DUNE PDELab Tutorial 00
Piecewise Linear Finite Elements for the Poisson Equation on Simplices

DUNE/PDELab Team

June 2, 2016

Contents

1 Introduction

In this tutorial we solve Poisson's equation with piecewise linear conforming finite elements on simplicial elements in one, two and three space dimensions. This can be considered as the "hello world" example in the numerical solution of partial differential equations which every software should be able to solve easily (well, there are codes which have difficulties with triangles). We first provide a short review of the problem and its finite element solution in order to fix the notation. Then we demonstrate how this problem is solved using DUNE and PDELab.

Depends On

This tutorial depends on no other tutorials.

2 Problem Formulation

2.1 Strong Formulation

In this tutorial we consider the *Poisson equation*

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

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

where $\Omega \subset \mathbb{R}^d$ (domains are open and connected sets) is a given, polyhedral domain (elements with curved boundaries are possible in DUNE but will not be considered here). The problem with homogeneous right hand side $f \equiv 0$ is sometimes called *Laplace equation*. This problem is one of the basic equations of mathematical physics which describes gravitational and electric potential as well as stationary heat or groundwater flow. Poisson's equation is an instance of an *elliptic partial differential equation*. More about modeling with partial differential equations can be found in [6, 1].

A function $u \in C^2(\Omega) \cap C^0(\bar{\Omega})$ (the space of twice continuously differentiable functions in Ω which are continuous up to the boundary) is called a *strong solution* if it satisfies equations (1a), (1b) pointwise. Condition (1b) is called a *Dirichlet boundary condition*. Boundary conditions are necessary to render the solution unique and sometimes one speaks also of a *boundary value problem*. Formally, one often reduces (1a), (1b) to a problem with homogeneous Dirichlet boundary conditions $g \equiv 0$ as the starting point for theoretical considerations. We will deliberately not do this here as it is also not done in the computer implementation of the method.

2.2 Weak Formulation

Existence, uniqueness and stability of solutions, i.e. well-posedness in the sense of Hadamard, is easier to prove for so-called weak solutions. As the weak formulation is also the basis of the finite element method we explain it here.

As a start, suppose u is a strong solution of (1a), (1b) and take *any* function $v \in C^1(\Omega) \cap C^0(\bar{\Omega})$ with $v = 0$ on $\partial\Omega$, then we have by integration by parts:

$$\int_\Omega (-\Delta u) v \, dx = \int_\Omega \nabla u \cdot \nabla v \, dx = \int_\Omega f v \, dx.$$

Observe that the boundary integral $\int_{\partial\Omega} \nabla u \cdot n v \, dx$ vanishes due to the fact that $v = 0$ on $\partial\Omega$. Loosely speaking $v = 0$ is a consequence of the Dirichlet boundary condition $u = g$ on $\partial\Omega$.

Introducing the abbreviations

$$a(u, v) = \int_\Omega \nabla u \cdot \nabla v \, dx, \qquad l(v) = \int_\Omega f v \, dx \tag{2}$$

one can on the other hand ask the question: Is there a class, more specific a vector space, of functions V with $V_g = \{v \in V : v = g \text{ on } \partial\Omega\}$ and $V_0 = \{v \in V : v = 0 \text{ on } \partial\Omega\}$ such that the problem

$$u \in V_g : \qquad a(u, v) = l(v) \qquad \forall v \in V_0 \tag{3}$$

has a unique solution. The answer is yes and in particular one can prove the following:

i) With $V = H^1(\Omega)$, the Sobolev space of functions with square integrable weak derivatives, the problem (3) has a unique solution provided the bilinear form $a : V \times V \to \mathbb{R}$ is continuous and coercive on the subspace $V_0 \subset V$ and the linear form $l : V \to \mathbb{R}$ is also continuous. Coercivity on V_0 follows from Friedrich's inequality and for the continuity of the right hand side functional $f \in L^2(\Omega)$ is a sufficient condition.

ii) If in addition $u \in C^2(\Omega) \cap C^0(\bar{\Omega})$, then the solutions of (3) and (1a), (1b) coincide.

We call (3) the *weak formulation* of (1a), (1b). As it has a unique solution under more general conditions than (1a), (1b), e.g. discontinuous right hand side functions f, it can be considered a generalization of the problem.

3 The Finite Element Method

The (conforming) finite element method, in a nutshell, is based on the weak formulation where the function space V is replaced by a subspace $V_h \subset V$ which is *finite dimensional*. Here, the subscript h relates to the dimension of the function space. One major part of the finite element method is to construct such so-called *finite element spaces*. Typically, finite element spaces consist of piecewise polynomial functions. We consider one particular choice, the space of linear functions on simplicial elements. There are many text books about the finite element method, a small selection is [6, 7, 4, 2, 3, 5, 9, 10, 11, 1].

3.1 Finite Element Mesh

In order to construct a finite element space a finite element mesh is required. For simplicity we just consider meshes consisting of d-simplices, where a simplex in d dimensions is the convex hull of $d+1$ points $x_0, \ldots, x_d \in \mathbb{R}^d$ (thus a simplex is, by definition, a closed set of points).

The finite element mesh consists of an ordered set

$$\mathcal{X}_h = \{x_1, \ldots, x_N\} \tag{4}$$

of points in $\mathbb{R}^d$ called *vertices* and an ordered set

$$\mathcal{T}_h = \{T_1, \ldots, T_M\} \tag{5}$$

of d-simplices called *elements*. The elements form a partition of the polygonal domain Ω

$$\bigcup_{T \in \mathcal{T}_h} T = \overline{\Omega}, \quad \forall T, T' \in \mathcal{T}_h, T \neq T' : \mathring{T} \cap \mathring{T}' = \emptyset, \tag{6}$$

and where each of the vertices $x_{T,0}, \ldots, x_{T,d}$ of a d-simplex $T \in \mathcal{T}_h$ coincides with a vertex of the set $\mathcal{X}_h$ (and every $x \in \mathcal{X}_h$ is a vertex of at least one element).

A simplicial mesh is called *conforming* if the intersection of two different elements, $T \cap T'$, is either empty or a facet (a simplex of lower dimension, i.e. a vertex, edge, face) of *both* elements.

The association of the local numbering of vertices within each element and the global numbering of vertices in the vertex set is facilitated via the *local to global map* defined by:

$$\forall T \in \mathcal{T}_h, 0 \leq i \leq d : g_T(l) = j \Leftrightarrow x_{T,l} = x_j. \tag{7}$$

The map $g_T : \{0, \ldots, d\} \to \mathbb{N}$ plays also a very important role in the implementation of the finite element method. Note also that the symbol g is used for the local to global map and the Dirichlet boundary conditions but it should always be clear from the context which function is meant.

The *index set of the vertices* is denoted by $\mathcal{I}_h = \{1, \ldots, N\}$. It can be partitioned into indices of interior and boundary vertices:

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

In order to illustrate the notation introduced, Figure 1 shows an example of a conforming finite element mesh in two space dimensions with its numbering of vertices and elements, as well as the local to global map.

Finally, by

$$h = \max_{T \in \mathcal{T}_h} \operatorname{diam}(T) \tag{8}$$

we denote the *mesh size*.

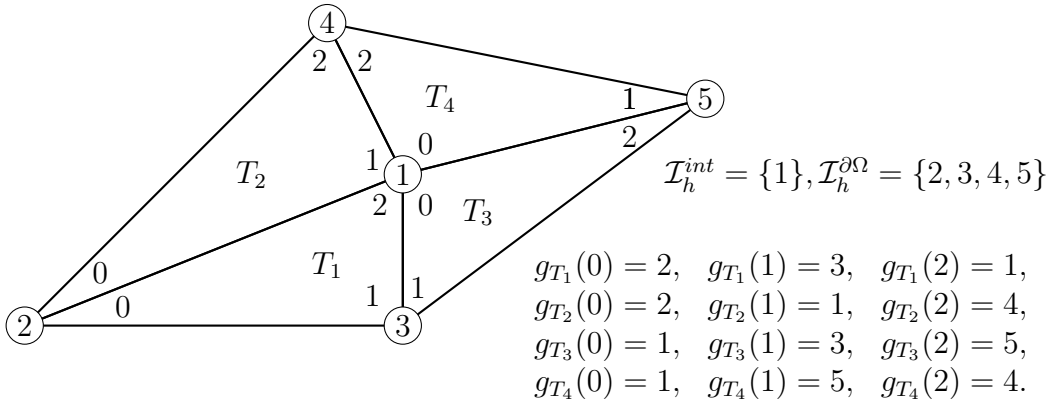

Figure 1: Example of a finite element mesh and the local to global numbering.

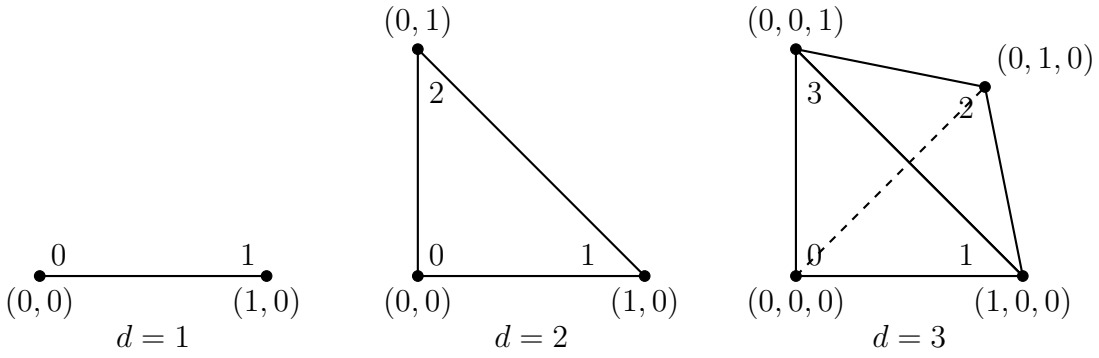

Figure 2: The reference simplices in dimension 1, 2, 3 with vertex positions and local numbering

Reference Elements and Element Transformation

The geometry of the mesh elements can be described more easily by reference elements and an element transformation map. To that end, the reference d-simplex is defined by

$$\hat{T}^d = \left\{ x \in \mathbb{R}^d \,:\, 0 \leq \sum_{i=0}^{d} (x)_i \leq 1 \wedge \forall i : (x)_i \geq 0 \right\}.$$

The vertices of the reference d simplex are given by

$$\hat{x}_0^d = (0, \ldots, 0)^T, \quad \forall i, j \in \{1, \ldots, d\} : (\hat{x}_i^d)_j = \delta_{i,j}.$$

Figure 2 shows the reference elements of dimension 1, 2 and 3 with their numbering of the vertices.

The relation between the reference elements and the elements of the mesh is provided by the *element transformation maps*. For each element $T \in \mathcal{T}$ there is a map

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

which maps points of the reference element to the given element T. In an *affine*

mesh the map μ_T is affine linear, i.e. it has the form

$$\mu_T(\hat{x}) = B_T\hat{x} + a_T$$

for given $d \times d$ matrices B_T and d-vectors a_T. Consistency of the local vertex numbering is ensured by the condition

$$\forall i \in \{0, \ldots, d\} : \mu_T(\hat{x}_i) = x_{T,i}.$$

3.2 Piecewise Linear Finite Element Functions

Given a conforming, simplicial and affine finite element mesh in d dimensions we now can define the space of piecewise linear finite element functions V_h. It is given by

$$V_h(\mathcal{T}_h) = \{v \in C^0(\overline{\Omega}) \ : \ \forall T \in \mathcal{T}_h : v|_T \in \mathbb{P}_1^d\} \tag{9}$$

where

$$\mathbb{P}_1^d = \{p : \mathbb{R}^d \to \mathbb{R} \ : \ p(x) = a^T x + b, a \in \mathbb{R}^d, b \in \mathbb{R}\}$$

is the vector space of multivariate polynomials of degree one in $\mathbb{R}^d$. It turns out that the condition of continuity is crucial to ensure that $V_h \subset H^1(\Omega)$. This definition of the finite element space given above does not refer to a basis. However, for the practical computations one requires a basis for the finite element space. Analysis reveals that the dimension of the finite element space V_h is related to the number of vertices of the mesh:

$$\dim V_h = N = \dim \mathcal{X}_h.$$

Therefore, one may construct a basis $\Phi_h = \{\phi_1, \ldots, \phi_N\}$ of V_h where each basis function ϕ_i is related to vertex $x_i \in \mathcal{X}_h$ in the following way:

$$\forall i, j \in \mathcal{I}_h \ : \ \phi_i(x_j) = \delta_{i,j}.$$

A basis with this property is called a *Lagrange basis*. Exploiting this property of the basis we may define the subspace of finite element functions satisfying homogeneous Dirichlet boundary conditions

$$V_{h,0} = \{v \in V_h \ : \ \forall i \in \mathcal{I}_h^{\partial\Omega} : v(x_i) = 0\}$$

and the set of finite element functions satisfying the given boundary conditions (1b):

$$V_{h,g} = \{v \in V_h \ : \ \forall i \in \mathcal{I}_h^{\partial\Omega} : v(x_i) = g(x_i)\}.$$

Note that $V_{h,0}$ is a subspace of V_h and satisfies the homogeneous boundary data exactly whereas $V_{h,g}$ is *not* a subspace (it is an affine space) and only *approximates* the given boundary data by piecewise linear functions. Twodimensional Lagrange basis functions are illustrated in Figure 3.

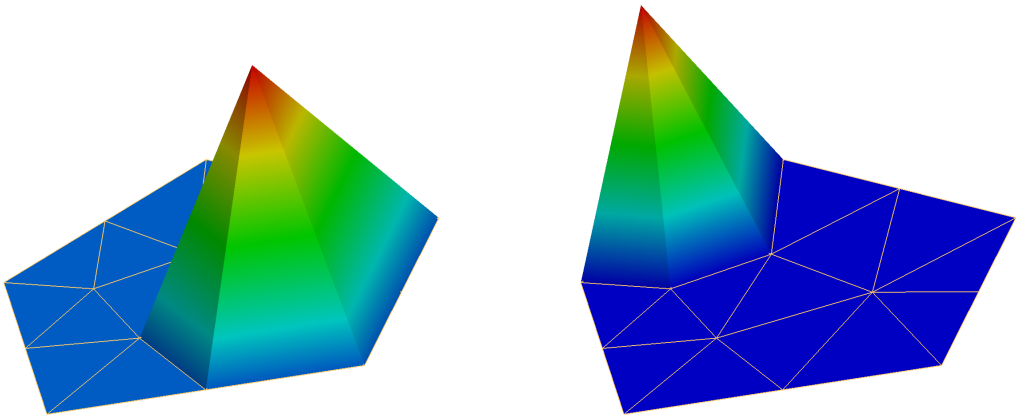

Figure 3: Illustration of piecewise linear Lagrange basis functions in 2d. Left one corresponds to an interior vertex, right to a boundary vertex.

3.3 Finite Element Solution

We are now in a position to define and solve the finite element problem. As pointed out above, the idea is to solve the weak formulation in appropriate finite-dimensional spaces, i.e.:

$$u_h \in V_{h,g}: \quad a(u_h, v) = l(v) \quad \forall v \in V_{h,0}.$$

Using the Lagrange basis defined above we may expand $u_h \in V_h$ as

$$u_h = \sum_{j=1}^{N} (z)_j \phi_j$$

with coefficient vector $z \in \mathbb{R}^N$. Inserting into the discrete weak formulation (3.3) yields:

$$\begin{aligned} & a(u_h, v) = l(v) \quad \forall v \in V_{h,0} && \text{(discrete weak problem)}, \\ \Leftrightarrow\ & a\left(\sum_{j=1}^{N} (z)_j \phi_j, \phi_i\right) = l(\phi_i) \quad \forall i \in \mathcal{I}_h^{int} && \text{(insert basis, linearity)}, \\ \Leftrightarrow\ & \sum_{j=1}^{N} (z)_j a(\phi_j, \phi_i) = l(\phi_i) \quad \forall i \in \mathcal{I}_h^{int} && \text{(linearity)}. \end{aligned} \tag{10}$$

The condition $u_h \in V_{h,g}$ can be formulated as a set of equations

$$u_h(x_i) = z_i = g(x_i) \quad \forall i \in \mathcal{I}_h^{\partial\Omega} \tag{11}$$

which are also linear. Combining the equations (10) and (11) into a single system of linear equations results in

$$Az = b \tag{12}$$

where

$$(A)_{i,j} = \begin{cases} a(\phi_j, \phi_i) & i \in \mathcal{I}_h^{int} \\ \delta_{i,j} & i \in \mathcal{I}_h^{\partial\Omega} \end{cases}, \quad (b)_i = \begin{cases} l(\phi_i) & i \in \mathcal{I}_h^{int} \\ g(x_i) & i \in \mathcal{I}_h^{\partial\Omega} \end{cases}. \tag{13}$$

This system may be solved in various ways. The first option are direct solvers based on some form of the Gaussian elimination technique. However, the matrix A is very sparse as it contains only relatively few nonzero elements per row (3 for $d = 1$, about 7 for $d = 2$ and about 14 for $d = 3$ and Gaussian elimination may have difficulties to exploit this fact, especially for $d = 3$.

Another option is to solve the system iteratively. As the matrix is symmetric and positive definite there is a variety of methods available which produce, starting from an initial iterate z^0, a convergent sequence $\lim_{k\to\infty} z^k = z$. As a solution one accepts the first iterate which satisfies the computable criterion $\|b - Az^k\| < \epsilon\|b - Az^0\|$ for a given reduction factor ϵ.

A very simple (but not very effective) method is *Richardson iteration* which is used to illustrate the concept. It is given by the formula

$$z^{k+1} = z^k + \omega(b - Az^k).$$

Algorithmically, this iterative method can be implemented as follows:

1: Given A, b, ϵ and z ▷ input data
2: $d = b - Az$ ▷ compute initial defect
3: $\tau = \epsilon\|d\|$ ▷ compute target threshold
4: **while** $\|d\| \geq \tau$ **do** ▷ run until convergence
5: $z = z + \omega d$ ▷ update solution
6: $y = Ad$ ▷ matrix-vector product
7: $d = d - \omega y$ ▷ update defect
8: **end while**

It can be observed that the matrix A is only involved in matrix-vector products in lines 2 and 6, an observation that is true for most iterative solvers. This operation can effectively take into account the sparsity structure of the matrix and only computations for non-zero elements are necessary. Thus, one iteration can be implemented with effort $O(N)$. The other major factor in the total work is then the number of iterations needed to achieve the convergence criterion. To keep this number at an acceptable level effective preconditioners are required. We do ignore any discussions on effective preconditioners here but they may require a major portion of the total work.

Matrix-vector products $y = Az$, require between one and three memory accesses for two floating point operations as there is never any reuse of the matrix elements. The exact number depends on the cache reuse of x and y (this is true for dense and sparse matrices). This fact leads to a very low floating point performance on modern processors which are much better at computations than at memory access. A possible way out of this dilemma is to perform the matrix-vector product in a *matrix-free* fashion, i.e. to recompute the matrix elements while performing the matrix-vector product instead of storing them. This may lead to a faster execution of this operation, especially for certain high-order elements. Let us consider the matrix-free execution of the matrix-vector product, better called *operator evaluation*, in more detail. For $i \in \mathcal{I}_h^{int}$ we get

$$(Az)_i = \sum_{j=1}^{N} (A)_{i,j}(z)_j = \sum_{j=1}^{N} a(\phi_j, \phi_i)(z)_j = a\left(\sum_{j=1}^{N} (z)_j\phi_j, \phi_i\right) = a(u_h, \phi_i) \qquad (14)$$

where u_h is the finite element function with the coefficients z. On the other hand, for $i \in \mathcal{I}_h^{\partial\Omega}$ we have

$$(Az)_i = \sum_{j=1}^{N} \delta_{i,j}(z)_j = (z)_i.$$

We may summarize the typical steps needed to solve the finite element problem as follows:

1) Assembling the matrix A. This mainly involves the computation of the matrix elements $a(\phi_j, \phi_i)$ and storing them in an appropriate data structure.

2) Assembling the right hand side vector b. This mainly involves evaluations of the right hand side functional $l(\phi_i)$.

3) Perform a matrix free operator evaluation $y = Az$. This involves evaluations of $a(u_h, \phi_i)$ for all test functions ϕ_i and a given function u_h.

3.4 Implementation of the Solution Steps

We now consider the three operations outlined in the previous section in more detail. The efficient implementation of these operations involves the reference elements and the element transformation as part of the following tools:

Tool 1) Transformation formula for integrals. For $T \in \mathcal{T}_h$ we have

$$\int_T y(x)\,dx = \int_{\hat{T}} y(\mu_T(\hat{x}))|\det B_T|\,dx.$$

Tool 2) Quadrature formula. The midpoint rule reads

$$\int_{\hat{T}} q(\hat{x})\,dx = q(\hat{S}_d)w_d$$

where $\hat{S}_d$ is the center of mass of the reference simplex $\hat{T}^d$ and w_d is the volume of $\hat{T}^d$. This quadrature formula is exact for linear functions.

Tool 3) Shape functions. On the reference simplex the linear Lagrange basis functions are $\hat{\phi}_i(\hat{x}) = (\hat{x})_i$ for $i > 0$ and $\hat{\phi}_0(\hat{x}) = 1 - \sum_{i=1}^{d}(\hat{x})_i$. The basis functions on a general element T can then be defined via transformation

$$\phi_{T,i}(\mu_T(\hat{x})) = \hat{\phi}_i(\hat{x}).$$

This construction principle can be extend to any function defined on the reference element. Given $\hat{w}(\hat{x})$ then

$$w(\mu_T(\hat{x})) = \hat{w}(\hat{x}) \tag{15}$$

is the corresponding function on the general element.

Tool 4) Computation of gradients. The construction via the reference element is particularly useful when computing gradients of functions on the general element. Applying the chain rule to (15) gives

$$B_T^T \nabla w(\mu_T(\hat{x})) = \hat{\nabla} \hat{w}(\hat{x}) \quad \Leftrightarrow \quad \nabla w(\mu_T(\hat{x})) = B_T^{-T} \hat{\nabla} \hat{w}(\hat{x}).$$

Gradients can be computed by computing gradients on the reference element and multiplying them with B_T^{-T}.

Note that all these tools can be extended to higher order basis functions and more general element transformations.

Assembly of the Right Hand Side

We start with the assembly of the right hand side vector b defined in equation (13). Since there are typically much more interior vertices than boundary vertices we may first compute $(b)_i = l(\phi_i)$ for *all* $i \in \mathcal{I}_h$ and then overwrite the entries on the boundary with $(b)_i = g(x_i)$. Moreover, when considering the global index i only the pairs in the set

$$C(i) = \{(T, m) \in \mathcal{T}_h \times \{0, \dots, d\} \,:\, g(T, m) = i\}$$

contribute to the computation, which can be carried out in the following way:

$$\begin{aligned}
(b)_i = l(\phi_i) &= \int_\Omega f \phi_i \, dx && \text{(definition)} \\
&= \sum_{T \in \mathcal{T}_h} \int_T f \phi_i \, dx && \text{(use mesh)} \\
&= \sum_{(T,m) \in C(i)} \int_{\hat{T}} f(\mu_T(\hat{x})) \hat{\phi}_m(\hat{x}) |\det B_T| \, dx && \text{(localize)} \\
&= \sum_{(T,m) \in C(i)} f(\mu_T(\hat{S}_d)) \hat{\phi}_m(\hat{S}_d) |\det B_T| w_d + \text{error}. && \text{(employ quadrature)}
\end{aligned}$$

Note that for general f the integral cannot be computed exactly. The quadrature formula here only yields exact results for elementwise constant functions f as ϕ_i is linear. From now on we ignore this quadrature error.

The computations for all components $i \in \mathcal{I}_h$ are now arranged in such a way that all computations involving element T are carried out together. These computations at element T are:

$$(b_T)_m = f(\mu_T(\hat{S}_d)) \hat{\phi}_m(\hat{S}_d) |\det B_T| w_d \quad \forall m = 0, \dots, d. \tag{16}$$

Then define the restriction matrix $R_T : \mathbb{R}^N \to \mathbb{R}^{d+1}$ as

$$(R_T x)_m = (x)_i \quad \forall\, 0 \le m \le d,\, g_T(m) = i, \tag{17}$$

extracting all components involved with element T. Then the assembly of the right hand side can be written in compact form as

$$b = \sum_{T \in \mathcal{T}_h} R_T^T b_T. \tag{18}$$

Assembly of the Matrix

The assembly of the matrix A defined in (13) can be carried out in a similar way. We assemble first the entries as $(A)_{i,j} = a(\phi_j, \phi_i)$ for all $i, j \in \mathcal{I}_h$ and then modify the matrix to respect the Dirichlet boundary conditions. In the computation of $(A)_{i,j}$ only the triples

$$C(i,j) = \{(T,m,n) \in \mathcal{T}_h \times \{0,\dots,d\} \,:\, g_T(m) = i \wedge g_T(n) = j\}$$

are involved due to the locality of the Lagrange basis functions:

$$\begin{aligned}
(A)_{i,j} = a(\phi_j,\phi_i) &= \int_\Omega \nabla\phi_j \cdot \nabla\phi_i \, dx && \text{(definition)}\\
&= \sum_{T\in\mathcal{T}_h} \int_T \nabla\phi_j \cdot \nabla\phi_i \, dx && \text{(use mesh)}\\
&= \sum_{(T,m,n)\in C(i,j)} \int_{\hat{T}} (B_T^{-T}\hat{\nabla}\hat{\phi}_n(\hat{x})) \cdot (B_T^{-T}\hat{\nabla}\hat{\phi}_m(\hat{x}))|\det B_T| \, d\hat{x} && \text{(localize)}\\
&= \sum_{(T,m,n)\in C(i,j)} (B_T^{-T}\hat{\nabla}\hat{\phi}_n(\hat{S}_d)) \cdot (B_T^{-T}\hat{\nabla}\hat{\phi}_m(\hat{S}_d))|\det B_T| w_d. && \text{(quadrature)}
\end{aligned}$$

Note that the quadrature formula is exact since gradients of linear basis functions and B_T are constant on the element.

Again, the computations are arranged in such a way that all the computations necessary at a single element are collected. To that end, the gradients of the basis functions on the reference element (which are *independent* of position) are collected in the $d \times d+1$ matrix

$$\hat{G} = \left[\hat{\nabla}\hat{\phi}_0(\hat{S}_d)), \dots, \hat{\nabla}\hat{\phi}_d(\hat{S}_d))\right].$$

The matrix $\hat{G}$ need only be computed once as it does not depend on the particular element. With the matrix of transformed gradients $G = B_T^{-T}\hat{G}$ all computations at element T are combined in the so-called *local stiffness matrix* given by

$$A_T = G^T G |\det B_T| w_d. \tag{19}$$

and the system matrix A can be computed as

$$A = \sum_{T\in\mathcal{T}_h} R_T^T A_T R_T. \tag{20}$$

Matrix-free Operator Evaluation

Finally, the considerations above can be applied to the matrix-free operator evaluation (14):

$$
\begin{aligned}
(Az)_i = a(u_h, \phi_i) = \int_\Omega \nabla u_h \cdot \nabla \phi_i \, dx = & \qquad \text{(definition)} \\
= \sum_{T \in \mathcal{T}_h} \int_T \nabla u_h \cdot \nabla \phi_i \, dx & \qquad \text{(use mesh)} \\
= \sum_{(T,m) \in C(i)} \int_{\hat{T}} \left(\sum_{n=0}^{d} (z)_{g_T(n)} B_T^{-T} \hat{\nabla} \hat{\phi}_n \right) \cdot (B_T^{-T} \hat{\nabla} \hat{\phi}_m) |\det B_T| \, d\hat{x} & \qquad \text{(localize)} \\
= \sum_{(T,m) \in C(i)} \left(\sum_{n=0}^{d} (z)_{g_T(n)} B_T^{-T} \hat{\nabla} \hat{\phi}_n \right) \cdot (B_T^{-T} \hat{\nabla} \hat{\phi}_m) |\det B_T| w_d. & \qquad \text{(quadrature)}
\end{aligned}
$$

Again, computations for all indices can be arranged in an element-wise fashion which now computes per element

$$y_T = |\det B_T| w_d G^T G R_T z \tag{21}$$

and then accumulates

$$Az = \sum_{T \in \mathcal{T}_h} R_T^T y_T. \tag{22}$$

Generic Assembly Procedure

Comparing the formulas (18), (20) and (22) for the three basic operations necessary for finite element computations reveals a joint algorithmic form:

1: **for** $T \in \mathcal{T}_h$ **do** ▷ loop over mesh elements
2: $z_T = R_T z$ ▷ load element data
3: $q_T = \text{compute}(T, z_T)$ ▷ element local computations
4: Accumulate(q_T) ▷ store result in global data structure
5: **end for**

It turns out that this basic structure is the same for a huge number of finite element and finite volume methods independently of the partial differential equation to be solved, including linear and nonlinear equations, stationary and time-dependent equations and even systems of equations. Only the element-local computations in step (3) need to be exchanged. Therefore PDELab provides a generic assembler class carrying out steps (1), (2) and (4) while the element-local computations are supplied by a parameter class.

4 Realization in PDELab

The solution of Poisson's equation with piecewise linear finite elements in dimension 1, 2 and 3 is now realized using PDELab. The dimension-independent implementation is an important aspect of this example. The main file is `tutorial00.cc` which includes several other files containing different solution components:

1) File `poissonp1.hh` contains the class template `PoissonP1` realizing the element-local computations comprising the piecewise linear finite element method as described in Subsection 3.4.

2) File `driver.hh` contains the function template `driver` setting up and solving the finite element problem on a particular grid.

3) And finally the file `tutorial00.cc` includes all the other files and contains the `main` function which reads the user parameters, creates a finite element mesh and calls the `driver` function to solve the problem on the given mesh.

We discuss these functions and classes in detail in a top down manner.

4.1 Function `main`

The file `tutorial00.cc` contains the `main` function which is the starting point of every C++ program. All the DUNE code should be within a try block in order to catch any exceptions DUNE might throw and to print meaningful error messages:

```
try{
  ...
}
catch (Dune::Exception &e){
  std::cerr << "Dune␣reported␣error:␣" << e << std::endl;
  return 1;
}
```

The function starts by instantiating the `MPIHelper` singleton:

```
  Dune::MPIHelper&
    helper = Dune::MPIHelper::instance(argc, argv);
  if(Dune::MPIHelper::isFake)
    std::cout<< "This␣is␣a␣sequential␣program." << std::endl;
  else
    std::cout << "Parallel␣code␣run␣on␣"
              << helper.size() << "␣process(es)" << std::endl;
```

In case of a parallel code it initializes the MPI (message passing interface) library. Even if there is no MPI library to initialize there is a default version, so you can always use this code.

The next block of four lines uses the parameter tree parser to read the user data from an input file (colloquially called ini-file) and store it in a parameter tree object named `ptree`:

```
  Dune::ParameterTree ptree;
  Dune::ParameterTreeParser ptreeparser;
  ptreeparser.readINITree("tutorial00.ini",ptree);
  ptreeparser.readOptions(argc,argv,ptree);
```

It is customary that the input file has the same name as the main file of the application with the extension `.ini`. Here is the content of the file `tutorial00.ini`:

```
[grid]
dim=2
refinement=5

[grid.oned]
a=0.0
b=1.0
elements=10

[grid.twod]
filename=unitsquare.msh

[grid.threed]
filename=unitcube.msh

[output]
filename=tuttut
```

The parameter file is structured hierarchically into sections beginning with the section name in square brackets. Within each section names can be associated with strings which can be interpreted in various ways. E.g. the following two lines from the `main` function read the grid's dimension and number of global refinements from the block `[grid]`:

```
    const int dim = ptree.get("grid.dim",(int)2);
    const int refinement = ptree.get<int>("grid.refinement");
```

The first version provides a default value in case the key is not contained in the input file. Note that the type of the object returned by the `get`-method (and the corresponding interpretation of the string in the file) is determined by the type of the default value. In the second version the type is explicitly given by the template parameter and an exception is thrown if the key is not contained in the file.

The rest of the main function creates meshes in 1, 2 and 3 dimensions and calls the function `driver`. Since the grid dimension is a template parameter in DUNE but we want to select the dimension at run-time all three variants need to be compiled.

For the one-dimensional case we use the `OneDGrid` implementation of the DUNE grid interface and construct the initial grid from the user data provided in the ini file.

For the two- and three-dimensional case either `ALUGrid` or `UGGrid` are used and `ALUGrid` is preferred if it is available. If neither is present the code cannot be run. Both grid managers can read two- and three-dimensional simplicial grids generated by the program `gmsh` [8]. In the following we just explain the 2d section here as all sections are similar.

First we define the type `Grid` to be either `ALUGrid` or `UGGrid` using some preprocessor magic. An error message is printed if neither grid manager is installed:

```
#if HAVE_DUNE_ALUGRID
  typedef Dune::ALUGrid<2,2,Dune::simplex,
                        Dune::nonconforming> Grid;
#elif HAVE_UG
```

```
  typedef Dune::UGGrid<2> Grid;
#else  // ! (HAVE_UG || HAVE_DUNE_ALUGRID)
  std::cout << "Example␣requires␣a␣simplex␣grid!" << std::endl;
#endif
```

Now we can create a DUNE grid initialized with the coarse mesh from the gmsh input file:

```
#if (HAVE_UG || HAVE_DUNE_ALUGRID)
  std::string filename = ptree.get("grid.twod.filename",
                                   "unitsquare.msh");
  Dune::GridFactory<Grid> factory;
  Dune::GmshReader<Grid>::read(factory,filename,true,true);
```

Here the method `get` is provided with the name of the entry in the parameter file and a default value in case the entry is not present in the file.

The next few lines refine the mesh globally the specified number of times and report the time spent:

```
  Dune::Timer timer;
  gridp->globalRefine(refinement);
  std::cout << "Time␣for␣mesh␣refinement␣" << timer.elapsed()
            << "␣seconds" << std::endl;
```

Voilà, we can call the function `driver` to solve the finite element problem on the given mesh (which in this case is the finest mesh in the complete hierarchy, the so called leaf view):

```
  driver(gridp->leafGridView(),ptree);
```

4.2 Function `driver`

The generic `driver` function contains all the PDELab code that sets up and solves the finite element problem. The solution of complicated problems such as nonlinear problems, instationary problems or systems of partial differential equations follows the same pattern with some of the components exchanged, as will become clear in further examples.

The function `driver` has the following interface:

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

The first argument is supposed to provide a leaf grid view of a conforming simplicial grid in any space dimension. A grid view is a subset of a hierarchical finite element mesh as it is provided by the DUNE grid interface. A leaf grid view provides the finest mesh defined in the hierarchy and here it represents the mesh $\mathcal{T}_h$ on which we want to solve the finite element problem. The second argument provides a parameter tree containing user data. Currently only the output file name is taken from the parameter tree.

The function starts by extracting the dimension of the grid (we assume that grid and world dimension coincide) and the type used by the grid to represent

coordinates. Then the type to be used for the entries of the vectors and matrices is defined:

```
const int dim = GV::dimension;
typedef typename GV::Grid::ctype DF; // type for ccordinates
```

The next step is to instantiate objects representing the data of the partial differential equation to be solved:

```
auto flambda = [](const auto& x){
  return Dune::FieldVector<RF,1>(-2.0*x.size());};
auto f = Dune::PDELab::
  makeGridFunctionFromCallable(gv,flambda);
```

Here we use one way where a generic lambda function `flambda` (a C++14 feature) is defined which returns a value associated with a point `x` in global coordinates. In this case it denotes the right hand side f in the partial differential equation. Note that it is mandatory to explicitly specify the type for the return value in order to make the automatic extraction of the return type work. Then the lambda function (actually closure) is used by the function `makeGridFunctionFromCallable` which wraps the lambda function into a class having the interface of a PDELab `GridFunction`. Such types and corresponding objects can then be used to interpolate finite element functions or provide graphical output.

The same code can now be used to define a function g extending the Dirichlet boundary values into the interior. This function can be used to provide e.g. the exact solution of the problem for testing purposes or an initial guess for the iterative solvers.

```
auto glambda = [](const auto& x){
  RF s=0.0; for (std::size_t i=0; i<x.size(); i++) s+=x[i]*x[i];
  return s;};
auto g = Dune::PDELab::
  makeGridFunctionFromCallable(gv,glambda);
```

Finally we need to declare *where* Dirichlet boundary conditions are to be applied. In our example, Dirichlet boundary conditions are applied on all of $\partial\Omega$ but in general one may apply also other boundary conditions.

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

The return type of the lambda function must be `bool`. Note that the function `makeBoundaryConditionFromCallable` is now used as the specification of constraints on function spaces, which requires a class with a different interface in PDELab.

The purpose of the next block of lines

```
typedef Dune::PDELab::PkLocalFiniteElementMap<GV,DF,RF,1> 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("P1");
```

is to set up a grid function space represented by the type `GFS`. It can be considered to represent the finite element space V_h, i.e. it knows about the dimension of the space, the basis functions as well as the local to global map.

The first two lines set up a finite element map of type `PkLocalFiniteElementMap`. A finite element map associates finite element basis functions, defined on the corresponding reference element, with each element of the mesh. In our simple case every element of the mesh is supposed to have the same basis functions but in general, e.g. in *hp* finite element methods, every element could have a different set of basis functions. In addition, information is provided how the global finite element space V_h is to be constructed from its local, element-wise pieces. This involves the identification of global degrees of freedom via the local to global map.

The next line defines the type `CON`, a so-called constraints class, which provides a way to assemble constraints on a function space. In our case it is used to identify degrees of freedom constrained by Dirichlet boundary conditions.

The following line defines the type `VBE` which provides a vector backend. PDELab is designed in such a way that different iterative solver libraries can be used. Such libraries also provide their own data types for vectors and (sparse) matrices and we would like PDELab to be able to directly fill the data into these data structures without a copy step. In this case here we use the ISTL vector backend using DUNE's own iterative solver library ISTL.

Now all template parameters are in place to define the type `GFS` from the class template `GridFunctionSpace`. This class template combines all the given information to construct the global finite element space V_h on a given grid view. Finally an object of this class is instantiated and the space is given a name.

The grid function space actually corresponds to the unconstrained function space V_h. The `CON` class does not provide the constraints themselves but rather *a way to determine the constraints*. The determination of the constraints for a specific grid function space from the boundary condition type function `b` constructed above is done by the following lines:

```
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;
```

The `constraints` function assembles the constraints, in our case the index set $\mathcal{I}_h^{\partial\Omega}$, into the constraints container `cc` of type `CC`. The separation of the function space V_h and the constraints set $\mathcal{I}_h^{\partial\Omega}$ allows one to reuse the function space together with different constraints, e.g. for a system of partial differential equations.

The next step is to declare a variable that will later on contain the solution vector z:

```
using Z = Dune::PDELab::Backend::Vector<GFS,RF>;
```

```
Z z(gfs); // initial value
```

The type `Z` representing the $\mathbb{R}^N$ is extracted from the vector backend of the grid function space while we are still able to specify the type `RF` for each component of the solution vector.

The finite element isomorphism $\mathrm{FE}_h : \mathbb{R}^N \to V_h$,

$$\mathrm{FE}_h(z) = \sum_{j=1}^{N} (z)_j \phi_j$$

provides a one-to-one correspondence between coefficient vectors and finite element functions. The following lines produce a finite element function from a coefficient vector:

```
typedef Dune::PDELab::DiscreteGridFunction<GFS,Z> ZDGF;
ZDGF zdgf(gfs,z);
```

You should be aware that the object `zdgf` stores a *reference* to the object `z` which means that when you change entries of the coefficient vector then also the function changes.

Often one wants to fill a coefficient vector such that $\mathrm{FE}_h(z)$ approximates a given function, say w. In general, w is not a finite element function, but if it is, then $w = \mathrm{FE}_h(z)$ should hold. So what we seek is $z = \mathrm{FE}_h^{-1}(P(w))$ where P is a projection into the finite element space. This is provided by the following line:

```
Dune::PDELab::interpolate(g,gfs,z);
```

In this case the projection P used by the function `interpolate` is the Lagrange interpolation of the function g:

$$P(g) = \sum_{j=1}^{N} g(x_j)\phi_j,$$

i.e. $(z)_j = g(x_j)$ where x_j are the mesh vertices. The projection to be used depends on the finite element space and is part of the definition of the grid function space. For example in the case of discontinuous finite element functions it might be an L^2-projection.

The line

```
Dune::PDELab::set_nonconstrained_dofs(cc,0.0,z);
```

then sets all interior degrees of freedom to zero.

As pointed out in Subsection 3.4 the assembly of the right hand side b and the matrix A as well as matrix-free computation of Az can be separated into a generic part looping over the finite element mesh and doing element-local computations. This separation is represented in the code by first setting up a *local operator*, here of the type `LOP`

```
typedef PoissonP1<decltype(f),FEM> LOP;
LOP lop(f,fem.find(*gv.template begin<0>()));
```

providing the element-wise computations. The class template `PoissonP1` implementing the piecewise linear finite element method for Poisson's equation is described in detail below. The constructor requires the right hand side function as the first argument and the finite element of the first element of the mesh as the second argument. The finite element is used to precompute the basis functions on the reference element as well as the gradients. Importantly, we assume that the same basis is used for all mesh elements.

Now the local operator is used as one of the template arguments in the global assembler or grid operator:

```
typedef Dune::PDELab::istl::BCRSMatrixBackend<> MBE;
MBE mbe(1<<(dim+1)); // guess nonzeros per row
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);
```

The grid operator implemented by the type `GO` provides the generic part of the assembly procedure. It requires the types for ansatz and test space, which may be different in general, the local operator type, a matrix backend, the types to be used for components of coefficients vectors of the ansatz and test space as well as the entries of the Matrix A and last but not least the types of the constraints containers of ansatz and test space. The constructor then takes the corresponding objects as arguments. Note that here an object `mbe` of the matrix backend type is needed. It is constructed with a guess of the average number of nonzeros per row.

As the next step we need to select a solver that will be used to solve the linear system $Az = b$. This is done by the following two lines:

```
typedef Dune::PDELab::ISTLBackend_SEQ_CG_AMG_SSOR<GO> LS;
LS ls(100,3);
```

Since we used the ISTL backends for vectors and matrices we need to select a solver from the ISTL library. Complete solvers are packaged by PDELab for sequential and parallel computations. Here we select the conjugate gradient method with algebraic multigrid as preconditioner and symmetric successive overrelaxation as smoother in multigrid in its sequential implementation. The linear solver object `ls` is initialized with the maximum number of iterations and a verbose parameter.

So far, *no* finite element computations have actually been performed, only the necessary components have been configured to now do the real work together. We can now set up the matrix A as well as the right hand side b and then solve the system. Since this is required often, there is a class in PDELab for this:

```
typedef Dune::PDELab::
  StationaryLinearProblemSolver<GO,LS,Z> SLP;
SLP slp(go,ls,z,1e-10);
```

The object `slp` of type `StationaryLinearProblemSolver` receives the grid operator, the selected linear solver backend and a coefficient vector with the initial guess and the correct Dirichlet boundary data and solves the problem up to a given accuracy upon a call of the apply method:

```
slp.apply(); // here all the work is done!
```

In the given example a problem with a known exact solution, which is given by the function g, is solved. In order to compare the computed solution with the exact solution we initialize another coefficient vector with the Lagrange interpolant of the exact solution and provide a grid function:

```
Z w(gfs); // Lagrange interpolation of exact solution
Dune::PDELab::interpolate(g,gfs,w);
ZDGF wdgf(gfs,w);
```

Finally it is time to write the results to disk for postprocessing with VTK/ParaView. This is done with DUNE's `VTKWriter` class:

```
Dune::VTKWriter<GV> vtkwriter(gv,Dune::VTK::conforming);
typedef Dune::PDELab::VTKGridFunctionAdapter<ZDGF> VTKF;
vtkwriter.addVertexData(std::shared_ptr<VTKF>(new
                                              VTKF(zdgf,"fesol")));
vtkwriter.addVertexData(std::shared_ptr<VTKF>(new
                                              VTKF(wdgf,"exact")));
vtkwriter.write(ptree.get("output.filename","output"),
                Dune::VTK::appendedraw);
```

The VTK writer is not part of PDELab and uses a different interface to represent functions on a grid. Therefore we need to use the adapter class `VTKF` to pass PDELab grid functions to the VTK writer. Moreover, objects of this adapter class should be passed via a `std::shared_ptr` to the VTK writer object since this takes care about the memory management. The output file is written when the `write` method is called.

4.3 Local Operator `PoissonP1`

The finite element method itself is implemented in the so-called *local operator* realized by the class template `PoissonP1` in file `poissonp1.hh`. It provides all the necessary element-local computations as described in Subsection 3.4 and is declared as follows:

```
template<typename F, typename FiniteElementMap>
class PoissonP1;
```

The first template parameter provides the right hand side function of the PDE and the second parameter provides a finite element map giving access to finite element basis functions on the reference element for all elements of the grid. The class derives from the PDELab classes `FullVolumePattern` and `LocalOperatorDefaultFlags` which provide some default constants and methods.

The basic assumption of this implementation of the finite element method is that all elements of the mesh are simplices of dimension d which use the same polynomial

degree 1. In order to make the code faster it is a good idea to do the evaluation of the basis functions and their gradients on the reference element *once* before the computations start. This will be done in the constructor, but before we can do so we need to do some preparations.

Type Definitions and Data Members

The class begins by extracting important types. The finite element map provides a finite element for each element of the map. Its type is

```
typedef typename FiniteElementMap::Traits::FiniteElementType
  FiniteElementType;
```

Among other things the finite element contains the basis functions on the reference element which can be accessed via the following type:

```
typedef typename FiniteElementType::Traits::LocalBasisType
  LocalBasisType;
```

DUNE thinks of basis functions on the reference element to be of the most general form

$$\hat{\phi} : \mathbb{A}^d \to \mathbb{B}^k, \quad \nabla\hat{\phi} : \mathbb{A}^d \to \mathbb{B}^{k\times d},$$

i.e. they may be vector-valued. The following type definitions

```
typedef typename LocalBasisType::Traits::DomainType
  DomainType;
typedef typename LocalBasisType::Traits::RangeFieldType
  RF;
typedef typename LocalBasisType::Traits::RangeType
  RangeType;
typedef typename LocalBasisType::Traits::JacobianType
  JacobianType;
```

provide types to represent arguments and results of basis function evaluations. `DomainType` represents $\mathbb{A}^d$, `RF` represents $\mathbb{B}$, `RangeType` represents $\mathbb{B}^k$ and finally `JacobianType` represents $\mathbb{B}^{k\times d}$.

Next, we extract some important constants, the dimension of the grid and the number of basis functions per element:

```
enum {dim=LocalBasisType::Traits::dimDomain};
enum {n=dim+1};
```

As private data members the class stores an instance of the right hand side function `f` provided by the `driver`:

```
const F f;                // right hand side function
```

the midpoint quadrature rule

```
DomainType qp;            // center of mass of refelem
double weight;            // quadrature weight on refelem
```

where `qp` is $\hat{S}_d$ and `weight` is w_d, and the values of the basis functions at the quadrature point and their gradients:

```
  double phihat[n];        // basis functions at qp
  double gradhat[dim][n]; // coordinate x #basisfct
```

Then, already in the public part, we need to define some constants that control the operation of the grid operator doing the global assembly:

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

These constants are evaluated at *compile time* and tell the grid operator class which methods have been implemented in the local operator by the user. Actually, the base class `LocalOperatorDefaultFlags` provides all possible flags with the value `false` and we just need to overwrite the ones that are needed. The constant `doPatternVolume` tells the global assembler to determine the sparsity pattern of the matrix A from a method `pattern_volume` which is inherited from the base class `FullVolumePattern`. This default implementation inserts nonzeros between all degrees of freedom of an element. The constants `doAlphaVolume` and `doLambdaVolume` determine that our finite element method contains a volume integral involving the finite element solution u_h and a right hand side integral which does not involve the finite element solution.

Setting `doAlphaVolume` to true implies that the local operator class implements the methods `alpha_volume`, `jacobian_apply_volume` and `jacobian_volume`. Setting `doLambdaVolume` to true implies that the method `lambda_volume` must be implemented.

Constructor

The constructor of the class has the following signature:

```
  PoissonP1 (const F& f_, const FiniteElementType& fel)
```

It takes the right hand side function `f` and a finite element `fel` as argument. The finite element is obtained from the finite element map and the first element of the grid in the function `driver`.

First thing to do is to get the lowest order quadrature rule for simplices from DUNE:

```
    Dune::GeometryType gt = fel.type();
    const Dune::QuadratureRule<RF,dim>&
      rule = Dune::QuadratureRules<RF,dim>::rule(gt,1);
```

Then we check that this is actually the midpoint rule

```
    if (rule.size()>1) {
      std::cout << "Wrong␣quadrature␣rule!" << std::endl;
      exit(1);
    }
```

and store the first quadrature point in the local data members:

```
    weight = rule[0].weight();
    qp = rule[0].position();
```

It is also a good idea to check that the basis given by the user has at least the correct size:

```
    if (fel.localBasis().size()!=n) {
      std::cout << "Wrong basis!" << std::endl;
      exit(1);
    }
```

Now the basis functions can be evaluated at the quadrature point in the reference element and the results are stored in the data members of the class:

```
    std::vector<RangeType> phi(n);
    fel.localBasis().evaluateFunction(qp,phi);
    for (int i=0; i<n; i++) phihat[i] = phi[i];
```

And the same now for the gradients:

```
    std::vector<JacobianType> js(n);
    fel.localBasis().evaluateJacobian(qp,js);
    for (int i=0; i<n; i++)
      for (int j=0; j<dim; j++)
        gradhat[j][i] = js[i][0][j];
```

Note that the last index loops over the number of basis functions.

Method `lambda_volume`

This method computes the contributions b_T to the right hand side vector for a given element as given in Eq.(16). It has the following signature:

```
  template<typename EG, typename LFSV, typename R>
  void lambda_volume (const EG& eg, const LFSV& lfsv,
                      R& r) const
```

Argument `eg` provides the element T in a wrapped form such that PDELab need not operate directly on a DUNE grid. With `eg.geometry()` the geometry of the element can be accessed in the form of a `Dune::Geometry`. With `eg.entity()` one can access the underlying codim 0 entity of the DUNE grid. The second argument `lfsv` provides the test functions on the reference element and `r` provides a container where the result should be stored.

First thing to do is to evaluate the right hand side function at the quadrature point:

```
    typename F::Traits::RangeType fval;
    f.evaluate(eg.entity(),qp,fval);
```

Next, we compute the factor that is common to all entries of b_T:

```
    RF factor=fval*weight*eg.geometry().integrationElement(qp);
```

Note that the method `integrationElement` on the geometry provides the value of $|\det B_T|$.

Finally, we can compute the entries and store them in the results container:

```
for (int i=0; i<n; i++)
  r.accumulate(lfsv,i,-factor*phihat[i]);
```

Here it is important to note the minus sign because PDELab actually solves the weak formulation as

$$r(u_h, v) = a(u_h, v) - l(v) = 0 \quad \forall v \in V$$

since this is more appropriate in the case of nonlinear partial differential equations.

Method `jacobian_volume`

Next we need to compute the element contributions to the stiffness matrix as described in Eq.(19). This is done by the method `jacobian_volume` with the following interface:

```
template<typename EG, typename LFSU, typename X,
         typename LFSV, typename M>
void jacobian_volume (const EG& eg, const LFSU& lfsu,
                      const X& x, const LFSV& lfsv,
                      M& mat) const
```

Its arguments are: `eg` providing the wrapped codim 0 entity T, `lfsu` providing the basis functions of the ansatz space, `x` providing the coefficients of the current iterate of the finite element solution, `lfsv` providing the test functions and `mat` a container to store the result.

The `jacobian_volume` method works in the same way also for nonlinear problems. Nonlinear problems are solved iteratively, e.g. using Newton's method or a fixed-point iteration, where the method should provide a linearization at the current iterate given by the combination of `lfsu` and `x`. In our case of a linear problem the result *does not depend* on the current iterate. Moreover, the basis functions for the test space are precomputed so we need not access them via `lfsv`. Note also that in general the ansatz and test space might be different.

First thing we need is to get B_T^{-T} and store it into `S`:

```
const auto geo = eg.geometry();
const auto S = geo.jacobianInverseTransposed(qp);
```

Next, $|\det B_T|$ is retrieved from the geometry and the factor that is common to all entries of the local stiffness matrix is computed:

```
RF factor = weight*geo.integrationElement(qp);
```

Now form the matrix of transformed gradients $G = B_T^{-T}\hat{G}$ and store it in `grad`:

```
double grad[dim][n] = {{0.0}}; // coordinate x #basisfct
for (int i=0; i<dim; i++) // rows of S
  for (int k=0; k<dim; k++) // columns of S
    for (int j=0; j<n; j++) // columns of gradhat
      grad[i][j] += S[i][k] * gradhat[k][j];
```

The computations are arranged in such a way that the innermost loop has the dimension number of basis functions. In 3d there are four basis functions and the loop has a chance to get vectorized.

Now the local stiffness matrix $A_T = G^T G$ (up to the factor $|\det B_T| w_d$) is formed

```
double A[n][n] = {{0.0}};
for (int i=0; i<n; i++)
  for (int k=0; k<dim; k++)
    for (int j=0; j<n; j++)
      A[i][j] += grad[k][i]*grad[k][j];
```

and stored in the results container (now multiplying with the common factor):

```
for (int i=0; i<n; i++)
  for (int j=0; j<n; j++)
    mat.accumulate(lfsu,i,lfsu,j,A[i][j]*factor);
```

Method `alpha_volume`

The method `alpha_volume` provides the element-local computations for the matrix-free evaluation of $a(u_h, \phi_i)$ for all test functions ϕ_i as given by Eq.(21). It has the interface:

```
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
```

Its arguments are: `eg` providing the wrapped codim 0 entity T, `lfsu` providing the basis functions of the ansatz space, `x` providing the coefficients of the current iterate of the finite element solution, `lfsv` providing the test functions and `r` a container to store the result.

The computations are actually quite similar to those in `jacobian_volume`. In particular, the computation of B_T^{-T}, $|\det B_T|$ and $G = B_T^{-T}\hat{G}$ are the same.

Extracting the element local coefficients $z_T = R_T z$ is done by:

```
double z_T[n];
for (int j=0; j<n; j++) z_T[j] = x(lfsu,j); // read coeffs
```

Now we may compute ∇u_h via Gz_T:

```
double graduh[dim] = {0.0};
for (int k=0; k<dim; k++) // rows of grad
  for (int j=0; j<n; j++) // columns of grad
    graduh[k] += grad[k][j]*z_T[j];
```

Finally, the result $a_T = G^T \nabla u_h$ is formed:

```
double a_T[n] = {0.0};
for (int k=0; k<dim; k++) // rows of grad
  for (int j=0; j<n; j++)
    a_T[j] += grad[k][j]*graduh[k];
```

and stored in the results container (while being multiplied with the common factor):

```
    for (int i=0; i<n; i++)
      r.accumulate(lfsv,i,a_T[i]*factor);
```

Method `jacobian_apply_volume`

In the case of a *nonlinear* partial differential equation the finite element method results in a weak form

$$u_h \in V_h \;:\quad r(u_h, v) = \alpha(u_h, v) - \lambda(v) = 0 \quad \forall v \in V_h$$

which is *nonlinear* in its *first* argument. Inserting the finite element basis results in a nonlinear algebraic problem

$$R(z) = 0$$

with $(R(z))_i = r(\mathrm{FE}_h(z), \phi_i)$ which is typically solved by Newton's iteration or some other iterative method. In case of Newton's method, each step involves the solution of a *linear* system of the form

$$J(z)\, w = R(z)$$

where $(J(z))_{i,j} = \frac{(\partial R(z))_i}{\partial z_j} = \frac{\partial \alpha(\mathrm{FE}_h(z), \phi_i)}{\partial z_j}$ is the Jacobian of the nonlinear map R.

Naturally, the nonlinear case also includes the linear case described in this tutorial by setting $r(u, v) = a(u, v) - l(v)$. Then, due to the linearity of a in its first argument, one can show that $J(z) = A$ and

$$(J(z)\, w)_i = (Aw)_i = a(\mathrm{FE}_h(w), \phi_i).$$

This is *not* true in the nonlinear case. There, the evaluation of the form $\alpha(\mathrm{FE}_h(w), \phi_i)$ and the application of the Jacobian Jw are different operations. Therefore, PDELab provides two functions with the application of the Jacobian implemented in `jacobian_apply_volume` with the interface:

```
template<typename EG, typename LFSU, typename X,
         typename LFSV, typename R>
void jacobian_apply_volume (const EG& eg, const LFSU& lfsu,
                            const X& z, const LFSV& lfsv,
                            R& r) const
```

Note, this is the same interface as for `alpha_volume`. Since our problem is linear, Jacobian application is identical to bilinear form evaluation and therefore we may just forward the call to the function `alpha_volume`:

```
    alpha_volume(eg,lfsu,z,lfsv,r);
```

4.4 Running the Example

Now we can run the tutorial and look at the results. One may check that $u(x) = \sum_{i=1}^{d}(x)_i^2$ solves the PDE $-\Delta u = -2d$ in d dimensions (in $d = 1$ this is not a PDE but a two-point boundary value problem). So we provide the exact solution as Dirichlet boundary data and set $f = -2d$.

The program can be run by typing

```
./tutorial00
```

on the command line. It then produces some output on the console and a VTK file with the extension `.vtp` on one space dimension and `.vtu` in two and three space dimensions.

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 133 nodes
file contains 268 elements
number of real vertices = 133
number of boundary elements = 36
number of elements = 228
```

Now an instance of a DUNE grid is created and refined the required number of times:

```
Created serial ALUGrid<2,2,simplex,nonconforming>.
Time for mesh refinement 0.011566 seconds
```

Next, Dirichlet boundary constraints are evaluated and statistics are reported:

```
constrained dofs=288 of 7441
```

Now the matrix and right hand side are set up:

```
=== matrix setup (max) 0.013918 s
=== matrix assembly (max) 0.012634 s
=== residual assembly (max) 0.009034 s
```

The solver is started, in this case using the conjugate gradient method with an algebraic multigrid preconditioner. This preconditioner needs a set up phase which produces the following output:

```
=== solving (reduction: 1e-10)
Using a direct coarse solver (UMFPACK)
Building hierarchy of 2 levels (inclusive coarse solver) took
0.022688 seconds.
```

Finally, the solver prints some statistics about the convergence:

```
=== CGSolver
   12       1.9016e-09
=== rate=0.144679, T=0.021999, TIT=0.00183325, IT=12
0.044764 s
```

Figure 4 shows the finite element solution u_h as well as the absolute error $|u - u_h|$ for the $2d$ problem with the exact solution $u(x, y) = x^2 + y^2$. The largest error is obtained in the vertices of the coarse mesh.

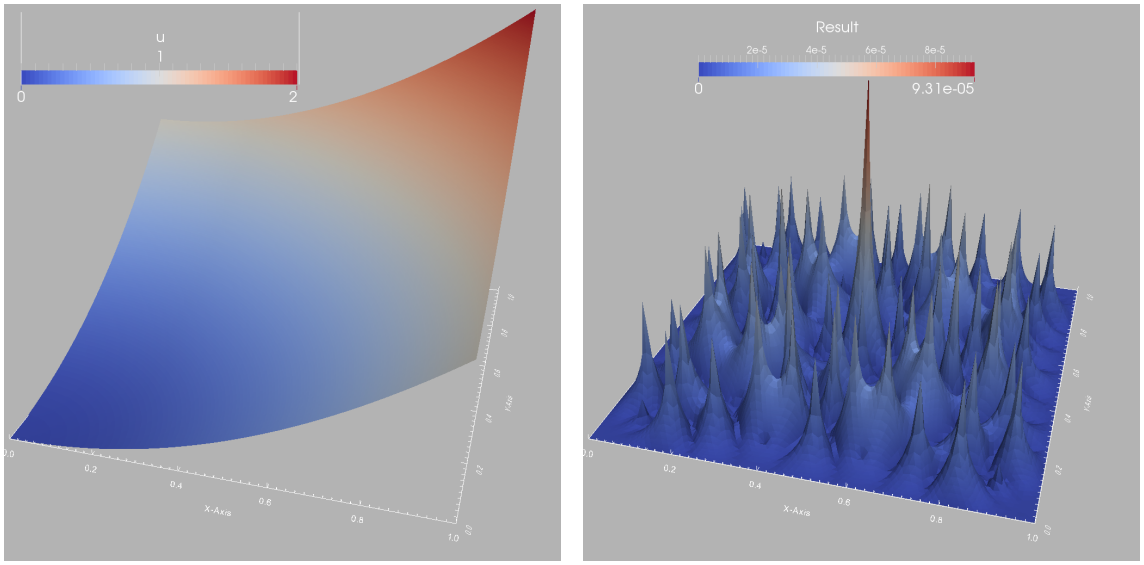

Figure 4: Finite element solution u_h (left) and absolute error $|u - u_h|$ (right) visualized as surface for a two-dimensional simulation.

5 Outlook

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

- Run the problem on various levels of refinement and determine the maximum error. The maximum error should behave like $O(h^2)$ with the mesh size.
- Try a different solution, like $u(x, y) = x^3 + y^3$, change the program accordingly and study the error with respect to mesh refinement.
- Replace the algebraic multigrid preconditioner with a different one, e.g. the BiCGStab method with SSOR preconditioner:

```
typedef Dune::PDELab::ISTLBackend_SEQ_BCGS_SSOR LS;
LS ls(5000,true);
```

Other, more involved options which will be covered in further tutorials are:

- Implementation of Neumann type boundary conditions involving boundary integrals.
- Extension of the finite element method to cube elements using multi-linear basis functions.
- Extension of the finite element method to higher order polynomials.

References

[1] P. Bastian. Lecture notes on scientific computing with partial differential equations. http://conan.iwr.uni-heidelberg.de/teaching/numerik2_ss2014/num2.pdf, 2014.

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

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

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

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

[6] 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.

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

[8] Christophe Geuzaine and Jean-François Remacle. Gmsh: A 3-d finite element mesh generator with built-in pre- and post-processing facilities. *International Journal for Numerical Methods in Engineering*, 79(11):1309–1331, 2009.

[9] C. Großmann and H.-G. Roos. *Numerische Behandlung partieller Differentialgleichungen*. Teubner, 2006.

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

[11] R. Rannacher. Einführung in die Numerische Mathematik II (Numerik partieller differentialgleichungen). http://numerik.iwr.uni-heidelberg.de/~lehre/notes, 2006.