension of the grid and some important types:

```
  // dimension and important types
  const int dim = GV::dimension;
  typedef double RF;                    // type for computations
```

An important difference to tutorial 00 is that all parameters of the PDE to be solved are encapsulated in a separate class with a prescribed interface. This class is then given to the local operator as a template parameter. This makes sense here because problem (1) has five parameters: the functions q, f, g and j as well as the partitioning of the domain boundary in Dirichlet and Neumann part. This is a general pattern followed by many PDELab local operators.

The interface of the parameter class is defined by the implementor of the local operator and is not part of PDELab. As shown in tutorial 00 it is perfectly possible to have a local operator without a parameter class. The following code segment instantiates the problem class which is called `Problem` here (it is explained in detail below):

```
  // make PDE parameter class
  RF eta = ptree.get("problem.eta",(RF)1.0);
  Problem<RF> problem(eta);
```

Now there are two places where information from the PDE is used in PDELab. First of all we need to have an object that can be used to as an argument to `Dune::PDELab::interpolate` to initialize a vector which represents the initial guess and the Dirichlet boundary conditions. The class `Problem` defines a method which we need to use to define a class with the interface of `Dune::PDELab::GridFunction`. This is accomplished by the following code using C++-14 generic lambdas:

```
  auto glambda = [&](const auto& e, const auto& x)
    {return problem.g(e,x);};
  auto g = Dune::PDELab::
    makeGridFunctionFromCallable(gv,glambda);
```

Similarly, we need an object that can be passed to `Dune::PDELab::constraints` to fill a constraints container which is used to build a subspace of a function space. Again, the class `Problem` defines such a method which is extracted with a lambda function:

```
auto blambda = [&](const auto& i, const auto& x)
  {return problem.b(i,x);};
auto b = Dune::PDELab::
  makeBoundaryConditionFromCallable(gv,blambda);
```

The next step is to define the grid function space. This is exactly the same code as in tutorial 00 except that the finite element map is passed as an argument to the `driver` function from outside:

```
// Make grid function space
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");
```

Now comes unchanged code to assemble the constraints, instantiate a coefficient vector, making a discrete grid function that can be used for visualization and interpolating the initial guess and Dirichlet boundary conditions:

```
// Assemble constraints
typedef typename GFS::template
  ConstraintsContainer<RF>::Type CC;
CC cc;
Dune::PDELab::constraints(b,gfs,cc); // assemble constraints
std::cout << "constrained␣dofs=" << cc.size() << "␣of␣"
          << gfs.globalSize() << std::endl;

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

// Make a grid function out of it
typedef Dune::PDELab::DiscreteGridFunction<GFS,Z> ZDGF;
ZDGF zdgf(gfs,z);

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

The next step is to instantiate a local operator, called `NonlinearPoissonFEM`, containing the implementation of the element-wise computations of the finite element method. As explained above the local operator is parametrized by the class `Problem`. In addition, also the finite element map is passed as a template parameter for reasons that will become clear below:

```
// Make a local operator
typedef NonlinearPoissonFEM<Problem<RF>,FEM> LOP;
```

```
LOP lop(problem);
```

Now the grid function space, local operator, matrix backend and constraints container are used to set up a grid operator facilitating the global residual assembly, Jacobian assembly and matrix-free Jacobian application. The matrix backend is initialized with a guess of the approximate number of nonzero matrix entries per row.

```
// Make a global operator
typedef Dune::PDELab::ISTL::BCRSMatrixBackend<> MBE;
int degree = ptree.get("fem.degree",(int)1);
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);
```

In order to prepare for the solution process an appropriate linear solver needs to be selected:

```
// Select a linear solver backend
typedef Dune::PDELab::ISTLBackend_SEQ_CG_AMG_SSOR<GO> LS;
LS ls(100,2);
```

Since the problem is nonlinear we use the implementation of Newton's method in PDELab. It provides the inexact Newton method in the sense that the iterative solution of the linear subproblems is stopped early and uses line search as globalization strategy:

```
// set up nonlinear solver
Dune::PDELab::NewtonMethod<GO,LS> newton(go,ls);
newton.setParameters(ptree.sub("newton"));
```

Now, finally do all the work and solve the problem:

```
// solve nonlinear problem
newton.apply(z);
```

At the end we can write the VTK file with subsampling:

```
// Write VTK output file
int subsampling = ptree.get("output.subsampling",(int)1);
Dune::SubsamplingVTKWriter<GV> vtkwriter(
  gv,
  Dune::refinementIntervals(subsampling)
  );
typedef Dune::PDELab::VTKGridFunctionAdapter<ZDGF> VTKF;
vtkwriter.addVertexData(std::shared_ptr<VTKF>(new
                                        VTKF(zdgf,"fesol")));
```

```
  vtkwriter.write(ptree.get("output.filename","output"),
                  Dune::VTK::appendedraw);
```

4.4 The `Problem` Class

The class `Problem` contained in the file `problem.hh` provides all parameter functions for the PDE problem. It is parameterized with the floating point type to be used:

```
template<typename Number>
class Problem
```

Its constructor takes a parameter η as argument:

```
  //! Constructor takes eta parameter
  Problem (const Number& eta_) : eta(eta_) {}
```

Now come the parameter functions defining the PDE problem. First is the nonlinearity $q(u)$:

```
  //! nonlinearity
  Number q (Number u) const
  {
    return eta*u*u;
  }
```

We also provide the derivative of the function q as a seperate method:

```
  //! derivative of nonlinearity
  Number qprime (Number u) const
  {
    return 2*eta*u;
  }
```

This allows the implementation of an exact Jacobian later (illustrated in tutorial 02) and is actually not needed here as we will use a numerical Jacobian.

Next is the right hand side function f which gets an element `e` and a local coordinate `x` within the corresponding reference element as a parameter:

```
  //! right hand side
  template<typename E, typename X>
  Number f (const E& e, const X& x) const
  {
    return -2.0*x.size();
  }
```

The argument `x` can be expected to be an instance of `Dune::FieldVector` which has a method `size` giving the number of components of the vector, i.e. the space dimension.

The next method simply called `b` is the boundary condition type function. It should return true if the position given by intersection `i` and a local coordinate `x` within the reference element of the intersection is on the Dirichlet boundary. In the particular instance here we set $\Gamma_D = \partial\Omega$:

```
  //! boundary condition type function (true = Dirichlet)
  template<typename I, typename X>
  bool b (const I& i, const X& x) const
  {
    return true;
  }
```

The value of the Dirichlet boundary condition is now defined by the method `g`. As explained above in Section 3.3 it is more appropriate to provide a function u_g that can be evaluated on $\overline{\Omega}$ and gives the value of g on the Dirichlet boundary and the initial guess for the nonlinear solver on all other points:

```
  //! Dirichlet extension
  template<typename E, typename X>
  Number g (const E& e, const X& x) const
  {
    auto global = e.geometry().global(x);
    Number s=0.0;
    for (std::size_t i=0; i<global.size(); i++) s+=global[i]*global[i];
    return s;
  }
```

As with the function `f` above the arguments are an element and a local coordinate in its reference element. Here we evaluate it as $u_g(e,x) = \|\mu_e(x)\|^2$.

Finally, there is a method defining the value of the Neumann boundary condition. Although there is no Neumann boundary here, the method has to be provided but is never called. The arguments of the method are the same as for the boundary condition type function `b`:

```
  //! Neumann boundary condition
  template<typename I, typename X>
  Number j (const I& i, const X& x) const
  {
    return 0.0;
  }
```

4.5 Local Operator `NonlinearPoissonFEM`

The class `NonlinearPoissonFEM` implements the element-wise computations of the finite element method introduced in Section 3.4 above. Evaluation of the residual $R(z)$ is accomplished by the three types of contributions shown in equation (7). In order to make things as simple as possible we chose to implement the evaluation of the Jacobian and the matrix-free Jacobian application with finite differences.

The definition of class `NonlinearPoissonFEM` starts as follows:

```
template<typename Param, typename FEM>
class NonlinearPoissonFEM :
  public Dune::PDELab::
    NumericalJacobianVolume<NonlinearPoissonFEM<Param,FEM> >,
  public Dune::PDELab::
```

```
    NumericalJacobianApplyVolume<NonlinearPoissonFEM<Param,FEM> >,
  public Dune::PDELab::FullVolumePattern,
  public Dune::PDELab::LocalOperatorDefaultFlags
```

The class is parametrized by a parameter class and a finite element map. Implementation of element-wise contributions to the Jacobian and matrix-free Jacobian evaluation is achieved through inheriting from the classes `NumericalJacobianVolume` and `NumericalJacobianApplyVolume`. Using the *curiously recurring template pattern* these classes provide the corresponding methods without any additional coding effort based on the `alpha_volume` method explained below. The other two base classes are the same as in tutorial 00.

The private data members are a cache for evaluation of the basis functions on the reference element:

```
  typedef typename FEM::Traits::FiniteElementType::
     Traits::LocalBasisType LocalBasis;
  Dune::PDELab::LocalBasisCache<LocalBasis> cache;
```

a reference to the parameter object:

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

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

```
  int incrementorder; // increase of integration order
```

The public part of the class starts with the definition of the flags controlling the generic assembly process. The `doPatternVolume` flag specifies that the sparsity pattern of the Jacobian is determined by couplings between degrees of freedom associated with single elements. The corresponding default pattern assembly method is inherited from the class `FullVolumePattern`:

```
  // pattern assembly flags
  enum { doPatternVolume = true };
```

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 { doLambdaVolume = true };
  enum { doLambdaBoundary = true };
  enum { doAlphaVolume = true };
```

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

```
  //! constructor stores a copy of the parameter object
  NonlinearPoissonFEM (Param& param_, int incrementorder_=0)
    : param(param_), incrementorder(incrementorder_)
  {}
```

Method `lambda_volume`

This method was also present in the local operator `PoissonP1` in tutorial 00. It implements the term $\mathcal{L}_T^V$ and has the interface:

```
//! right hand side integral
template<typename EG, typename LFSV, typename R>
void lambda_volume (const EG& eg, const LFSV& lfsv,
                    R& r) const
```

The implementation here uses numerical quadrature of sufficiently high order which is selected at the beginning of the method:

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

The DUNE quadrature rules provide a container of quadrature points that can be iterated over:

```
// loop over quadrature points
for (const auto& ip : rule)
  {
    // evaluate basis functions
    auto& phihat = cache.evaluateFunction(ip.position(),
                          lfsv.finiteElement().localBasis());

    // integrate -f*phi_i
    decltype(ip.weight()) factor = ip.weight()*
      geo.integrationElement(ip.position());
    auto f=param.f(eg.entity(),ip.position());
    for (size_t i=0; i<lfsv.size(); i++)
      r.accumulate(lfsv,i,-f*phihat[i]*factor);
  }
```

At each quadrature point all basis functions are evaluated. The local function space argument `lfsv` provides all the basis functions on the reference element. Evaluations are cached for each point as the evaluation may be quite costly, especially for high order. In addition, copying of data is avoided as the cache returns only a reference to the data stored in the cache. The integration factor is the product of the weight of the quadrature point and the value of $|\det J_{\mu_T}(\hat{x})|$. The implementation works also for non-affine element transformation. The quadrature order should be increased by providing a value for `incrementorder` in the constructor. Then the parameter function can be evaluated and finally the residual contributions for each test function are stored in the result object `r`.

Method `lambda_boundary`

The `lambda_boundary` implements the residual contributions due to Neumann boundary conditions. It implements the term $\mathcal{L}_T^B$ and has the following interface:

```
// Neumann boundary integral
template<typename IG, typename LFSV, typename R>
void lambda_boundary (const IG& ig, const LFSV& lfsv,
                      R& r) const
```

The difference to `lambda_volume` is that now an intersection is provided as first argument.

The method begins by evaluating the type of the boundary condition at the midpoint of the edge:

```
  // evaluate boundary condition type
  auto localgeo = ig.geometryInInside();
  auto facecenterlocal =
    referenceElement(localgeo).position(0,0);
  bool isdirichlet=param.b(ig.intersection(),facecenterlocal);
```

To that end the center of the reference element of the intersection is computed in the variable `facecenterlocal` before the parameter function can be called.

If the boundary condition type evaluated at the face center is Dirichlet then the complete face is assumed to be part of the Dirichlet boundary:

```
  // skip rest if we are on Dirichlet boundary
  if (isdirichlet) return;
```

It is thus assumed that the mesh resolves all positions where the boundary type changes.

Now that we are on a Neumann boundary an appropriate quadrature rule is selected for integration:

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

And here is the integral over the face:

```
  // loop over quadrature points and integrate normal flux
  for (const auto& ip : rule)
    {
      // quadrature point in local coordinates of element
      auto local = localgeo.global(ip.position());

      // evaluate shape functions (assume Galerkin method)
      auto& phihat = cache.evaluateFunction(local,
                           lfsv.finiteElement().localBasis());

      // integrate j
      decltype(ip.weight()) factor = ip.weight()*
        globalgeo.integrationElement(ip.position());
      auto j = param.j(ig.intersection(),ip.position());
      for (size_t i=0; i<lfsv.size(); i++)
```

```
        r.accumulate(lfsv,i,j*phihat[i]*factor);
    }
```

Every quadrature point on the face needs to be mapped to the reference of the volume element for evaluation of the basis functions. The evaluation uses the basis function cache. Then the integration factor is computed and the contributions for all the test functions are accumulated.

Method `alpha_volume`

This method was already present in tutorial 00. It implements the term $\mathcal{R}_T^V(R_T z)$ and 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, const X& x,
                   const LFSV& lfsv, R& r) const
```

The method starts by extracting the space dimension and the floating point type to be used for computations:

```
  // types & dimension
  const int dim = EG::Entity::dimension;
  typedef decltype(Dune::PDELab::
                   makeZeroBasisFieldValue(lfsu)) RF;
```

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);
```

and the quadrature loop is started

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

Within the quadrature loop the basis functions are evaluated

```
      // evaluate basis functions
      auto& phihat = cache.evaluateFunction(ip.position(),
                           lfsu.finiteElement().localBasis());
```

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

```
      // evaluate u
      RF u=0.0;
      for (size_t i=0; i<lfsu.size(); i++)
        u += x(lfsu,i)*phihat[i];
```

Then the gradients of the basis functions on the reference element are evaluated via the evaluation cache:

```
        // evaluate gradient of shape functions
        auto& gradphihat = cache.evaluateJacobian(ip.position(),
                                lfsu.finiteElement().localBasis());
```

Now the gradients need to be transformed from the reference element to the transformed element by multiplication with $J_{\mu_T}^{-1}(\hat{x})$:

```
        // transform gradients of shape functions to real element
        const auto S = geo.jacobianInverseTransposed(ip.position());
        auto gradphi = makeJacobianContainer(lfsu);
        for (size_t i=0; i<lfsu.size(); i++)
          S.mv(gradphihat[i][0],gradphi[i][0]);
```

Note that, as explained in tutorial 00, DUNE allows basis functions in general to be vector valued. Therefore `gradphi[i][0]` contains the gradient (with d components) of the component 0 of basis function number i.

Now ∇u_h can be computed

```
        // compute gradient of u
        Dune::FieldVector<RF,dim> gradu(0.0);
        for (size_t i=0; i<lfsu.size(); i++)
          gradu.axpy(x(lfsu,i),gradphi[i][0]);
```

and we are in the position to finally compute the residual contributions:

```
        // integrate (grad u)*grad phi_i + q(u)*phi_i
        auto factor = ip.weight()*
          geo.integrationElement(ip.position());
        auto q = param.q(u);
        for (size_t i=0; i<lfsu.size(); i++)
          r.accumulate(lfsu,i,(gradu*gradphi[i][0]+
                               q*phihat[i])*factor);
```

4.6 Running the Example

Running tutorial 01 by typing

```
./tutorial01
```

yields an output similar to the following:

```
Parallel code run on 1 process(es)
constrained dofs=128 of 1089
  Initial defect:   5.2962e-02
  Newton iteration  1.  New defect:   1.2256e-04.  Reduction (this):   2.3142e-03.  Reduction (total):   2.3142e-03
  Newton iteration  2.  New defect:   2.2284e-09.  Reduction (this):   1.8182e-05.  Reduction (total):   4.2076e-08
  Newton iteration  3.  New defect:   5.4824e-15.  Reduction (this):   2.4602e-06.  Reduction (total):   1.0351e-13
```

The program reports the number of constrained degrees of freedom (i.e. Lagrange points on the Dirichlet boundary) as well as the total number of degrees of freedom. Then the initial nonlinear residual and the reduction within each Newton iteration is reported.

An illustration of the influence of the nonlinearity q on the solution is given in Figure 1.

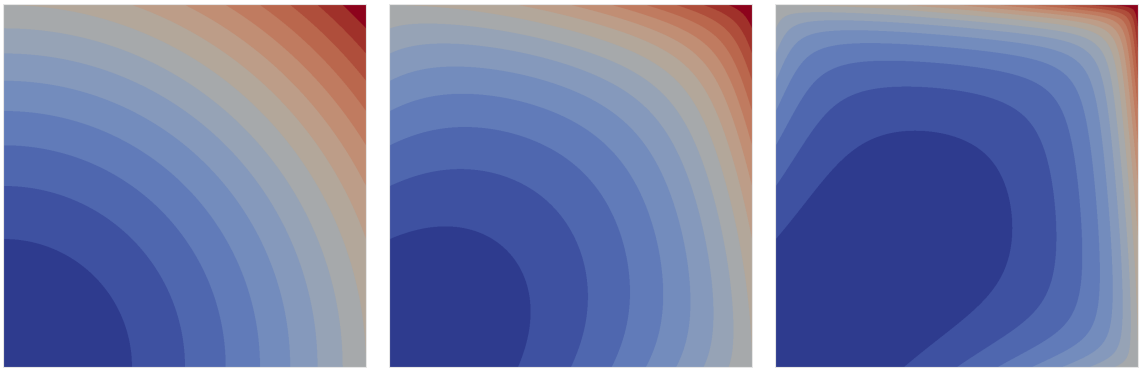

Figure 1: Illustration of the influence of the parameter η in nonlinearity on the solution. $\eta = 0$ (left), $\eta = 10$ (middle), $\eta = 100$ (right).

5 Outlook

Here are some suggestions how to test and modify this example:

- Play with other nonlinearities, e.g. $q(u) = \exp(\eta u)$.
- Compare cost and accuracy of different polynomial degrees and simplicial meshes vs. cube meshes.
- Implement Nitsche's method to incorporate Dirichlet boundary conditions in a weak sense. This method is based on the following residual form:

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

  and requires an additional method `alpha_boundary` with the following interface:

```
template<typename IG, typename LFSU, typename X,
         typename LFSV, typename R>
void alpha_boundary (const IG& ig,
                     const LFSU& lfsu_s, const X& x_s,
                     const LFSV& lfsv_s, R& r_s) const
```

- Implement the streamline diffusion method for a convection term, see [4] for details.

References

[1] D. Braess. *Finite Elemente.* Springer, 3rd edition, 2003.

[2] S. C. Brenner and L. R. Scott. *The mathematical theory of finite element methods.* Springer, 1994.

[3] P. G. Ciarlet. *The finite element method for elliptic problems*. Classics in Applied Mathematics. SIAM, 2002.

[4] H. Elman, D. Silvester, and A. Wathen. *Finite Elements and Fast Iterative Solvers*. Oxford University Press, 2005.

[5] 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`.

[6] A. Ern and J.-L. Guermond. *Theory and practice of finite element methods*. Springer, 2004.

[7] W. Hackbusch. *Theorie und Numerik elliptischer Differentialgleichungen*. Teubner, 1986. `http://www.mis.mpg.de/preprints/ln/lecturenote-2805.pdf`.