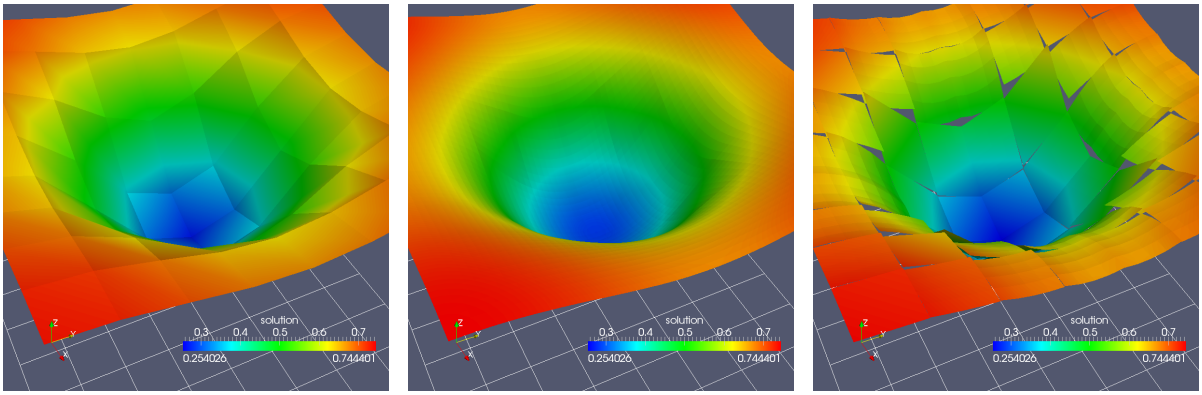
dune-pdelab **Howto**

DUNE TEAM
Universität Heidelberg
Interdisziplinäres Zentrum für Wissenschaftliches Rechnen
Im Neuenheimer Feld 368, D-69120 Heidelberg
email: Peter.Bastian@iwr.uni-heidelberg.de

March 22, 2010



http://www.dune-project.org/

This article contains concepts for a general discretization module for the "Distributed Numerics Environment" DUNE [BBD+08b, BBD+08a]. It should enable one to build up a library of finite element methods in an easy and extendable way that is closely related to the mathematical formulation of finite element methods.

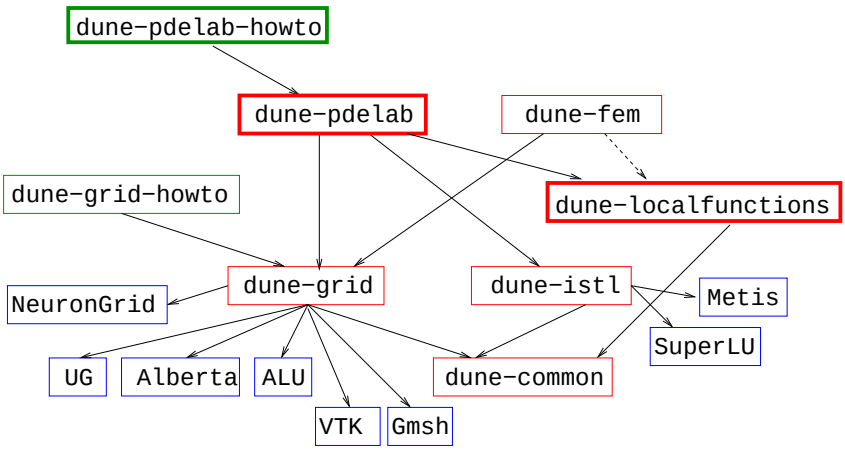
Contents

1 Introduction

1.1 PDELab Aims and Features

- Rapid prototyping: Substantially reduce time to implement discretizations and solvers for systems of PDEs based on DUNE.
- Simple things should be simple — suitable for teaching.
- Discrete function spaces spaces:
  - Conforming and non-conforming,
  - hp-refinement,
  - general approach to constraints,
  - generic generation of product spaces for systems.
- Operators based on weighted residual formulation:
  - Linear and nonlinear,
  - stationary and transient,
  - FE and FV schemes requiring at most face-neighbors.
- Exchangeable linear algebra backend.
- User only involved with "local" view on (reference) element.

Major DUNE modules are:



To work through the examples the following DUNE modules are required:

- dune−common,
- dune−grid,
- dune−istl,

- dune−localfunctions,
- dune−pdelab,
- dune−pdelab−howto,

In addition, at least one of the grid managers UG, ALU or Alberta is required to do the examples on simplex grids.

1.2 How to Read this Manual

The main idea of this howto is to introduce the concepts by working through a set of increasingly complex examples. We will start by solving stationary elliptic problems first without and then with constrained spaces (Dirichlet boundary conditions). After some excursion to linear algebra backends and working with CAD models instationary problems and systems will be treated.

2 Solving Stationary Problems

Code reuse requires abstraction.

- Put mathematical problem in abstract form.
- Approach is based on *weighted residual formulation.*
- Represent mathematical concepts by classes.

Abstractions should support orthogonality.

- Features that can be combined independently:
- various kinds of grids and function spaces.
- sequential and parallel (overlapping/nonoverlapping).
- stationary and instationary.
- linear and nonlinear.

Abstractions enable team work.

We demonstrate these concepts in the following.

2.1 Unconstrained Elliptic Model Problem

Problem and Weak Formulation

Consider the following model problem:

$$\begin{aligned} -\Delta u + au &= f && \text{in } \Omega \subset \mathbb{R}^d \text{ (open, connected)}, \\ \nabla u \cdot n &= 0 && \text{on } \partial\Omega. \end{aligned}$$

Weak formulation. Set $U = H^1(\Omega)$.

$$u \in U \quad : \quad \underbrace{\int_\Omega \nabla u \cdot \nabla v + auv - fv \, dx}_{r(u,v)} = 0 \qquad \forall v \in U.$$

Has unique solution for $a(x) \geq a_0 > 0$.

We call $r(u, v)$ residual form.

Other boundary conditions are treated later.

Conforming Finite Element Method

Needs conforming triangulation $E_h^0 = \{e_o, \ldots, e_{N_h^0 - 1}\}$ of Ω.

Define the conforming finite element space

$$U_h^k = \left\{ u \in C^0(\overline{\Omega}) \ : \ u|_{\Omega_e} \in P_{k_e} \forall e \in E_h^0 \right\} \subset H^1(\Omega).$$

- Ω_e: domain of element $e \in E_h^0$.
- P_k: Polynomials of degree k.
- k_e: Polynomial degree on element e.

Discrete problem then reads:

$$u_h \in U_h^k \quad : \quad r(u_h, v) = 0 \qquad \forall v \in U_h^k.$$

Affine Finite Element Spaces

Construct functions in U_h^k from local basis on reference elements:

$$U_h^k \ni u_h(x) = \sum_{e \in E_h^0} \sum_{l=0}^{n(e)-1} (\mathbf{u})_{g(e,l)} \, \hat{\phi}_{e,l}(\mu_e^{-1}(x)) \, \chi_e(x).$$

- $n(e)$: Number of basis functions on element $e \in E_h^0$.
- $\hat{\Omega}_e$: Reference element of element $e \in E_h^0$.
- $\mu_e : \hat{\Omega}_e \to \Omega_e$: Element transformation.
- $\hat{\phi}_{e,l} : \hat{\Omega}_e \to \mathbb{R}$: Local basis function.
- $\mathcal{I}_{U_h^k} = \{0, \ldots, N_{U_h^k} - 1\}$: Global index set.
- $g : E_h^0 \times \mathbb{N}_0 \to \mathcal{I}_{U_h^k}$: Local to global index map.
- $\mathbf{u} \in \mathbf{U} = \mathbb{R}^{\mathcal{I}_{U_h^k}}$: Global vector of degrees of freedom.
- χ_e: Characteristic function of element e.

Note: We might have a different set of basis functions on each element.

Global Basis; Finite Element Isomorphism

For $j \in \mathcal{I}_{U_h^k}$ set $L(j) = \{(e,l) \, : \, g(e,l) = j\}$ (all local degrees of freedom associated with global degree of freedom j.

Global basis:

$$\Phi_{U_h^k} = \left\{ \phi_j(x) = \sum_{(e,l) \in L(j)} \hat{\phi}_{e,l}(\mu_e^{-1}(x)) \, \chi_e(x) \, : \, j \in \mathcal{I}_{U_h^k} \right\}.$$

Finite Element Isomorphism:

$$\mathrm{FE}_{\Phi_{U_h^k}} : \mathbf{U} \to U_h^k, \qquad \mathrm{FE}_{\Phi_{U_h^k}}(\mathbf{u}) = \sum_{j \in \mathcal{I}_{U_h^k}} (\mathbf{u})_j \phi_j.$$

Algebraic Problem

Using the basis the discrete problem can be written equivalently as a (in general nonlinear) algebraic problem:

$$\begin{aligned} & u_h \in U_h^k \quad : \quad r(u_h, v) = 0 && \forall v \in U_h^k, \\ \Leftrightarrow \quad & \mathbf{u} \in \mathbf{U} \quad : \quad r\left(\mathrm{FE}_{\Phi_{U_h^k}}(\mathbf{u}), \phi_i\right) = 0 && i \in \mathcal{I}_{U_h^k}, \\ \Leftrightarrow \quad & \mathbf{u} \in \mathbf{U} \quad : \quad \mathcal{R}(\mathbf{u}) = \mathbf{0}. \end{aligned}$$

where

$$\mathcal{R} : \mathbf{U} \to \mathbf{U}, \qquad (\mathcal{R}(\mathbf{u}))_i := r\left(\mathrm{FE}_{\Phi_{U_h^k}}(\mathbf{u}), \phi_i\right).$$

For linear PDEs $\mathcal{R}$ is affine linear: $\mathcal{R}(\mathbf{u}) = \mathbf{A}\mathbf{u} - \mathbf{b}$.

Residual Assembly

$$\begin{aligned} (\mathcal{R}(\mathbf{u}))_i &= r\left(\mathrm{FE}(\mathbf{u}), \phi_i\right) = \sum_{e \in E_h^0} \int_{\Omega_e} \nabla \mathrm{FE}(\mathbf{u}) \cdot \nabla \phi_i + a\, \mathrm{FE}_{\Phi_{U_h^k}}(\mathbf{u}) \phi_i - f \phi_i \, dx \\ &= \sum_{e \in E_h^0} \int_{\Omega_e} \left[\sum_{l=0}^{n(e)-1} (\mathbf{u})_{g(e,l)} \nabla_x \hat{\phi}_{e,l}(\mu_e^{-1}(x)) \right] \cdot \nabla_x \underbrace{\hat{\phi}_{e,m}}_{g(e,m)=i} (\mu_e^{-1}(x)) \\ &+ a \left[\sum_{l=0}^{n(e)-1} (\mathbf{u})_{g(e,l)} \hat{\phi}_{e,l}(\mu_e^{-1}(x)) \right] \hat{\phi}_{e,m}(\mu_e^{-1}(x)) - f \hat{\phi}_{e,m}(\mu_e^{-1}(x)) \, dx \\ &= \sum_{e \in E_h^0} \int_{\hat{\Omega}_e} \Bigg\{ \left[\sum_{l=0}^{n(e)-1} (\mathbf{u})_{g(e,l)} (\nabla \mu_e(\hat{x}))^{-T} \nabla_{\hat{x}} \hat{\phi}_{e,l}(\hat{x}) \right] \cdot (\nabla \mu_e(\hat{x}))^{-T} \nabla_{\hat{x}} \hat{\phi}_{e,m}(\hat{x}) \\ &+ a \left[\sum_{l=0}^{n(e)-1} (\mathbf{u})_{g(e,l)} \hat{\phi}_{e,l}(\hat{x}) \right] \hat{\phi}_{e,m}(\hat{x}) - f \hat{\phi}_{e,m}(\hat{x}) \Bigg\} \det \nabla \mu_e(\hat{x}) \, d\hat{x}. \end{aligned}$$

Local Operator

Define restriction to local degrees of freedom

$$\mathbf{U}_e = \mathbb{R}^{n(e)}, \qquad \mathbf{R}_e : \mathbf{U} \to \mathbf{U}_e, \qquad (\mathbf{U}_e(\mathbf{u}))_l = (\mathbf{u})_{g(e,l)} \quad 0 \le l < n(e).$$

Define *local operator* $\boldsymbol{\alpha}_{h,e}^{\text{vol}} : \mathbf{U}_e \to \mathbf{U}_e$ (user part):

$$\begin{aligned}
&\left(\boldsymbol{\alpha}_{h,e}^{\text{vol}}(\mathbf{u})\right)_m = \\
&\int_{\hat{\Omega}_e} \left\{ \left[\sum_{l=0}^{n(e)-1} (\mathbf{u})_l (\nabla \mu_e(\hat{x}))^{-T} \nabla_{\hat{x}} \hat{\phi}_{e,l}(\hat{x}) \right] \cdot (\nabla \mu_e(\hat{x}))^{-T} \nabla_{\hat{x}} \hat{\phi}_{e,m}(\hat{x}) \right. \\
&\left. + a \left[\sum_{l=0}^{n(e)-1} (\mathbf{u})_l \hat{\phi}_{e,l}(\hat{x}) \right] \hat{\phi}_{e,m}(\hat{x}) - f \hat{\phi}_{e,m}(\hat{x}) \right\} \det \nabla \mu_e(\hat{x}) \, d\hat{x}.
\end{aligned}$$

Residual assembly is written generically:

$$\mathcal{R}(\mathbf{u}) = \sum_{e \in E_h^0} \mathbf{R}_e^T \boldsymbol{\alpha}_{h,e}^{\text{vol}}(\mathbf{R}_e \mathbf{u})$$

Solving the Algebraic System

Use damped Newton method.

Given $\mathbf{u}^0 \in \mathbf{U}$. Compute $\mathbf{r}^0 = \mathcal{R}(\mathbf{u}^0)$. Set $k = 0$.

Iterate until convergence:

1. Assemble Jacobian System $\mathbf{A}^k = \nabla \mathcal{R}(\mathbf{u}^k)$.
2. Solve $\mathbf{A}^k \mathbf{z}^k = \mathbf{r}^k$ with some linear solver.
3. Update $\mathbf{u}^{k+1} = \mathbf{u}^k - \sigma^k \mathbf{z}^{k+1}$. $\sigma \in (0, 1]$.
4. Compute new residual $\mathbf{r}^{k+1} = \mathcal{R}(\mathbf{u}^{k+1})$.
5. Set $k = k + 1$.

We need methods to compute $\mathcal{R}(\mathbf{u})$ and $\nabla \mathcal{R}(\mathbf{u})$.

Jacobian

The Jacobian matrix is defined as

$$(\mathbf{A}^k)_{i,j} = (\nabla \mathcal{R}(\mathbf{u}^k))_{i,j} = \frac{\partial (\mathcal{R})_i}{\partial (\mathbf{u})_j}(\mathbf{u}^k) = \sum_{e \in E_h^0} \frac{\partial (\boldsymbol{\alpha}_{h,e}^{\text{vol}})_m}{\partial (\mathbf{u})_l}(\mathbf{R}_e \mathbf{u}),$$

where $g(e, m) = l, g(e, l) = j$.

Again, the Jacobian can be computed from local contributions:

$$\mathbf{A}^k = \sum_{e \in E_h^0} \mathbf{R}_e^T \nabla \boldsymbol{\alpha}_{h,e}^{\text{vol}}(\mathbf{R}_e \mathbf{u}) \, \mathbf{R}_e.$$

The local Jacobians can be

- programmed explicitly by the user, or
- derived generically through numerical differentiation. This requires only coding of the local residual contributions $\boldsymbol{\alpha}_{h,e}^{\text{vol}}$.

The Linear Case

is a special case of the nonlinear case ...

1. Given $\mathbf{u}^0 \in \mathbf{U}$.
2. Compute $\mathbf{r} = \mathcal{R}(\mathbf{u}^0)$.
3. Assemble Jacobian System $\mathbf{A} = \nabla\mathcal{R}(\mathbf{u}^0)$.
4. Solve $\mathbf{Az} = \mathbf{r}$ with some linear solver.
5. Update $\mathbf{u} = \mathbf{u}^0 - \mathbf{z}$.

2.2 Example 1

Example 1 Overview

The first example implements model problem (1).

It consists of the following files:

- example01.cc – the file to be compiled.
- example01_main.hh – main function. Instantiates a grid and runs the variants.
- example01a_Q1.hh – solve model problem (1) with Q_1 elements.
- example01a_Q2.hh – same with Q_2 elements.
- example01a_RT.hh – same with nonconforming rotated bilinear ("Rannacher-Turek" element).
- example01a_operator.hh – local operator implementing $\boldsymbol{\alpha}_{h,e}^{\text{vol}}$.
- example01b_Q2.hh – solve nonlinear variant of the model problem (1) with Q_2 elements.
- example01b_operator.hh – local operator for nonlinear variant.

For completeness we show the main function that just instantiates a YaspGrid object and calls the variants.

Main functions will not be shown in later examples.

Listing 1 (File examples/example01_main.hh).

```
1 int main(int argc, char** argv)
2 {
3   try{
4     //Maybe initialize Mpi
5     Dune::MPIHelper& helper = Dune::MPIHelper::instance(argc, argv);
6     if(Dune::MPIHelper::isFake)
7       std::cout<< "This is a sequential program." << std::endl;
8     else
```

```
9       {
10        if(helper.rank()==0)
11          std::cout << "parallel run on " << helper.size() << " process(es)" << std::endl;
12      }
13
14    if (argc!=2)
15      {
16        if(helper.rank()==0)
17          std::cout << "usage: ./example01 <level>" << std::endl;
18        return 1;
19      }
20
21    int level;
22    sscanf(argv[1],"%d",&level);
23
24    // sequential version
25    if (helper.size()==1)
26    {
27      Dune::FieldVector<double,2> L(1.0);
28      Dune::FieldVector<int,2> N(1);
29      Dune::FieldVector<bool,2> periodic(false);
30      int overlap=0;
31      Dune::YaspGrid<2> grid(L,N,periodic,overlap);
32      grid.globalRefine(level);
33      typedef Dune::YaspGrid<2>::LeafGridView GV;
34      const GV& gv=grid.leafView();
35      example01a_Q1(gv);
36      example01a_Q2(gv);
37      example01a_RT(gv);
38      example01b_Q2(gv);
39    }
40  }
41  catch (Dune::Exception &e){
42    std::cerr << "Dune reported error: " << e << std::endl;
43    return 1;
44  }
45  catch (...){
46    std::cerr << "Unknown exception thrown!" << std::endl;
47    return 1;
48  }
49 }
```

Driver for Solving Stationary Linear Problems About example01a_Q1.hh

1. Define useful constants/types like dimension or basic numeric type.
2. Make *grid function space* which corresponds here to U_h^k. It requires
   - a *finite element map* defining a local basis on each element.
   - a method to set up *constraints* on the function space (empty here).
   - a suitable vector backend.
3. Make *grid operator space* computing $\mathcal{R}(\mathbf{u})$, $\nabla\mathcal{R}(\mathbf{u})$. It requires
   - a local operator which provides $\boldsymbol{\alpha}_{h,e}^{\text{vol}}$.
   - trial and test grid function spaces, possibly with constraints.
   - a suitable matrix backend.

4. Select a linear solver backend (see vector/matrix backend).
5. Solve the linear problem given with the selected solver backend.
   - *Vector container* is used to store degrees of freedom $\mathbf{u} \in \mathbf{U}$.
6. Output graphics files to visualize solution with ParaView.
   - *Discrete grid function* implements finite element isomorphism.

Listing 2 (File examples/example01a_Q1.hh).

```
1 template<class GV>
2 void example01a_Q1 (const GV& gv)
3 {
4   // <<<1>>> Choose domain and range field type
5   typedef typename GV::Grid::ctype Coord;
6   typedef double Real;
7   const int dim = GV::dimension;
8
9   // <<<2>>> Make grid function space
10  typedef Dune::PDELab::Q1LocalFiniteElementMap<Coord,Real,dim> FEM;
11  FEM fem;
12  typedef Dune::PDELab::NoConstraints CON;
13  typedef Dune::PDELab::ISTLVectorBackend<1> VBE;
14  typedef Dune::PDELab::GridFunctionSpace<GV,FEM,CON,VBE> GFS;
15  GFS gfs(gv,fem);
16  typedef typename GFS::template ConstraintsContainer<Real>::Type CC;
17
18  // <<<3>>> Make grid operator space
19  typedef Example01aLocalOperator LOP;
20  LOP lop;
21  typedef Dune::PDELab::ISTLBCRSMatrixBackend<1,1> MBE;
22  typedef Dune::PDELab::GridOperatorSpace<GFS,GFS,LOP,CC,CC,MBE> GOS;
23  GOS gos(gfs,gfs,lop);
24
25  // <<<4>>> Select a linear solver backend
26  typedef Dune::PDELab::ISTLBackend_SEQ_BCGS_SSOR LS;
27  LS ls(5000,true);
28
29  // <<<5>>> solve linear problem
30  typedef typename GFS::template VectorContainer<Real>::Type U;
31  U u(gfs,0.0); // initial value
32  typedef Dune::PDELab::StationaryLinearProblemSolver<GOS,LS,U> SLP;
33  SLP slp(gos,u,ls,1e-10);
34  slp.apply();
35
36  // <<<6>>> graphical output
37  typedef Dune::PDELab::DiscreteGridFunction<GFS,U> DGF;
38  DGF udgf(gfs,u);
39  Dune::VTKWriter<GV> vtkwriter(gv,Dune::VTKOptions::conforming);
40  vtkwriter.addVertexData(new Dune::PDELab::VTKGridFunctionAdapter<DGF>(udgf,"solution"));
41  vtkwriter.write("example01_Q1",Dune::VTKOptions::binaryappended);
42 }
```

Local Operator

- The local operator implements $\boldsymbol{\alpha}_{h,e}^{\text{vol}}$ (and more).
- Class template GridOperatorSpace builds on a local operator and provides $\mathcal{R}(\mathbf{u})$, $\nabla\mathcal{R}(\mathbf{u})$ *generically*.

- Works for many different discretizations (see below).
- Works as well for systems of PDEs (see below).

Local Operator Implementation About example01a_operator.hh

A local operator is a class providing the following:

- Flags controlling the sparsity pattern assembly.
- Method pattern_volume assembling sparsity pattern (default provided).
- Flags controlling which terms to assemble.
- Method alpha_volume computing $\boldsymbol{\alpha}_{h,e}^{\mathrm{vol}}(\mathbf{u})$.
- Method jacobian_volume computing $\nabla\boldsymbol{\alpha}_{h,e}^{\mathrm{vol}}(\mathbf{u})$. This method can be provided generically through numerical differentiation.
- Method jacobian_apply_volume computing $\nabla\boldsymbol{\alpha}_{h,e}^{\mathrm{vol}}(\mathbf{u})\mathbf{u}$. This method can be provided generically through numerical differentiation.
- Possibly more methods (to be introduced later):
  - alpha_boundary, alpha_skeleton – boundary/interior face integrals.
  - lambda_volume, lambda_boundary – parts of the residual depending on the test function only (optional).

alpha_volume Method

The method alpha_volume has the following signature:

```
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;
```

Where the arguments are:

- eg – a codim 0 entity $e \in E_h^0$.
- lfsu – local basis $\hat{\phi}_{e,l}$ for trial space.
- x – local coefficients $(\mathbf{u})_l$.
- lfsv – local basis $\hat{\psi}_{e,l}$ for the test space
- r – local contribution to residual (the result).

Listing 3 (File examples/example01a_operator.hh).

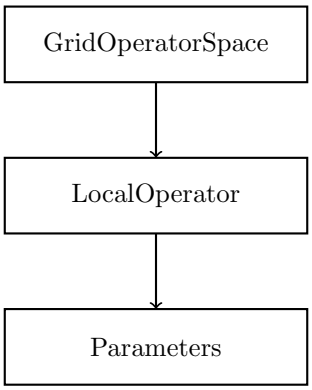
```
1#include<dune/grid/common/genericreferenceelements.hh>
2#include<dune/grid/common/quadraturerules.hh>
3#include<dune/pdelab/common/geometrywrapper.hh>
4#include<dune/pdelab/localoperator/pattern.hh>
5#include<dune/pdelab/localoperator/flags.hh>
6
7/** a local operator for solving the equation
```

```
 8 *
 9 *   - \Delta u + au    = f    in \Omega
10 *     \nabla u \cdot n = 0    on \partial\Omega
11 *
12 * with conforming finite elements on all types of grids in any dimension
13 *
14 */
15 class Example01aLocalOperator :
16   public Dune::PDELab::NumericalJacobianApplyVolume<Example01aLocalOperator>,
17   public Dune::PDELab::NumericalJacobianVolume<Example01aLocalOperator>,
18   public Dune::PDELab::FullVolumePattern,
19   public Dune::PDELab::LocalOperatorDefaultFlags
20 {
21 public:
22   // pattern assembly flags
23   enum { doPatternVolume = true };
24
25   // residual assembly flags
26   enum { doAlphaVolume = true };
27
28   Example01aLocalOperator (unsigned int intorder_=2)
29     : intorder(intorder_)
30   {}
31
32   // volume integral depending on test and ansatz functions
33   template<typename EG, typename LFSU, typename X, typename LFSV, typename R>
34   void alpha_volume (const EG& eg, const LFSU& lfsu, const X& x, const LFSV& lfsv, R& r) const
35   {
36     // extract some types
37     typedef typename LFSU::Traits::LocalFiniteElementType::
38       Traits::LocalBasisType::Traits::DomainFieldType DF;
39     typedef typename LFSU::Traits::LocalFiniteElementType::
40       Traits::LocalBasisType::Traits::RangeFieldType RF;
41     typedef typename LFSU::Traits::LocalFiniteElementType::
42       Traits::LocalBasisType::Traits::JacobianType JacobianType;
43     typedef typename LFSU::Traits::LocalFiniteElementType::
44       Traits::LocalBasisType::Traits::RangeType RangeType;
45     typedef typename LFSU::Traits::SizeType size_type;
46
47     // dimensions
48     const int dim = EG::Geometry::dimension;
49     const int dimw = EG::Geometry::dimensionworld;
50
51     // select quadrature rule
52     Dune::GeometryType gt = eg.geometry().type();
53     const Dune::QuadratureRule<DF,dim>& rule = Dune::QuadratureRules<DF,dim>::rule(gt,intorder);
54
55     // loop over quadrature points
56     for (typename Dune::QuadratureRule<DF,dim>::const_iterator
57            it=rule.begin(); it!=rule.end(); ++it)
58       {
59         // evaluate basis functions on reference element
60         std::vector<RangeType> phi(lfsu.size());
61         lfsu.localFiniteElement().localBasis().evaluateFunction(it->position(),phi);
62
63         // compute u at integration point
64         RF u=0.0;
65         for (size_type i=0; i<lfsu.size(); i++)
66           u += x[i]*phi[i];
67
68         // evaluate gradient of basis functions on reference element
69         std::vector<JacobianType> js(lfsu.size());
70         lfsu.localFiniteElement().localBasis().evaluateJacobian(it->position(),js);
71
```

```
72          // transform gradients from reference element to real element
73          const Dune::FieldMatrix<DF,dimw,dim>
74            jac = eg.geometry().jacobianInverseTransposed(it->position());
75          std::vector<Dune::FieldVector<RF,dim> > gradphi(lfsu.size());
76          for (size_type i=0; i<lfsu.size(); i++)
77            jac.mv(js[i][0],gradphi[i]);
78
79          // compute gradient of u
80          Dune::FieldVector<RF,dim> gradu(0.0);
81          for (size_type i=0; i<lfsu.size(); i++)
82            gradu.axpy(x[i],gradphi[i]);
83
84          // evaluate parameters
85          Dune::FieldVector<RF,dim>
86            globalpos = eg.geometry().global(it->position());
87          Dune::FieldVector<RF,dim> midpoint(0.5);
88          globalpos -= midpoint;
89          RF f;
90          if (globalpos.two_norm()<0.25) f = -10.0; else f = 10.0;
91          RF a =  10.0;
92
93          // integrate grad u * grad phi_i + a*u*phi_i - f phi_i
94          RF factor = it->weight()*eg.geometry().integrationElement(it->position());
95          for (size_type i=0; i<lfsu.size(); i++)
96            r[i] += ( gradu*gradphi[i] + a*u*phi[i] - f*phi[i] )*factor;
97        }
98    }
99
100 private:
101   unsigned int intorder;
102 };
```

Remark on Local Operators

In practice one would separate parameter functions such as a, f, j and the boundary condition type from the implementation of the local operator.



The interface between LocalOperator and Parameters is up to you.

Figure 1 shows visualizations of the results computed with example01.

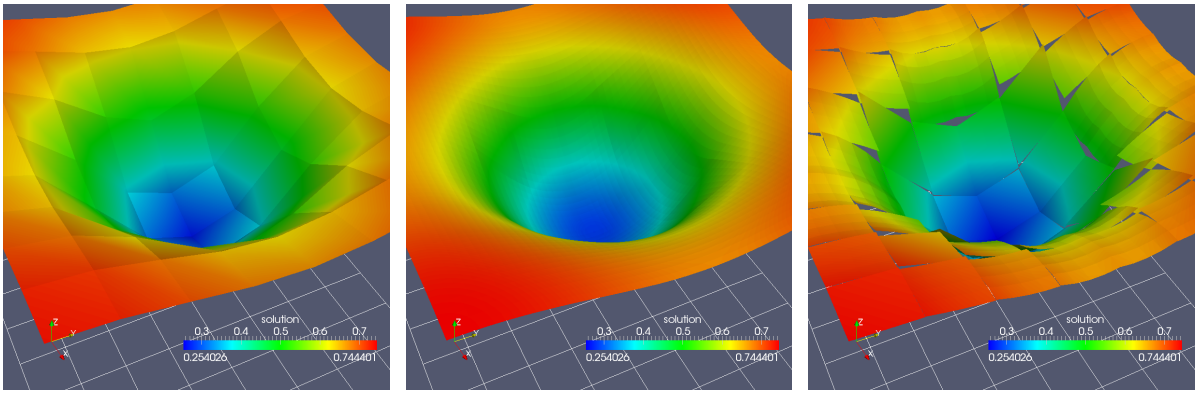


Figure 1: Results for example 1a computed with three different finite element spaces. From left: Q_1 elements, Q_2 elements, rotated bilinear (Rannacher-Turek) element on an 8×8 grid.

Using Q_2 Elements

... is quite simple, see example01a_Q2.hh. Just

- Use another finite element map Q22DLocalFiniteElementMap.
- Increase quadrature order on the local operator to 4.
- Use SubsamplingVTKWriter to allow visualization of higher order polynomials.
- Use a new name for the output file.

Explore the Rannacher Turek element in example01a_RT.hh

Listing 4 (File examples/example01a_Q2.hh).

```
1 template<class GV>
2 void example01a_Q2 (const GV& gv)
3 {
4   // <<<1>>> Choose domain and range field type
5   typedef typename GV::Grid::ctype Coord;
6   typedef double Real;
7   const int dim = GV::dimension;
8
9   // <<<2>>> Make grid function space
10  typedef Dune::PDELab::Q22DLocalFiniteElementMap<Coord,Real> FEM;          // <= NEW
11  FEM fem;
12  typedef Dune::PDELab::NoConstraints CON;
13  typedef Dune::PDELab::ISTLVectorBackend<1> VBE;
14  typedef Dune::PDELab::GridFunctionSpace<GV,FEM,CON,VBE> GFS;
15  GFS gfs(gv,fem);
16  typedef typename GFS::template ConstraintsContainer<Real>::Type CC;
17
18  // <<<3>>> Make grid operator space
19  typedef Example01aLocalOperator LOP;
20  LOP lop(4);                                                               // <= NEW
21  typedef Dune::PDELab::ISTLBCRSMatrixBackend<1,1> MBE;
22  typedef Dune::PDELab::GridOperatorSpace<GFS,GFS,LOP,CC,CC,MBE> GOS;
23  GOS gos(gfs,gfs,lop);
24
```

```
25   // <<<4>>> Select a linear solver backend
26   typedef Dune::PDELab::ISTLBackend_SEQ_BCGS_SSOR LS;
27   LS ls(5000,true);
28
29   // <<<5>>> solve linear problem
30   typedef typename GFS::template VectorContainer<Real>::Type U;
31   U u(gfs,0.0); // initial value
32   typedef Dune::PDELab::StationaryLinearProblemSolver<GOS,LS,U> SLP;
33   SLP slp(gos,u,ls,1e-10);
34   slp.apply();
35
36   // <<<6>>> graphical output
37   typedef Dune::PDELab::DiscreteGridFunction<GFS,U> DGF;
38   DGF udgf(gfs,u);
39   Dune::SubsamplingVTKWriter<GV> vtkwriter(gv,3);                    // <= NEW
40   vtkwriter.addVertexData(new Dune::PDELab::VTKGridFunctionAdapter<DGF>(udgf,"solution"));
41   vtkwriter.write("example01_Q2",Dune::VTKOptions::binaryappended);   // <= NEW
42 }
```

Going Nonlinear

. . . is also easy. Just

- Make a new local operator where the coefficients a and f depend on the solution u, see example01b_operator.
- Use this new local operator in the grid operator space.
- Use class Newton to solve the nonlinear algebraic problem.
- Use a new name for the output file :-).

Listing 5 (File examples/example01b_Q2.hh)**.**

```
1 template<class GV>
2 void example01b_Q2 (const GV& gv)
3 {
4   // <<<1>>> Choose domain and range field type
5   typedef typename GV::Grid::ctype Coord;
6   typedef double Real;
7   const int dim = GV::dimension;
8
9   // <<<2>>> Make grid function space
10  typedef Dune::PDELab::Q22DLocalFiniteElementMap<Coord,Real> FEM;
11  FEM fem;
12  typedef Dune::PDELab::NoConstraints CON;
13  typedef Dune::PDELab::ISTLVectorBackend<1> VBE;
14  typedef Dune::PDELab::GridFunctionSpace<GV,FEM,CON,VBE> GFS;
15  GFS gfs(gv,fem);
16  typedef typename GFS::template ConstraintsContainer<Real>::Type CC;
17
18  // <<<3>>> Make grid operator space
19  typedef Example01bLocalOperator LOP;                               // <= NEW
20  LOP lop(4);
21  typedef Dune::PDELab::ISTLBCRSMatrixBackend<1,1> MBE;
22  typedef Dune::PDELab::GridOperatorSpace<GFS,GFS,LOP,CC,CC,MBE> GOS;
23  GOS gos(gfs,gfs,lop);
24
25  // <<<4>>> Select a linear solver backend
26  typedef Dune::PDELab::ISTLBackend_SEQ_BCGS_SSOR LS;
27  LS ls(5000,true);
28
```

```
29   // <<<5>>> solve nonlinear problem
30   typedef typename GFS::template VectorContainer<Real>::Type U;
31   U u(gfs,2.0); // initial value
32   Dune::PDELab::Newton<GOS,LS,U> newton(gos,u,ls);                    // <= NEW
33   newton.setReassembleThreshold(0.0);
34   newton.setVerbosityLevel(2);
35   newton.setReduction(1e-10);
36   newton.setMinLinearReduction(1e-4);
37   newton.setMaxIterations(25);
38   newton.setLineSearchMaxIterations(10);
39   newton.apply();
40
41   // <<<6>>> graphical output
42   typedef Dune::PDELab::DiscreteGridFunction<GFS,U> DGF;
43   DGF udgf(gfs,u);
44   Dune::SubsamplingVTKWriter<GV> vtkwriter(gv,3);
45   vtkwriter.addVertexData(new Dune::PDELab::VTKGridFunctionAdapter<DGF>(udgf,"solution"));
46   vtkwriter.write("example01b_Q2",Dune::VTKOptions::binaryappended);   // <= NEW
47 }
```

2.3 Constrained Elliptic Model Problem

Problem and Weak Formulation

Consider now the following model problem:

$$\begin{aligned} -\Delta u + au &= f && \text{in } \Omega \subset \mathbb{R}^d, \\ u &= g && \text{on } \Gamma_D \subseteq \partial\Omega, \\ -\nabla u \cdot n &= j && \text{on } \Gamma_N = \partial\Omega \setminus \Gamma_D, \end{aligned}$$

where $a > 0$ when $\Gamma_D = \emptyset$.

Weak formulation. For $w \in H^1(\Omega)$, $w = g$ on Γ_D ("extension of g"), set

$$\begin{aligned} \tilde{U} &= H^1_D(\Omega) = \{u \in H^1(\Omega) \,:\, u|_{\Gamma_D} = 0\} \subset H^1(\Omega), \\ w + \tilde{U} &= \{u \in H^1(\Omega) \,:\, u = w + \tilde{u} \wedge \tilde{u} \in \tilde{U}\}, \end{aligned}$$

Then find

$$u \in w + \tilde{U} \quad : \quad \underbrace{\int_\Omega \nabla u \cdot \nabla v + auv - fv \, dx + \int_{\Gamma_N} jv \, ds}_{r(u,v)} = 0 \qquad \forall v \in \tilde{U}.$$

What is different ?

- The residual form has a new term which is a boundary integral.
  - There will be an additional method on the local operator.
- The problem is solved in an affine subspace. We call $\tilde{U}$ a *constrained space*.

- In the linear case it suffices to solve a problem with homogeneous Dirichlet boundary conditions:

$$\begin{aligned} -\Delta\tilde{u} + a\tilde{u} &= f + \Delta w - aw && \text{in } \Omega \subset \mathbb{R}^d, \\ \tilde{u} &= 0 && \text{on } \Gamma_D \subseteq \partial\Omega, \\ -\nabla\tilde{u} \cdot n &= j + \nabla w \cdot n && \text{on } \Gamma_N = \partial\Omega \setminus \Gamma_D, \end{aligned}$$

  where w is an extension of g to Ω and $u = w + \tilde{u}$.

- In the nonlinear case this is not possible.

Finite Element Spaces in Constrained Case

- Define appropriate finite-dimensional subspace:

$$\tilde{U}_h^k = \left\{ u \in U_h^k \,:\, u|_{\Gamma_D} = 0 \right\} \subset U_h^k.$$

  (Mesh resolves the Dirichlet bopundary Γ_D.

- Provide extension $w \in U_h^k$ with "$w = g$" on Γ_D in an appropriate sense.
- Obviously, $\dim\tilde{U}_h^k < \dim U_h^k$.
- Note: Dirichlet boundary conditions could also be handled by penalty methods. This can be done easily in PDELab but is not shown here.

Realization of Dirichlet Constraints

- Assume $\Phi_{U_h^k}$ is a Lagrange basis:

$$\phi_j(x_i) = \delta_{i,j} \quad \text{where } x_i \text{ are the Lagrange points.}$$

- Construct subspace via basis representation:
  - $\mathcal{I}_{\tilde{U}_h^k} = \left\{ j \in \mathcal{I}_{U_h^k} \,:\, x_j \notin \Gamma_D \right\} \subset \mathcal{I}_{U_h^k}$.
  - $\Phi_{\tilde{U}_h^k} = \left\{ \phi_j \in \Phi_{U_h^k} \,:\, j \in \mathcal{I}_{\tilde{U}_h^k} \right\} \subset \Phi_{U_h^k}$.
  - $\tilde{U}_h^k = \mathrm{span}\Phi_{\tilde{U}_h^k}$.
- For the coefficient space there are two options:
  1. $\tilde{\mathbf{U}} = \mathbb{R}^{\mathcal{I}_{\tilde{U}_h^k}} \not\subset \mathbf{U}$.
  2. $\tilde{\mathbf{U}} = \left\{ \mathbf{u} \in \mathbf{U} \,:\, (\mathbf{u})_j = 0 \;\forall j \in \mathcal{I}_{U_h^k} \setminus \mathcal{I}_{\tilde{U}_h^k} \right\} \subset \mathbf{U}$.
- We choose (2) because
  - $w + \tilde{u}$ is just adding coefficient vectors.
  - Changing Γ_D, e.g. in time-dependent problems is easy.

General Constrained Spaces

- Constrained spaces turn up in a number of other cases:
  - Hanging nodes.
  - Functions with zero average, rigid body modes.
  - Varying polynomial degree in conforming finite elements (p-method).
  - Periodic boundary conditions.
  - Artificial essential boundary conditions or ghost degrees of freedom in parallelization.
- PDELab has a general concept to handle all types of constraints.
- Given U_h with index set $\mathcal{I}_{U_h^k}$, construct a basis of the subspace:
  - Partition index set: $\mathcal{I}_{U_h^k} = \tilde{\mathcal{I}} \cup \bar{\mathcal{I}}$.
  - Construct new basis from given basis:

$$\tilde{\phi}_i = \phi_i + \sum_{j \in \bar{\mathcal{I}}} \omega_{i,j} \phi_j, \qquad i \in \tilde{\mathcal{I}}.$$

  - $\tilde{U}_h$ is spanned by the new basis.
- Constrained space defined by splitting $\tilde{\mathcal{I}} \cup \bar{\mathcal{I}}$ and coefficients $\omega_{i,j}$.

2.4 Example 2

Example 2 Overview

The first example implements model problem (1).

It consists of the following files:

- example02.cc – the file to be compiled, main function.
- example02_bctype.hh – a function giving the splitting $\partial\Omega = \Gamma_D \cup \Gamma_N$.
- example02_bcextension.hh – a function for g and its extension w.
- example02_operator.hh – local operator including inhomogeneous Neumann boundary conditions.
- example02_Q1.hh – driver setting up and solving the problem.

Driver for Solving Constrained Linear Problem About example02_Q1.hh

- Class ConformingDirichletConstraints parametrizes the grid function space with the possibility of having Dirichlet constraints.
- Function constraints assembles constraints (i.e. the splitting $\mathcal{I}_{U_h^k} = \tilde{\mathcal{I}} \cup \bar{\mathcal{I}}$) from a given function.

- Function interpolate initializes a coefficient vector from a given function. At the same time this defines the extension w of g.
- The rest is the same as before.

Listing 6 (File examples/example02_Q1.hh).

```
1 template<class GV>
2 void example02_Q1 (const GV& gv)
3 {
4   // <<<1>>> Choose domain and range field type
5   typedef typename GV::Grid::ctype Coord;
6   typedef double Real;
7   const int dim = GV::dimension;
8
9   // <<<2>>> Make grid function space
10  typedef Dune::PDELab::Q1LocalFiniteElementMap<Coord,Real,dim> FEM;
11  FEM fem;
12  typedef Dune::PDELab::ConformingDirichletConstraints CON;      // constraints class
13  CON con;
14  typedef Dune::PDELab::ISTLVectorBackend<1> VBE;
15  typedef Dune::PDELab::GridFunctionSpace<GV,FEM,CON,VBE> GFS;
16  GFS gfs(gv,fem);
17  typedef BCType<GV> B;                                          // boundary condition type
18  B b(gv);
19  typedef typename GFS::template ConstraintsContainer<Real>::Type CC;
20  CC cc;
21  Dune::PDELab::constraints(b,gfs,cc);                           // assemble constraints
22  std::cout << "constrained␣dofs=" << cc.size()
23            << "␣of␣" << gfs.globalSize() << std::endl;
24
25  // <<<3>>> Make FE function extending Dirichlet boundary conditions
26  typedef typename GFS::template VectorContainer<Real>::Type U;
27  U u(gfs,0.0);
28  typedef BCExtension<GV,Real> G;                                // boundary value + extension
29  G g(gv);
30  Dune::PDELab::interpolate(g,gfs,u);                            // interpolate coefficient vector
31
32  // <<<4>>> Make grid operator space
33  typedef Example02LocalOperator<B> LOP;                         // operator including boundary
34  LOP lop(b);
35  typedef Dune::PDELab::ISTLBCRSMatrixBackend<1,1> MBE;
36  typedef Dune::PDELab::GridOperatorSpace<GFS,GFS,LOP,CC,CC,MBE> GOS;
37  GOS gos(gfs,cc,gfs,cc,lop);
38
39  // <<<5>>> Select a linear solver backend
40  typedef Dune::PDELab::ISTLBackend_SEQ_BCGS_SSOR LS;
41  LS ls(5000,true);
42
43  // <<<6>>> assemble and solve linear problem
44  typedef Dune::PDELab::StationaryLinearProblemSolver<GOS,LS,U> SLP;
45  SLP slp(gos,u,ls,1e-10);
46  slp.apply();
47
48  // <<<7>>> graphical output
49  typedef Dune::PDELab::DiscreteGridFunction<GFS,U> DGF;
50  DGF udgf(gfs,u);
51  Dune::VTKWriter<GV> vtkwriter(gv,Dune::VTKOptions::conforming);
52  vtkwriter.addVertexData(new Dune::PDELab::VTKGridFunctionAdapter<DGF>(udgf,"solution"));
53  vtkwriter.write("example02_Q1",Dune::VTKOptions::binaryappended);
54 }
```

Listing 7 (File examples/example02_bctype.hh).

```
1 /** \brief boundary grid function selecting boundary conditions
```

```
2 * 0 means Neumann, 1 means Dirichlet */
3 template<typename GV>
4 class BCType : public Dune::PDELab::BoundaryGridFunctionBase<
5     Dune::PDELab::BoundaryGridFunctionTraits<GV,int,1,Dune::FieldVector<int,1> >,BCType<GV> >
6 {
7   const GV& gv;
8 public:
9   typedef Dune::PDELab::BoundaryGridFunctionTraits<GV,int,1,Dune::FieldVector<int,1> > Traits;
10
11   //! construct from grid view
12   BCType (const GV& gv_) : gv(gv_) {}
13
14   //! return bc type at point on intersection
15   template<typename I>
16   inline void evaluate (I& i, const typename Traits::DomainType& xlocal,
17                         typename Traits::RangeType& y) const {
18     Dune::FieldVector<typename GV::Grid::ctype,GV::dimension>
19       x = i.geometry().global(xlocal);
20     y = 1; // default is Dirichlet
21     if (x[0]>1.0-1e-6)
22       y = 0; // Neumann
23     return;
24   }
25
26   //! get a reference to the grid view
27   inline const GV& getGridView () {return gv;}
28 };
```

Listing 8 (File examples/example02_bcextension.hh).

```
1 /** \brief A function that defines Dirichlet boundary conditions AND
2     its extension to the interior */
3 template<typename GV, typename RF>
4 class BCExtension
5   : public Dune::PDELab::GridFunctionBase<Dune::PDELab::
6             GridFunctionTraits<GV,RF,1,Dune::FieldVector<RF,1> >, BCExtension<GV,RF> > {
7   const GV& gv;
8 public:
9   typedef Dune::PDELab::GridFunctionTraits<GV,RF,1,Dune::FieldVector<RF,1> > Traits;
10
11   //! construct from grid view
12   BCExtension (const GV& gv_) : gv(gv_) {}
13
14   //! evaluate extended function on element
15   inline void evaluate (const typename Traits::ElementType& e,
16                         const typename Traits::DomainType& xlocal,
17                         typename Traits::RangeType& y) const
18   {
19     const int dim = Traits::GridViewType::Grid::dimension;
20     typedef typename Traits::GridViewType::Grid::ctype ctype;
21     Dune::FieldVector<ctype,dim> x = e.geometry().global(xlocal);
22     if (x[0]<1E-6 && x[1]>0.25 && x[1]<0.5) y = 1.0; else y = 0.0;
23     return;
24   }
25
26   //! get a reference to the grid view
27   inline const GV& getGridView () {return gv;}
28 };
```

alpha_boundary Method

Local operator is extended by a new method alpha_boundary computing the boundary integral.

alpha_boundary has the following signature:

```
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
```

Where the arguments are:

- ig – intersection with domain boundary.
- lfsu_s – local basis $\hat{\phi}_{e,l}$ for trial space on inside element.
- x_s – local coefficients on inside element.
- lfsv_s – local basis $\hat{\psi}_{e,l}$ for test space on inside element.
- r_s – local contribution to residual on inside element.

Listing 9 (File examples/example02_operator.hh).

```
#include<dune/grid/common/genericreferenceelements.hh>
#include<dune/grid/common/quadraturerules.hh>
#include<dune/pdelab/common/geometrywrapper.hh>
#include<dune/pdelab/localoperator/pattern.hh>
#include<dune/pdelab/localoperator/flags.hh>

/** a local operator for solving the equation
 *
 *   - \Delta u + a*u = f    in \Omega
 *                  u = g    on \Gamma_D\subseteq\partial\Omega
 *  -\nabla u \cdot n = j    on \Gamma_N = \partial\Omega\setminus\Gamma_D
 *
 * with conforming finite elements on all types of grids in any dimension
 *
 * \tparam B a function indicating the type of boundary condition
 */
template<class B>
class Example02LocalOperator :
  public Dune::PDELab::NumericalJacobianApplyVolume<Example02LocalOperator<B> >,
  public Dune::PDELab::NumericalJacobianVolume<Example02LocalOperator<B> >,
  public Dune::PDELab::NumericalJacobianApplyBoundary<Example02LocalOperator<B> >,
  public Dune::PDELab::NumericalJacobianBoundary<Example02LocalOperator<B> >,
  public Dune::PDELab::FullVolumePattern,
  public Dune::PDELab::LocalOperatorDefaultFlags
{
public:
  // pattern assembly flags
  enum { doPatternVolume = true };

  // residual assembly flags
  enum { doAlphaVolume = true };
  enum { doAlphaBoundary = true };                                  // assemble boundary

  Example02LocalOperator (const B& b_, unsigned int intorder_=2)  // needs boundary cond. type
    : b(b_), intorder(intorder_)
  {}

  // 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
  {
    // extract some types
    typedef typename LFSU::Traits::LocalFiniteElementType::
      Traits::LocalBasisType::Traits::DomainFieldType DF;
```

```
45      typedef typename LFSU::Traits::LocalFiniteElementType::
46        Traits::LocalBasisType::Traits::RangeFieldType RF;
47      typedef typename LFSU::Traits::LocalFiniteElementType::
48        Traits::LocalBasisType::Traits::JacobianType JacobianType;
49      typedef typename LFSU::Traits::LocalFiniteElementType::
50        Traits::LocalBasisType::Traits::RangeType RangeType;
51      typedef typename LFSU::Traits::SizeType size_type;
52
53      // dimensions
54      const int dim = EG::Geometry::dimension;
55      const int dimw = EG::Geometry::dimensionworld;
56
57      // select quadrature rule
58      Dune::GeometryType gt = eg.geometry().type();
59      const Dune::QuadratureRule<DF,dim>&
60        rule = Dune::QuadratureRules<DF,dim>::rule(gt,intorder);
61
62      // loop over quadrature points
63      for (typename Dune::QuadratureRule<DF,dim>::const_iterator
64             it=rule.begin(); it!=rule.end(); ++it)
65        {
66          // evaluate basis functions on reference element
67          std::vector<RangeType> phi(lfsu.size());
68          lfsu.localFiniteElement().localBasis().evaluateFunction(it->position(),phi);
69
70          // compute u at integration point
71          RF u=0.0;
72          for (size_type i=0; i<lfsu.size(); i++)
73            u += x[i]*phi[i];
74
75          // evaluate gradient of basis functions on reference element
76          std::vector<JacobianType> js(lfsu.size());
77          lfsu.localFiniteElement().localBasis().evaluateJacobian(it->position(),js);
78
79          // transform gradients from reference element to real element
80          const Dune::FieldMatrix<DF,dimw,dim>
81            jac = eg.geometry().jacobianInverseTransposed(it->position());
82          std::vector<Dune::FieldVector<RF,dim> > gradphi(lfsu.size());
83          for (size_type i=0; i<lfsu.size(); i++)
84            jac.mv(js[i][0],gradphi[i]);
85
86          // compute gradient of u
87          Dune::FieldVector<RF,dim> gradu(0.0);
88          for (size_type i=0; i<lfsu.size(); i++)
89            gradu.axpy(x[i],gradphi[i]);
90
91          // evaluate parameters;
92          Dune::FieldVector<RF,dim>
93            globalpos = eg.geometry().global(it->position());
94          RF f = 0;
95          RF a = 0;
96
97          // integrate grad u * grad phi_i + a*u*phi_i - f phi_i
98          RF factor = it->weight()*eg.geometry().integrationElement(it->position());
99          for (size_type i=0; i<lfsu.size(); i++)
100           r[i] += ( gradu*gradphi[i] + a*u*phi[i] - f*phi[i] )*factor;
101       }
102   }
103
104   // boundary integral
105   template<typename IG, typename LFSU, typename X, typename LFSV, typename R>
106   void alpha_boundary (const IG& ig, const LFSU& lfsu_s, const X& x_s,
107                        const LFSV& lfsv_s, R& r_s) const
108   {
```

```
109     // some types
110     typedef typename LFSV::Traits::LocalFiniteElementType::
111       Traits::LocalBasisType::Traits::DomainFieldType DF;
112     typedef typename LFSV::Traits::LocalFiniteElementType::
113       Traits::LocalBasisType::Traits::RangeFieldType RF;
114     typedef typename LFSV::Traits::LocalFiniteElementType::
115       Traits::LocalBasisType::Traits::RangeType RangeType;
116     typedef typename LFSV::Traits::SizeType size_type;
117
118     // dimensions
119     const int dim = IG::dimension;
120
121     // select quadrature rule for face
122     Dune::GeometryType gtface = ig.geometryInInside().type();
123     const Dune::QuadratureRule<DF,dim-1>&
124       rule = Dune::QuadratureRules<DF,dim-1>::rule(gtface,intorder);
125
126     // loop over quadrature points and integrate normal flux
127     for (typename Dune::QuadratureRule<DF,dim-1>::const_iterator it=rule.begin();
128      it!=rule.end(); ++it)
129       {
130         // evaluate boundary condition type
131         typename B::Traits::RangeType bctype;
132         b.evaluate(ig,it->position(),bctype);
133
134         // skip rest if we are on Dirichlet boundary
135         if (bctype>0) continue;
136
137         // position of quadrature point in local coordinates of element
138         Dune::FieldVector<DF,dim> local = ig.geometryInInside().global(it->position());
139
140         // evaluate basis functions at integration point
141         std::vector<RangeType> phi(lfsv_s.size());
142         lfsu_s.localFiniteElement().localBasis().evaluateFunction(local,phi);
143
144         // evaluate u (e.g. flux may depend on u)
145         RF u=0.0;
146         for (size_type i=0; i<lfsu_s.size(); i++)
147           u += x_s[i]*phi[i];
148
149         // evaluate flux boundary condition
150         Dune::FieldVector<RF,dim>
151           globalpos = ig.geometry().global(it->position());
152         RF j;
153         if (globalpos[1]<0.5) j = 1.0; else j = -1.0; // some outflow
154
155         // integrate j
156         RF factor = it->weight()*ig.geometry().integrationElement(it->position());
157         for (size_type i=0; i<lfsv_s.size(); i++)
158           r_s[i] += j*phi[i]*factor;
159       }
160   }
161
162 private:
163   const B& b;
164   unsigned int intorder;
165 };
```

Figure 1 shows visualizations of the results computed with example01.

Summary: Implementation of Mathematical Concepts

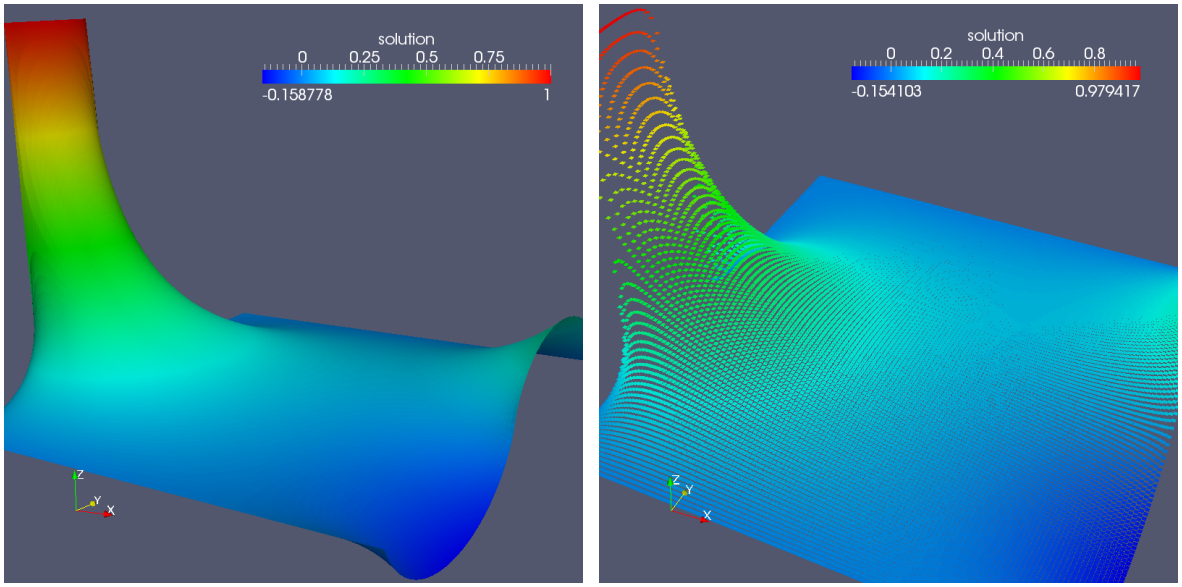


Figure 2: Result for example 2 computed with Q_1 elements on the right. Same problem solved with cell-centered finite volume method in example 4.

- A *grid function space* represents an unconstrained finite-dimensional function space on a grid:
  - gfs $\quad\leftrightarrow\quad$ $U_h(E_h^0)$.
- A grid function space with a *constraints container* represents a constrained finite-dimensional function space:
  - gfs + cc $\quad\leftrightarrow\quad$ $\tilde{U}_h(E_h^0)$.
- A *vector container* represents a vector space of coefficients:
  - U u $\quad\leftrightarrow\quad$ $\mathbf{u} \in \mathbf{U}$.
- A grid function space and a coefficient vector represent a function in a finite-dimensional function space:
  - gfs + u $\quad\leftrightarrow\quad$ $u_h \in U_h(E_h^0)$.
- A *grid operator space* realizes computations with residual forms:
  - gos.residual(u,r) $\quad\leftrightarrow\quad$ $\mathbf{r} = \mathcal{R}(\mathbf{u})$.
  - gos.jacobian(u,A) $\quad\leftrightarrow\quad$ $\mathbf{A} = \nabla\mathcal{R}(\mathbf{u})$.
- A *matrix container* represents a linear map (not shown yet):
  - Matrix A $\quad\leftrightarrow\quad$ $\mathbf{A} \in L(\mathbf{U}, \mathbf{V})$.

2.5 Cell-centered Finite Volumes

Problem

We wish to demonstrate that PDELab allows also the implementation of Finite Volume methods.

Consider again the elliptic model problem:

$$\begin{aligned} -\Delta u + au &= q && \text{in } \Omega \subset \mathbb{R}^d, \\ u &= g && \text{on } \Gamma_D \subseteq \partial\Omega, \\ -\nabla u \cdot n &= j && \text{on } \Gamma_N = \partial\Omega \setminus \Gamma_D, \end{aligned}$$

where $a > 0$ when $\Gamma_D = \emptyset$.

We show how to formulate this method in the framework of residual forms.

Discrete Weak Formulation I

For a grid E_h^0 define $W_h = \{u \in L_2(\Omega) : u|_{\Omega_e} = \text{const } \forall e \in E_h^0\}$. $v \in W_h$:

$$\begin{aligned} \sum_{e\in E_h^0} \int_{\Omega_e} (-\Delta u + au - q)v \, dx &= \sum_{e\in E_h^0} \left\{ \int_{\Omega_e} (au - q)v \, dx - \int_{\partial\Omega_e} (\nabla u \cdot n)v \, ds \right\} \\ &= \sum_{e\in E_h^0} \left\{ \int_{\Omega_e} (au - q)v \, dx - \int_{\partial\Omega_e\cap\Omega_e} (\nabla u \cdot n)v \, ds \right. \\ &\qquad \left. - \int_{\partial\Omega_e\cap\Gamma_D} (\nabla u \cdot n)v \, ds + \int_{\partial\Omega_e\cap\Gamma_N} jv \, ds \right\} \\ &= \sum_{e\in E_h^0} \int_{\Omega_e} (au - q)v \, dx - \sum_{f\in E_h^1} \int_{\Omega_f} (\nabla u \cdot n_f)[v] \, ds \\ &\qquad - \sum_{f\in B_h^1} \int_{\Omega_f\cap\Gamma_D} (\nabla u \cdot n)v \, ds + \sum_{f\in B_h^1} \int_{\Omega_f\cap\Gamma_N} jv \, ds. \end{aligned}$$

E_h^1: interior faces, B_h^1: boundary faces, n_f pointing from e_f^- to e_f^+, x_f^- center of e_f^-, x_f^+ center of e_f^+, $[v]|_f = v(x_f^-) - v(x_f^+)$.

Discrete Weak Formulation II

Approximation with difference quotient and midpoint rule:

$$\begin{aligned} r_h^{\mathrm{FV}}(u_h, v) = &\sum_{e \in E_h^0} \big(a(x_e)u_h(x_e) - q(x_e)\big)v(x_e)|\Omega_e| \\ &- \sum_{f \in E_h^1} \frac{u_h(x_f^+) - u_h(x_f^-)}{\|x_f^+ - x_f^-\|}\big(v(x_f^-) - v(x_f^+)\big)|\Omega_f| \\ &- \sum_{f \in B_h^1; \Omega_f \subseteq \Gamma_D} \frac{g(x_f) - u_h(x_f^-)}{\|x_f - x_f^-\|} v(x_f^-)|\Omega_f| \\ &+ \sum_{f \in B_h^1; \Omega_f \subseteq \Gamma_N} j(x_f)v(x_f)|\Omega_f|. \end{aligned}$$

The discrete problem then reads

$$u_h \in W_h : \quad r_h^{\mathrm{FV}}(u_h, v) = 0 \qquad \forall v \in W_h.$$

The interior face term is new! CCFV has no constraints!

2.6 Example 4

Example 4 Overview

Example 4 implements the cell-centered finite volume method for the elliptiv model problem.

It works only on axiparallel, cube grids.

It consists of the following files:

- example04.cc – the file to be compiled, main function.
- example04.hh – driver setting up and solving the problem.
- example04_operator.hh – local operator for cell centered finite volumes.

The Driver

New things in this driver are:

- Use of element-wise constant functions.
- The new local operator shown below.
- Use CG preconditioned with AMG as solver (just for fun).
- Write cell data in VTKWriter.

Listing 10 (File examples/example04.hh).

```
1 template<class GV>
2 void example04 (const GV& gv)
3 {
4   // <<<1>>> Choose domain and range field type
5   typedef typename GV::Grid::ctype Coord;
```

```
6   typedef double Real;
7   const int dim = GV::dimension;
8
9   // <<<2>>> Make grid function space
10  typedef Dune::PDELab::P0LocalFiniteElementMap<Coord,Real,dim> FEM;
11  FEM fem(Dune::GeometryType::cube);                          // supply element type for P0
12  typedef Dune::PDELab::NoConstraints CON;                    // we have no constraints!
13  typedef Dune::PDELab::ISTLVectorBackend<1> VBE;
14  typedef Dune::PDELab::GridFunctionSpace<GV,FEM,CON,VBE> GFS;
15  GFS gfs(gv,fem);
16
17  // <<<3>>> Make grid operator space
18  typedef Example04LocalOperator LOP;                         // our new operator
19  LOP lop;
20  typedef Dune::PDELab::ISTLBCRSMatrixBackend<1,1> MBE;
21  typedef Dune::PDELab::EmptyTransformation CC;
22  typedef Dune::PDELab::GridOperatorSpace<GFS,GFS,LOP,CC,CC,MBE> GOS;
23  GOS gos(gfs,gfs,lop);
24
25  // <<<4>>> Select a linear solver backend
26  typedef Dune::PDELab::ISTLBackend_SEQ_CG_AMG_SSOR<GFS> LS;
27  LS ls(2,100,2);
28
29  // <<<5>>> assemble and solve linear problem
30  typedef typename GFS::template VectorContainer<Real>::Type U;
31  U u(gfs,0.0);
32  Dune::PDELab::StationaryLinearProblemSolver<GOS,LS,U> slp(gos,u,ls,1e-10);
33  slp.apply();
34
35  // <<<6>>> graphical output
36  typedef Dune::PDELab::DiscreteGridFunction<GFS,U> DGF;
37  DGF udgf(gfs,u);
38  Dune::VTKWriter<GV> vtkwriter(gv,Dune::VTKOptions::conforming);
39  vtkwriter.addCellData(new Dune::PDELab::VTKGridFunctionAdapter<DGF>(udgf,"solution"));
40  vtkwriter.write("example04",Dune::VTKOptions::binaryappended);
41 }
```

alpha_skeleton Method

Local operator is extended by a new method alpha_skeleton called once for each interior intersection.

Optionally it can be called from each element touching the intersection.

alpha_boundary has the following signature:

```
template<typename IG, typename LFSU, typename X, typename LFSV,
         typename R>
void alpha_skeleton (const IG& ig,
            const LFSU& lfsu_s, const X& x_s, const LFSV& lfsv_s,
            const LFSU& lfsu_n, const X& x_n, const LFSV& lfsv_n,
            R& r_s, R& r_n) const
```

Where the arguments are:

- ig – interior intersection.
- lfsu_s, lfsu_n – trial space in self (inside) and neighbor (outside).
- x_s, x_n – local coefficients in self and neighbor.

- lfsv_s , lfsv_n – test space in self and neighbor.
- r_s , r_n – local contribution to residual in self and neighbor.

Listing 11 (File examples/example04_operator.hh).

```
1#include<dune/grid/common/genericreferenceelements.hh>
2#include<dune/grid/common/quadraturerules.hh>
3#include<dune/pdelab/common/geometrywrapper.hh>
4#include<dune/pdelab/localoperator/pattern.hh>
5#include<dune/pdelab/localoperator/flags.hh>
6
7/** a local operator for solving the equation
8 *
9 *   - \Delta u + a*u = f    in \Omega
10 *                  u = g    on \Gamma_D\subseteq\partial\Omega
11 *  -\nabla u \cdot n = j    on \Gamma_N = \partial\Omega\setminus\Gamma_D
12 *
13 * with cell-centered finite volumes on axiparallel, structured grids
14 *
15 * \tparam B a function indicating the type of boundary condition
16 * \tparam G a function for the values of the Dirichlet boundary condition
17 */
18class Example04LocalOperator :     // implement jacobian evaluation in base classes
19  public Dune::PDELab::NumericalJacobianApplyVolume<Example04LocalOperator>,
20  public Dune::PDELab::NumericalJacobianVolume<Example04LocalOperator>,
21  public Dune::PDELab::NumericalJacobianApplySkeleton<Example04LocalOperator>,
22  public Dune::PDELab::NumericalJacobianSkeleton<Example04LocalOperator>,
23  public Dune::PDELab::NumericalJacobianApplyBoundary<Example04LocalOperator>,
24  public Dune::PDELab::NumericalJacobianBoundary<Example04LocalOperator>,
25  public Dune::PDELab::FullSkeletonPattern,                    // matrix entries skeleton
26  public Dune::PDELab::FullVolumePattern,
27  public Dune::PDELab::LocalOperatorDefaultFlags
28{
29public:
30  // pattern assembly flags
31  enum { doPatternVolume = true };
32  enum { doPatternSkeleton = true };
33
34  // residual assembly flags
35  enum { doAlphaVolume  = true };
36  enum { doAlphaSkeleton  = true };                            // assemble skeleton term
37  enum { doAlphaBoundary  = true };
38
39  // volume integral depending on test and ansatz functions
40  template<typename EG, typename LFSU, typename X, typename LFSV, typename R>
41  void alpha_volume (const EG& eg, const LFSU& lfsu, const X& x, const LFSV& lfsv,
42                     R& r) const
43  {
44    // domain and range field type
45    typedef typename LFSV::Traits::LocalFiniteElementType::
46      Traits::LocalBasisType::Traits::DomainFieldType DF;
47    typedef typename LFSV::Traits::LocalFiniteElementType::
48      Traits::LocalBasisType::Traits::RangeFieldType RF;
49    const int dim = EG::Geometry::dimension;
50
51    // evaluate reaction term
52    Dune::FieldVector<DF,dim> center = eg.geometry().center();
53    RF a = 0.0;
54    RF f = 0.0;
55
56    r[0] = (a*x[0]-f)*eg.geometry().volume();
57  }
58
59  // skeleton integral depending on test and ansatz functions
```

```
60  // each face is only visited ONCE!
61  template<typename IG, typename LFSU, typename X, typename LFSV, typename R>
62  void alpha_skeleton (const IG& ig,
63                       const LFSU& lfsu_s, const X& x_s, const LFSV& lfsv_s,
64                       const LFSU& lfsu_n, const X& x_n, const LFSV& lfsv_n,
65                       R& r_s, R& r_n) const
66  {
67    // domain and range field type
68    typedef typename LFSU::Traits::LocalFiniteElementType::
69      Traits::LocalBasisType::Traits::DomainFieldType DF;
70    typedef typename LFSU::Traits::LocalFiniteElementType::
71      Traits::LocalBasisType::Traits::RangeFieldType RF;
72    const int dim = IG::dimension;
73
74    // distance between cell centers in global coordinates
75    Dune::FieldVector<DF,dim> inside_global = ig.inside()->geometry().center();
76    Dune::FieldVector<DF,dim> outside_global = ig.outside()->geometry().center();
77    inside_global -= outside_global;
78    RF distance = inside_global.two_norm();
79
80    // face geometry
81    RF face_volume = ig.geometry().volume();
82
83    // diffusive flux for both sides
84    r_s[0] -= (x_n[0]-x_s[0])*face_volume/distance;
85    r_n[0] += (x_n[0]-x_s[0])*face_volume/distance;
86  }
87
88  // skeleton integral depending on test and ansatz functions
89  // Here Dirichlet and Neumann boundary conditions are evaluated
90  template<typename IG, typename LFSU, typename X, typename LFSV, typename R>
91  void alpha_boundary (const IG& ig,
92                       const LFSU& lfsu_s, const X& x_s, const LFSV& lfsv_s,
93                       R& r_s) const
94  {
95    // domain and range field type
96    typedef typename LFSU::Traits::LocalFiniteElementType::
97      Traits::LocalBasisType::Traits::DomainFieldType DF;
98    typedef typename LFSU::Traits::LocalFiniteElementType::
99      Traits::LocalBasisType::Traits::RangeFieldType RF;
100   const int dim = IG::dimension;
101
102   // face geometry
103   Dune::FieldVector<DF,dim> face_center = ig.geometry().center();
104   RF face_volume = ig.geometry().volume();
105
106   // evaluate boundary condition type
107   int b;
108   if (face_center[0]>1.0-1e-6)
109     b = 0; // Neumann
110   else
111     b = 1; // Dirichlet
112
113   // do things depending on boundary condition type
114   if (b==0) // Neumann boundary
115     {
116       RF j; if (face_center[1]<0.5) j = 1.0; else j = -1.0;
117       r_s[0] += j*face_volume;
118       return;
119     }
120
121   if (b==1) // Dirichlet boundary
122     {
123       RF g;
```

```
124           if (face_center[0]<1E-6 && face_center[1]>0.25 && face_center[1]<0.5)
125             g = 1.0;
126           else
127             g = 0.0;
128           Dune::FieldVector<DF,dim> inside_global = ig.inside()->geometry().center();
129           inside_global -= face_center;
130           RF distance = inside_global.two_norm();
131           r_s[0] -= (g-x_s[0])*face_volume/distance;
132           return;
133         }
134     }
135 };
```

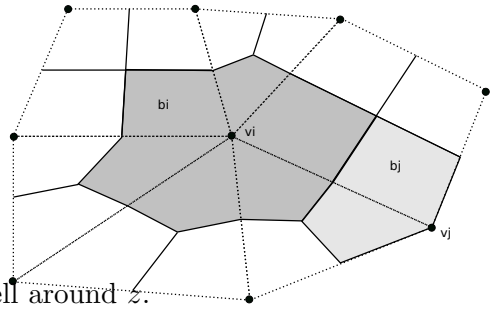
2.7 Other Schemes to Solve the Model Problem

Vertex-centered Finite Volumes This method uses a discontinuous test space that is constant on "cells". Let us introduce some notation to define this formally.

$E_h^d = \{z_0, \ldots, z_{N_h^d - 1}\}$ are the vertices of the mesh (entities of codimension d).

x_z is the position of $z \in E_h^d$.

$C_h = \{c_0, \ldots, c_{N_h^d - 1}\}$ are "cells" around each vertex of the mesh.



Ω_c is the domain of $c \in C_h$ and $x_c = x_z$ when c is the cell around z.

Define the *discontinuous* test function space:

$$V_h^0 = \{v \in L_2(\Omega) \,|\, \forall c \in C_h : v|_{\Omega_c} = \text{const}\},$$
$$\tilde{V}_h^0 = \{v \in V_h \,|\, \forall z \in E_h^d, x_z \in \Gamma_D : v(x_z) = 0\}.$$

The discontinuities are located on the *skeleton* which is given by:

$$\Gamma_h^{\text{int}} = \{\gamma_{e,c,c'} \,|\, \gamma_{e,c,c'} = \Omega_e \cap \partial\Omega_c \cap \partial\Omega_{c'}\}$$

and the boundary faces are given by

$$\Gamma_h^{\text{ext}} = \{\gamma_{e,c} \,|\, \gamma_{e,c} = \partial\Omega_e \cap \partial\Omega_c \cap \partial\Omega\}.$$

For $\gamma \in \Gamma_h^{\text{int}}$, $\nu_\gamma(x)$ is the *unique* unit normal vector to γ in point x.

Similarly, for $\gamma \in \Gamma_h^{\text{ext}}$, $\nu_\gamma(x)$ is the unit outer normal vector.

The jump of a function $v \in V_h$ in $x \in \gamma \in \Gamma_h^{\text{int}}$ given by

$$[v]_\gamma(x) = \lim_{\varepsilon \to 0-} v(x + \varepsilon\nu_\gamma(x)) - \lim_{\varepsilon \to 0+} v(x + \varepsilon\nu_\gamma(x)) \quad .$$

The discrete problem then reads as follows. Find $u_h \in w_h + \tilde{U}_h^1$ s.t.

$$\underbrace{-\sum_{\gamma\in\Gamma_h^{\text{int}}}\int_\gamma \nabla u_h\cdot\nu_\gamma\,[v]\,ds+\sum_{\substack{\gamma\in\Gamma_h^{\text{ext}},\\ \gamma\subseteq\Gamma_N}}\int_\gamma jv\,ds \quad -\int_\Omega fv\,dx}_{=r_h^{\text{FE}}(u_h,v)}=0,\quad \forall v\in\tilde{V}_h^0. \tag{5}$$

$\tilde{U}_h^1$ is the linear conforming finite element space and w_h is defined as before.

Typically, the integrals are evaluated with low order quadrature rules such as the midpoint rule.

This gives a simple example with non-conforming residual form and different trial and test functions.

Discontinuous Galerkin Let $k : E_h^0 \to \mathbb{N}_0$ be a function that associates an nonnegative integer with each element.

Define the discrete function space

$$W_h^k = \{u \in L_2(\Omega)\,|\,\forall e \in E_h^0 : u|_{\Omega_e} \in P_{k(e)}\}.$$

For any $x \in \Omega_f, f \in E_h^1$, define the jump of a function $u \in W_h^k$:

$$[u]_f(x) = \lim_{\epsilon\to 0-} u(x+\epsilon\nu_f) - \lim_{\epsilon\to 0+} u(x+\epsilon\nu_f).$$

For any $x \in \Omega_f, f \in E_h^1$, define the average of a function $u \in W_h^k$:

$$\langle u\rangle_f(x) = \frac{1}{2}\left(\lim_{\epsilon\to 0-} u(x+\epsilon\nu_f) + \lim_{\epsilon\to 0+} u(x+\epsilon\nu_f)\right).$$

The discrete problem for the OBB method [OBB98] then reads

$$u_h \in W_h^k \quad : \quad r_h^{\text{OBB}}(u_h, v) = 0 \qquad \forall v \in W_h^k,$$

where

$$\begin{aligned} r_h^{\text{OBB}}(u,v) = &\sum_{e\in E_h^0}\int_{\Omega_e}\nabla u\cdot\nabla v - fv\,dx \\ &+\sum_{f\in E_h^1}\int_{\Omega_f}\langle\nabla v\cdot\nu_f\rangle[u]_f - [v]_f\langle\nabla u\cdot\nu_f\rangle\,ds \\ &+\sum_{\substack{b\in B_h^1\\ \Omega_b\subseteq\Gamma_D}}\int_{\Omega_b}(\nabla v\cdot\nu_f)(u-g) - v(\nabla u\cdot\nu_f)\,ds + \sum_{\substack{b\in B_h^1\\ \Omega_b\subseteq\Gamma_N}}\int_{\Omega_b} jv\,ds. \end{aligned}$$

Note the seperation into volume, skeleton and boundary terms.

Crouzeix-Raviart Here we use the following discrete spaces:

$$X_h = \{u \in L_2(\Omega) \,|\, u|_{\Omega_e} \in P_1 \text{ and } u \text{ continuous at face centers}\},$$
$$\tilde{X}_h = \{u \in X_h \,|\, \text{“}u = 0\text{” on } \Gamma_D\}.$$

The discrete problem then reads

$$u_h \in w_h + \tilde{X}_h \quad : \quad r_h^{\mathrm{CR}}(u_h, v) = 0 \qquad \forall v \in \tilde{X}_h,$$

where

$$r_h^{\mathrm{CR}}(u, v) = \sum_{e \in E_h^0} \int_{\Omega_e} \nabla u \cdot \nabla v \, dx + \sum_{\substack{b \in B_h^1 \\ \Omega_b \subseteq \Gamma_N}} \int_{\Omega_b} jv \, ds - \sum_{e \in E_h^0} \int_{\Omega_e} fv \, dx.$$

Again $w_h \in X_h$ such that “$w_h = g$” on Γ_D.

2.8 Properties of the Residual Form

Definition 12 (Weighted Residual Formulation)**.** We claim that all problems we ever want to solve can be written in the form

$$\text{Find } u_h \in w_h + \tilde{U}_h : \qquad r_h(u_h, v) = 0 \qquad \forall v \in \tilde{V}_h. \tag{6}$$

Where:

- $\tilde{U}_h \subseteq U_h$, $\tilde{V}_h \subseteq V_h$ are finite-dimensional function spaces and corresponding subspaces.
- Affine shift: $u_h \in w_h + \tilde{U}_h$ for a given $w_h \in U_h$ means $u_h = w_h + \tilde{u}_h$ for some $\tilde{u}_h \in \tilde{U}_h$.
- $r_h : U_h \times V_h \to \mathbb{K}$ is the *residual form*.
  - r_h may be *nonlinear* in its first argument.
  - r_h *is always linear* in its second argument.
  - r_h may depend on the grid in non-conforming methods.
- We assume that this problem has a unique solution. □

The examples above imply the following properties of r_h.

Property 13 (Splitting)**.** r_h can be split into element, skeleton and boundary sums

$$r_h(u, v) = \sum_{e \in E_h^0} r_{h,e}^{\mathrm{vol}}(u, v) + \sum_{f \in E_h^1} r_{h,f}^{\mathrm{skel}}(u, v) + \sum_{b \in B_h^1} r_{h,b}^{\mathrm{bnd}}(u, v)$$

r_h can be split into a part depending on u and a part independent of u:

$$r_h(u, v) = \alpha_h(u, v) + \lambda_h(v).$$

Together we obtain

$$\begin{aligned} r_h(u,v) = &\sum_{e\in E_h^0} \alpha_{h,e}^{\mathrm{vol}}(u,v) + \sum_{f\in E_h^1} \alpha_{h,f}^{\mathrm{skel}}(u,v) + \sum_{b\in B_h^1} \alpha_{h,b}^{\mathrm{bnd}}(u,v) \\ &+ \sum_{e\in E_h^0} \lambda_{h,e}^{\mathrm{vol}}(v) + \sum_{f\in E_h^1} \lambda_{h,f}^{\mathrm{skel}}(v) + \sum_{b\in B_h^1} \lambda_{h,b}^{\mathrm{bnd}}(v). \end{aligned} \tag{7}$$

□

Property 14 (Linearity)**.** r_h, as well as, $r_{h,e}^{\mathrm{vol}}$, $r_{h,f}^{\mathrm{skel}}$ and $r_{h,b}^{\mathrm{bnd}}$ are linear in their second argument.

As a consequence we have $r_h(u,0) = 0$. □

Property 15 (Localization)**.** We assume that the following holds:

$$\begin{aligned} e \in E_h^0 : &\qquad r_{h,e}^{\mathrm{vol}}(u,v) = r_{h,e}^{\mathrm{vol}}(\chi_{\Omega_e} u, \chi_{\Omega_e} v), \\ f \in E_h^1 : &\qquad r_{h,f}^{\mathrm{skel}}(u,v) = r_{h,f}^{\mathrm{skel}}(\chi_{\Omega_{l(f)}\cup\Omega_{r(f)}} u, \chi_{\Omega_{l(f)}\cup\Omega_{r(f)}} v), \\ b \in B_h^1 : &\qquad r_{h,b}^{\mathrm{bnd}}(u,v) = r_{h,b}^{\mathrm{bnd}}(\chi_{\Omega_{l(b)}} u, \chi_{\Omega_{l(b)}} v). \end{aligned}$$

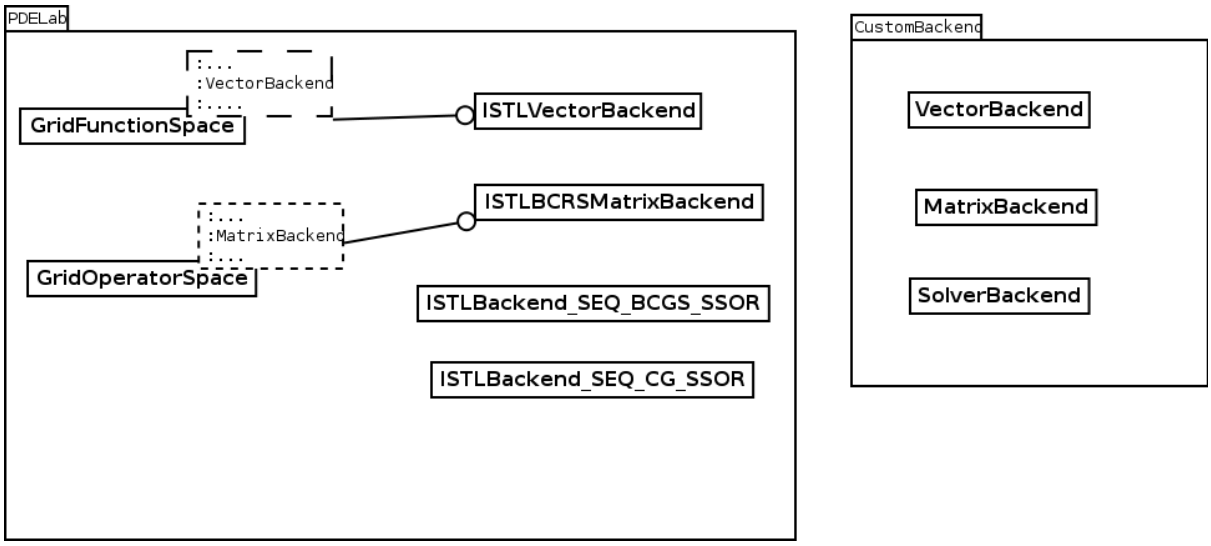
where $\chi_\omega(x) : \omega \to \{0,1\}$ is the characteristic function of ω.

This is a consequence of $r_{h,e}^{\mathrm{vol}}$, $r_{h,e}^{\mathrm{skel}}$ and $r_{h,e}^{\mathrm{bnd}}$ being integrals over an element, a face or a boundary face.

3 PDELab backends

- Linear algebra is decoupled from the discretization.
- Provides unique access interface for matrices, vectors and solvers.
- (Often) easily extensible for use of other libraries.
- Minimal knowledge of the underlying libraries required.

PDELab backend in UML



3.1 Interface

3.1.1 The Vector Backend

- It is a class not a namespace to allow template parameterization!
- Provides the actual container type as **template**<**class** T, **class** E> Vector.
- Associated type size_type used for index access.
- Const and mutable data access with access(Vector<T,E>& cont, size_type i)

Sample Vector Container

- Template parameter T is the type of the grid function used.
- Template parameter E is the value type (e.g. double, float).

```
template<typename T, typename E>
class VectorContainer
{
  public:
```

```
    // The stored element type
    typedef E ElementType;
    // The backend we are associated with
    typedef ISTLVectorBackend<BLOCKSIZE> Backend;

    //constructor with grid function space t
    VectorContainer (const T& t_);

    //constructor with grid function space t, initial value e
    VectorContainer (const T& t_, const E& e);

    // assignment
    VectorContainer& operator= (const E& e);
  };
```

3.1.2 The Matrix Backend

- It is a class not a namespace to allow template parameterization!
- Provides the actual container type as **template**<**class** T, **class** E> Matrix.
- Provides method clearRow(RI i, C& c) for setting the values of row with index i of container c to zero.
- Const and mutable data access with access(Matrix<T,E>& cont, size_type i, size_type j)

The Matrix Container

- Template parameter T is the type of the grid function used.
- Template parameter E is the value type (e.g. double, float).

```
template<class T, class E>
class MatrixContainer{
  public:
    // The stored element type
    typedef typename E ElementType;
    // The backend we are associated with
    typedef ISTLMatrixBackend<ROWBLOCKSIZE,COLBLOCKSIZE> Backend;

    //constructor with grid operator space t
    // Needs to setup the whole sparsity pattern.
    MatrixContainer (const T& t_);

    // assignment
    MatrixContainer& operator= (const E& e);
};
```

Sparse Matrix Setup in PDELab

- Setting up a sparse matrix is a two step procedure:
  1. Setting up the sparsity pattern.
  2. Filling the matrix with the values.

- First step is performed by the contructor of the matrix container.
- Second step is performed by the member function jacobian of the grid operator space.

3.1.3 The Linear Solver Backend

- Provides vector norm method needed by nonlinear PDELab solvers:

```
template<class V>
typename V::ElementType norm(const V& v) const;
```

- Member function result() returns statistics about the solution phase in an instance of the template<class T> LinearSolverResult class template.
- Use member function

```
template<class M, class V, class W>
apply(M& A, V& z, W& r, typename V::ElementType reduction);
```

  to (iteratively) solve $Az = r$ until the prescribed relative defect reduction is achieved.
- For ISTL it hides a lot of the compexity that heavy usage of templates imposes on the user.

3.2 The Iterative Solver Template Library

3.2.1 Block Structure in FE Matrices

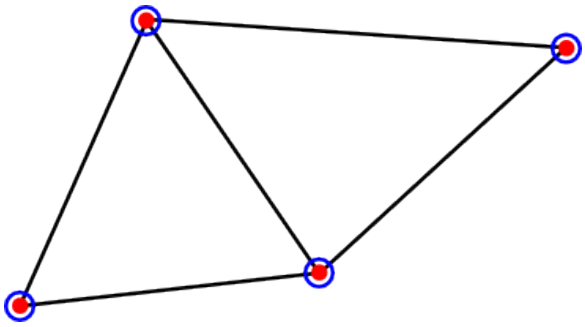
PDE Systems

We have several unknowns and therefore several grid functions when dealing with PDE Systems. For the simple case that each function has the same structure we can use the following special ordering and blocking approaches:

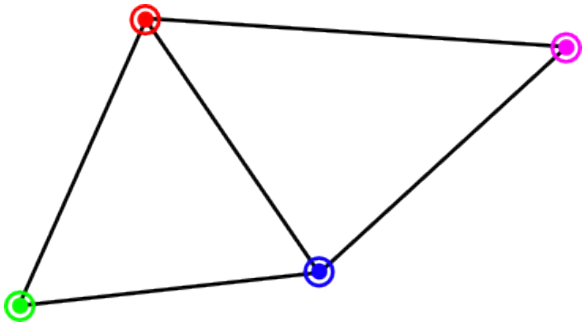
- Equation-wise blocking: All degrees of freedom associated with the same unknown are blocked together.
- Point-wise blocking: All degrees of freedom associated with the same point (or element) are blocked together.
- No blocking is used.

Currently the last two approaches are supported by PDELab with the ISTL backend. In the pictures below the degrees of freedom are visualized using circles at the positions/entities they are associated with. Circles with the same colour are blocked together. In the the entries in the resulting matrix are dense and sparse matrices for the equation-wise and point-wise blocking, respectively.

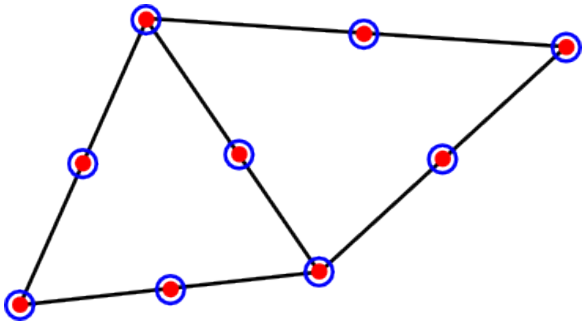
Some Examples for Block Structure



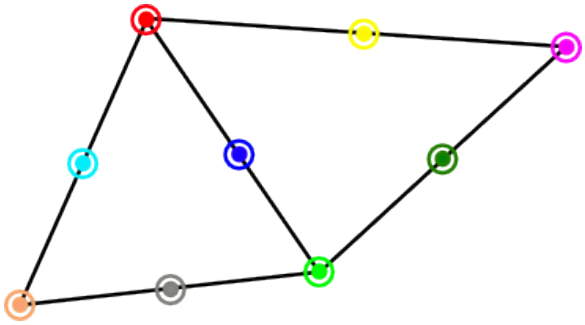
$P_1 \times P_1$ equation-wise



$P_1 \times P_1$ point block



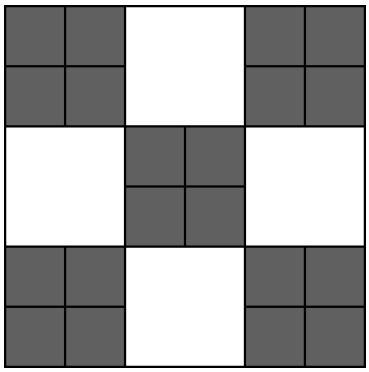
$P_2 \times P_2$ equation-wise



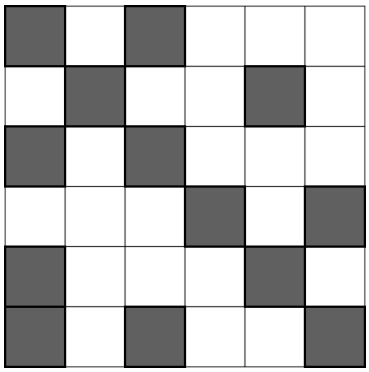
$P_2 \times P_2$ point block

3.2.2 Matrix Vector Components

Block Structure in ISTL Matrices



- Point block matrix is a sparse matrix with small dense blocks.
- Realized by BCRSMatrix< FieldMatrix<E,2,2> >.



- Sparse matrix of scalars.
- Realized by BCRSMatrix< FieldMatrix<E,1,1> >.

3.2.3 The Backend

ISTL Vector Backend

- **template**<**int** blocksize> **class** ISTLVectorBackend is the currently available backend for ISTL vectors.
- The backend uses BlockVector<FieldVector<E,blocksize> > as the underlying container.
- Attention: Make sure everything is correct for nonscalar backends blocksize> 1

ISTL Matrix Backend

- **template**<**int** brows,**int** bcols> **class** ISTLBCRSMatrixBackend is the currently available backend for ISTL sparse matrices.
- brows is the number rows and bcols is the number of columns of each matrix block.
- The backend uses BCRSMatrix<FieldMatrix<E,brows,bcols> > as the underlying container.
- Attention: Make sure everything is correct for nonscalar backends blocksize> 1.

ISTL Sequential Solver Backends

- ISTLBackend_SEQ_CG_SSOR: conjugate gradient method with SSOR preconditioner
- ISTLBackend_SEQ_BCGS_SSOR: stabilized bi-conjugate gradient method with SSOR preconditioner
- ISTLBackend_SEQ_SuperLU: SuperLU, a sparse direct solver
- **template**<**class** T> **class** ISTLBackend_SEQ_CG_AMG_SSOR and **template**<**class** T> **class** ISTLBackend_S the above methods preconditioned point-block AMG based on aggregation smoothed by SSOR.

Currently these sequential solvers are just a subset of the ones available in ISTL directly. More will follow in due time.

3.3 Parallel PDELab

- Go parallel by choosing
  1. a suitable parallel grid (e.g. an overlapping one for overlapping domain decomposition methods),
  2. the correct constraints for the discretization of the PDE, and
  3. a suitable and matching parallel solver backend of the PDELab backend.

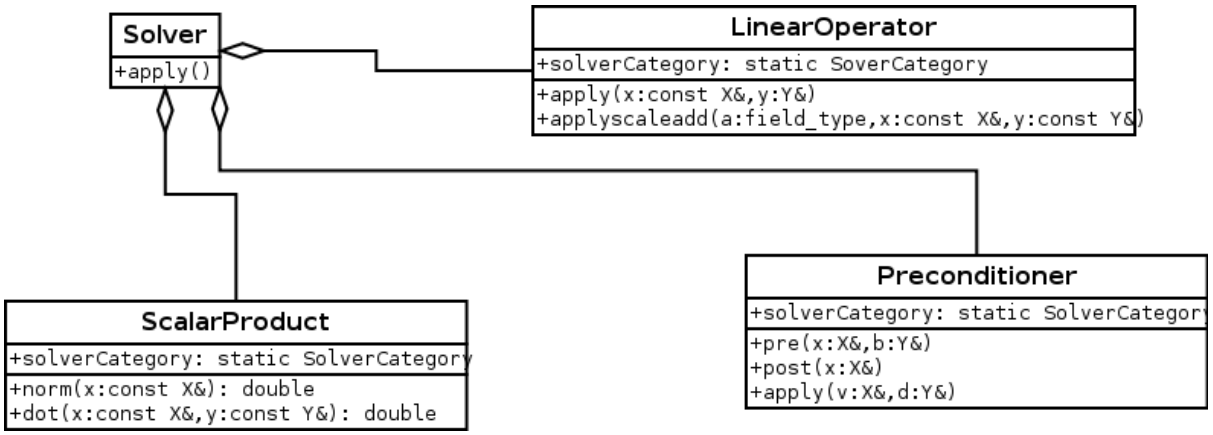
There are three different kinds of parallel solvers classes available in the ISTL backend:

- Nonoverlapping domain decomposition methods.

- Overlapping domain decomposition metods.
- Data parallel solvers (e.g. AMG).

Building Blocks for Parallel Solvers

- The solvers in ISTL do not use matrix and vector structures directly,
- but implementations of Preconditioner, LinearOperator and ScalarProduct.
- These components must match in the solver category (e.g. sequential, overlapping, nonoverlapping) and thus support the same data decomposition.
- Simply plugin parallel instances of the components to get parallel solvers.



Some Parallel Solver Backends

The solver backends can be found in header dune/pdelab/backend/istlsolverbackend.hh. Template parameter GFS is the type of the grid function space, template parameter C the type of the parallel constraints used.

- **template**<**class** GFS> **class** ISTLBackend_NOVLP_BCGS_NOPREC: parallel unpreconditioned stabilized bi-conjugate gradient method for nonoverlapping grids.
- **template**<**class** GFS> **class** ISTLBackend_OVLP_BCGS_SSORk: the above preconditioned with k steps of SSOR.
- **template**<**class** GFS, **class** C> **class** ISTLBackend_OVLP_BCGS_SuperLU: the first method preconditioned by an overlapping domain decomposition method with SuperLU for the problems local to the processors.
- **template**<**class** GFS> ISTLBackend_BCGS_AMG_SSOR: the first method preconsitioned by parallel AMG smoothed by SSOR. Requires an overlapping grid!

Nonoverlapping example

```
// 1. Create an non-overlapping grid
Dune::FieldVector<double,2> L(1.0);
Dune::FieldVector<int,2> N(16);
Dune::FieldVector<bool,2> periodic(false);
int overlap=0; // needs overlap 0 because overlap elements are not assembled
Dune::YaspGrid<2> grid(helper.getCommunicator(),L,N,periodic,overlap);
typedef Dune::YaspGrid<2>::LeafGridView GV;
const GV& gv=grid.leafView();

// 2. Create correctly constrained grid function space
typedef Dune::PDELab::Q1LocalFiniteElementMap<Coord,Real,dim> FEM;
FEM fem;
typedef Dune::PDELab::NonoverlappingConformingDirichletConstraints CON;
CON con;
typedef Dune::PDELab::ISTLVectorBackend<1> VBE;
typedef Dune::PDELab::GridFunctionSpace<GV,FEM,CON,VBE,
Dune::PDELab::SimpleGridFunctionStaticSize> GFS;
GFS gfs(gv,fem,con);
con.compute_ghosts(gfs); // con stores indices of ghost dofs
typedef ConvectionDiffusionProblem<GV,Real> Param;
Param param;
typedef Dune::PDELab::BoundaryConditionType_CD<Param> B;
B b(gv,param);
typedef Dune::PDELab::DirichletBoundaryCondition_CD<Param> G;
G g(gv,param);


// Compute constrained space
typedef typename GFS::template ConstraintsContainer<Real>::Type C;
C cg;
Dune::PDELab::constraints(b,gfs,cg);

// Compute affine shift
typedef typename GFS::template VectorContainer<Real>::Type V;
V x(gfs,0.0);
Dune::PDELab::interpolate(g,gfs,x);
Dune::PDELab::set_nonconstrained_dofs(cg,0.0,x);

// Make grid operator space
typedef Dune::PDELab::ConvectionDiffusion<Param> LOP;
LOP lop(param,2);
typedef Dune::PDELab::ISTLBCRSMatrixBackend<1,1> MBE;
typedef Dune::PDELab::GridOperatorSpace<GFS,GFS,LOP,C,C,MBE,true> GOS;
GOS gos(gfs,cg,gfs,cg,lop);

// 3. Choose a linear solver
typedef Dune::PDELab::ISTLBackend_NOVLP_BCGS_NOPREC<GFS> LS;
LS ls(gfs,5000,1);
...
```

Overlapping Example

```
// 1. Create an overlapping grid
Dune::FieldVector<double,2> L(1.0);
Dune::FieldVector<int,2> N(16);
Dune::FieldVector<bool,2> periodic(false);
int overlap=2;
Dune::YaspGrid<2> grid(helper.getCommunicator(),L,N,periodic,overlap);
typedef Dune::YaspGrid<2>::LeafGridView GV;
const GV& gv=grid.leafView();

// 2. Create correctly constrained grid function space
typedef Dune::PDELab::Q1LocalFiniteElementMap<Coord,Real,dim> FEM;
FEM fem;
typedef Dune::PDELab::OverlappingConformingDirichletConstraints CON;
typedef Dune::PDELab::ISTLVectorBackend<1> VBE;
typedef Dune::PDELab::GridFunctionSpace<GV,FEM,CON,VBE,
Dune::PDELab::SimpleGridFunctionStaticSize> GFS;
GFS gfs(gv,fem);

//  define problem parameters
typedef ConvectionDiffusionProblem<GV,Real> Param;
Param param;
typedef Dune::PDELab::BoundaryConditionType_CD<Param> B;
B b(gv,param);
typedef Dune::PDELab::DirichletBoundaryCondition_CD<Param> G;
G g(gv,param);


//  Compute constrained space
typedef typename GFS::template ConstraintsContainer<Real>::Type C;
```

```
C cg;
Dune::PDELab::constraints(b,gfs,cg);
//  Compute affine shift
typedef typename GFS::template VectorContainer<Real>::Type V;
V x(gfs,0.0);
Dune::PDELab::interpolate(g,gfs,x);
Dune::PDELab::set_nonconstrained_dofs(cg,0.0,x);
// Make grid operator space
typedef Dune::PDELab::ConvectionDiffusion<Param> LOP;
LOP lop(param,2);
typedef Dune::PDELab::ISTLBCRSMatrixBackend<1,1> MBE;
typedef Dune::PDELab::GridOperatorSpace<GFS,GFS,LOP,C,C,MBE> GOS;
GOS gos(gfs,cg,gfs,cg,lop);

// 3. Choose a linear solver
typedef Dune::PDELab::ISTLBackend_OVLP_BCGS_SuperLU<GFS,C> LS;
LS ls(gfs,cg,5000,2);
...
```

4 The DUNE Workflow for Simulations with CAD-Models

4.1 Overview

Real world problems can be defined on complex domains.

- Often, such domains are given by CAD-models.
- DUNE cannot handle such models directly.
- → Need of a workflow to import computational grids for CAD-models used in simulations.

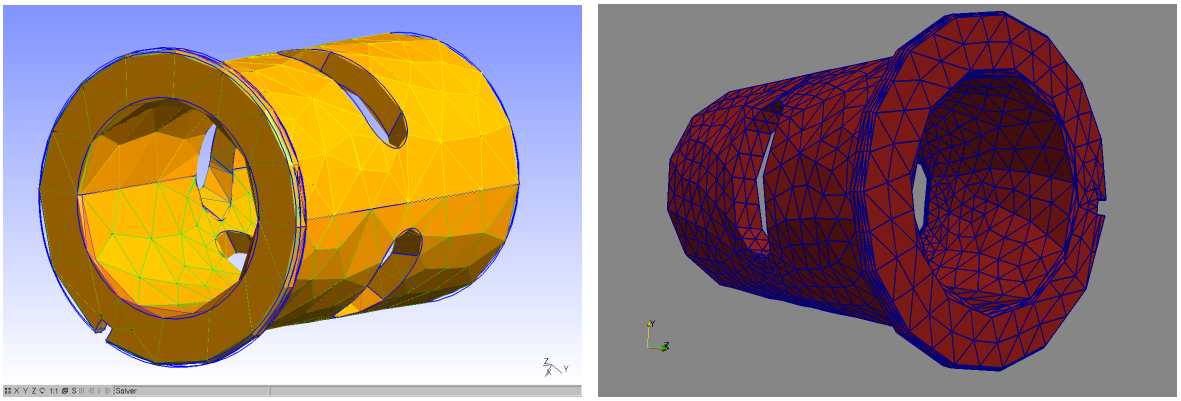
The interface to meshed CAD-Models in DUNE is via Gmsh-files.

The outline to import meshed CAD-models in DUNE is

1. Create or retrieve a CAD-model.
2. Edit and mesh the CAD-model with Gmsh (www.geuz.org/gmsh).
3. Export a .msh-File from Gmsh.
4. File can contain additional data such as connected subdomains.
5. Import the mesh file with the DUNE::GmshReader.
6. Evaluate physical data specified in the mesh file.

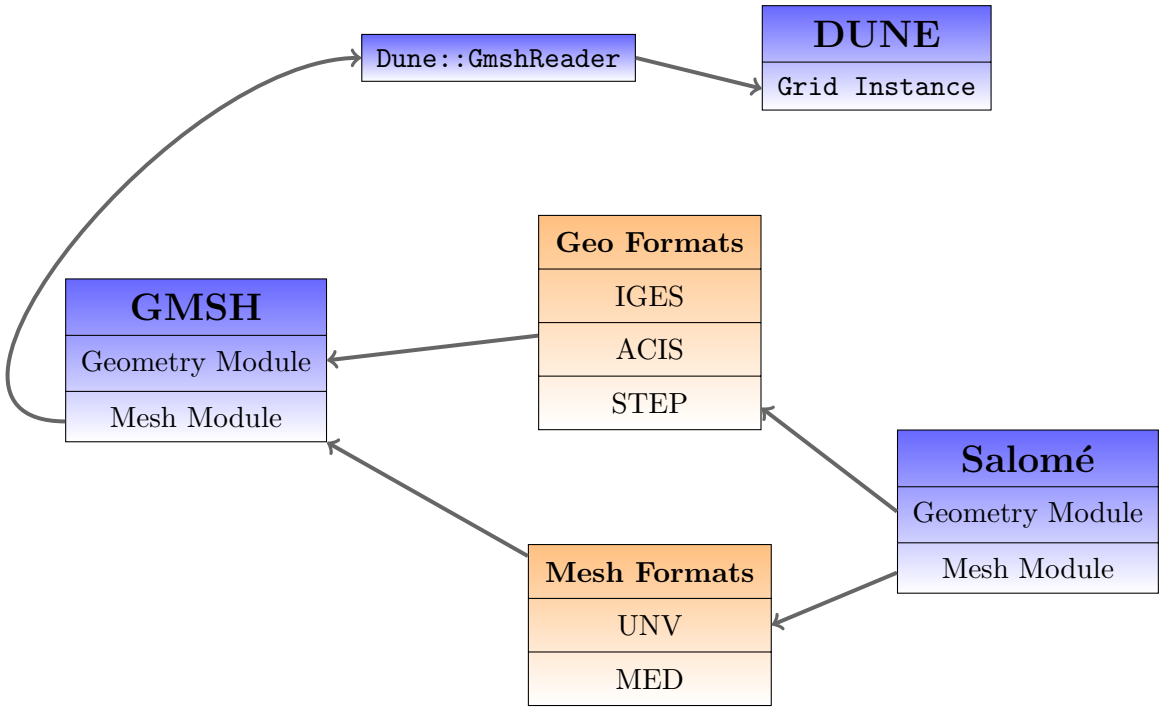
The important steps will be shown in detail in the following sections.

The picture shows a sample geometry meshed with Gmsh on the left and the mesh imported by DUNE on the right.



(CAD-Model from gCAD3D).

The flow charts depicts the general way of using gmsh or another external geometry modeler behind it to import CAD-models meshed by Gmsh into DUNE.



4.2 Gmsh and the DUNE Gmsh-Interface

The Dune::GmshReader

We begin with the import of a Mesh file meshfile.msh using DUNE's GmshReader:

```
1 // gridtest.cc: Read (sequential) gmsh files.
2
3 #ifdef HAVE_CONFIG_H
4 #include "config.h"
5 #endif
6 #include<iostream>
7 #include<vector>
8 #include<string>
9
10 #include<dune/common/exceptions.hh>
11 #include<dune/grid/io/file/vtk/vtkwriter.hh>
12 #include<dune/grid/io/file/gmshreader.hh>       // New: Dune::GmshReader
13 #if HAVE_UG                                    // New: Use UG here
14 #include<dune/grid/uggrid.hh>
15 #endif
16
17 //===============================================================
18 // Main program with grid setup
19 //===============================================================
20 int main(int argc, char** argv)
21 {
22   // check arguments
23   if (argc!=3)
24   {
25     std::cout << "usage: " << argv[0] << " <gridName> <level>" << std::endl;
26     return 1;
27   }
28
29   // refinement level
30   int level = 0;
31   sscanf(argv[2], "%d", &level);
32
33   // instanciate ug grid object (xxx MB heap)
34   typedef Dune::UGGrid<3> GridType;
35   GridType grid(400);
36
```

```
37   // read a gmsh file
38   std::string gridName = argv[1];
39   Dune::GmshReader<GridType> gmshreader;
40   gmshreader.read(grid, gridName);
41
42   // edit gridName
43   gridName.erase(0, gridName.rfind("/")+1);
44   gridName.erase(gridName.find(".",0), gridName.length());
45
46   // refine grid
47   grid.globalRefine(level);
48
49   // get a grid view
50   typedef GridType::LeafGridView GV;
51   const GV& gv = grid.leafView();
52
53   // plot celldata
54   std::vector<int> a(gv.size(0), 1);
55
56   //  output
57   Dune::VTKWriter<GV> vtkwriter(gv);
58   vtkwriter.addCellData(a, "celldata");
59   vtkwriter.write(gridName.c_str(), Dune::VTKOptions::ascii);
60
61   // done
62   return 0;
63 }
```

Changes to the code seen up to now are:

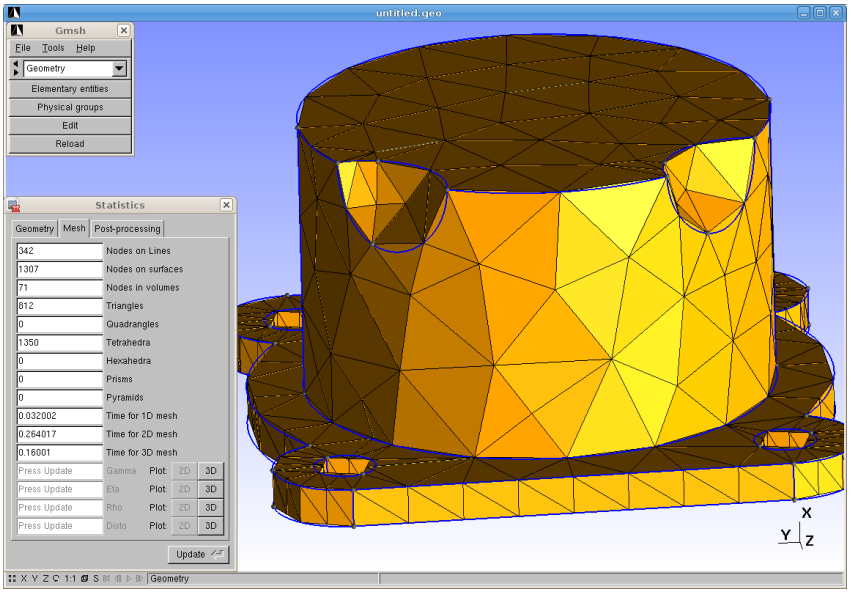
- Include the Gmsh-Interface header from dune−grid (line 12).
- Use UGGrid – Up to now UGGrid and ALUGrid are feasible for Gmsh meshes (line 33-35).
- Import gmsh file (line 37-40).

Gmsh

The mesh-files are created and exported by the exported software *Gmsh*, which is able to

- Import CAD models in various formats (GEO, STEP, IGES, BREP, ACIS, ...),
- Group subdomains of the models into phyiscal *entities* or groups,
- Mesh the models,
- Write the meshes including physical group regions to a mesh file (file-ending .msh).

It comes with a more or less user-friedly GUI:



(CAD-Model from gCAD3D).

Further features of Gmsh:

- Geometry Kernel: OpenCascade
  - Gmsh $\leq$ 2.3.1: Build from scratch for OpenCascde and Gmsh necessary to import CAD-models.
  - Gmsh $\geq$ 2.4.2: Precompiled build for Linux and MAC work with OpenCascade packages from distributor (Tested for Debian and MacOS).
- Does not cover full functionality of geometry kernel.

Further notes on ineraction between DUNE and Gmsh

DUNE and Gmsh Notes

- Possible: Second order boundary representation, readable by the GmshReader (Instable in the current trunk!).
- Underlying DUNE-Grid needs to support higher order boundaries. During grid refinement boundary is approximated the better.
- In DUNE: Higher order local FE bases available.
- Higher order elements are generally possible but not implemented yet in DUNE:
  - Need approriate grid-managers.
  - Need mesher which exports such elements.
  - GmshReader uses *GridFactories* to create grids.
    - GridFactory implementation depends on DUNE GridType.

* For UG and ALU the Factories are ready.
* For other grids, user may need to implement a specific factory.

4.3 Importing and Meshing CAD-Geometries with Gmsh

Common CAD-File Formats Readable by Gmsh

Gmsh allows to import CAD-models in common formats:

Format	Ending	Description
STEP	.stp, .step	ISO10303 norm, STandard for the Exchange of Product model data.
IGES	.igs, .iges	Transfers mainly geometry data 2D/3D shape models (NURBS, Bezier, ...)
ACIS	.sat	Volume modelling kernel. Widely used format.
BREP	.brp	List of convex polygons consisting of point lists In OpenCascade very limited!
GEO	.geo	The native Gmsh geometry format.

Meshing CAD-Models with Gmsh

Gmsh comes with a meshing functionality:

- Advancing front, and
- Delaunay meshes.
- Meshers can be parametrized by hypothesises.
- All meshes are handled as unstructured grids.
- Only simplicial meshes.

4.4 Attaching Data to a CAD-Geometry and its Mesh

Parameters can be associated to subdomains of the domain, e.g.

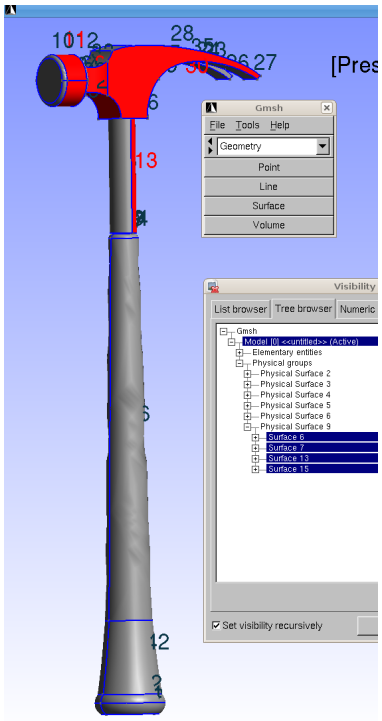
- Material properties (conductivities,...),
- Boundary conditions,
- ...

The subdomains available in Gmsh are:

- Volume groups,
- Surface groups,
- Line groups,
- Point groups.

How to set groups:

- Via `Geometry` → `Physical groups` in the menu window.



The Dune::GmshReader is able to read

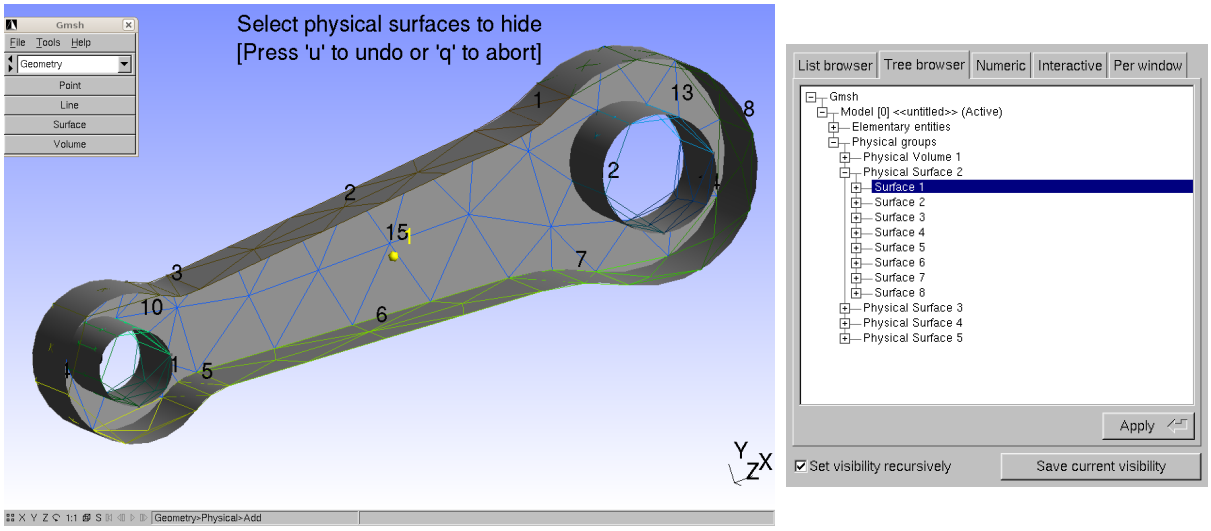
- Volume groups, and
- Surface groups

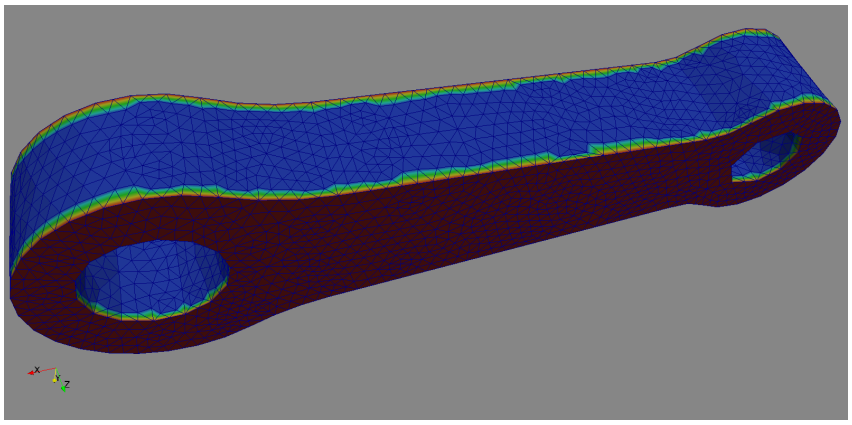
from a .msh-file. Mappings

- Codim 0 entities ⇔ Geometry volume groups, and
- Codim 1 entities ⇔ Geometry surface groups,

can be saved in two std::vector<**int**>s.

The next picture shows physical surface groups set by user interaction:





This above picture shows the mesh read by the GmshReader. Dirichlet constraints are set by the set_constrained_dofs method in Dune (red colored faces). Remaining faces have Neumann-0 boundary conditions.

4.5 A sample DUNE-Simulation importing a CAD-model

Listing 16.

```
1 /** \file
2
3     \brief Solve elliptic problem in constrained spaces with
4     conforming finite elements (as the stationary problems)
5 */
6 #ifdef HAVE_CONFIG_H
7 #include "config.h"
8 #endif
9
10 // dune includes
11 #include<math.h>
12 #include<iostream>
13 #include<vector>
```

```
14 //#include<map>
15#include<string>
16
17// dune includes
18#include<dune/common/mpihelper.hh>
19#include<dune/common/exceptions.hh>
20#include<dune/common/fvector.hh>
21#include<dune/common/static_assert.hh>
22#include<dune/common/timer.hh>
23#include<dune/grid/io/file/vtk/subsamplingvtkwriter.hh>
24#include<dune/grid/io/file/gmshreader.hh>
25#if HAVE_UG
26#include<dune/grid/uggrid.hh>
27#endif
28#include<dune/istl/bvector.hh>
29#include<dune/istl/operators.hh>
30#include<dune/istl/solvers.hh>
31#include<dune/istl/preconditioners.hh>
32#include<dune/istl/io.hh>
33#include<dune/istl/superlu.hh>
34
35// pdelab includes
36#include<dune/pdelab/common/function.hh>
37#include<dune/pdelab/common/vtkexport.hh>
38#include<dune/pdelab/finiteelementmap/p0fem.hh>
39#include<dune/pdelab/finiteelementmap/p1fem.hh>
40#include<dune/pdelab/finiteelementmap/conformingconstraints.hh>
41#include<dune/pdelab/gridfunctionspace/gridfunctionspace.hh>
42#include<dune/pdelab/gridfunctionspace/gridfunctionspaceutilities.hh>
43#include<dune/pdelab/gridfunctionspace/genericdatahandle.hh>
44#include<dune/pdelab/gridfunctionspace/interpolate.hh>
45#include<dune/pdelab/gridfunctionspace/constraints.hh>
46#include<dune/pdelab/gridoperatorspace/gridoperatorspace.hh>
47#include<dune/pdelab/gridoperatorspace/gridoperatorspaceutilities.hh>
48#include<dune/pdelab/backend/istlvectorbackend.hh>
49#include<dune/pdelab/backend/istlmatrixbackend.hh>
50#include<dune/pdelab/backend/istlsolverbackend.hh>
51#include<dune/pdelab/stationary/linearproblem.hh>
52
53// include application heaeders
54#include"cadsample_operator.hh"
55#include"cadsample_parameter.hh"
56#include"cadsample_P1.hh"
57
58//===============================================================
59// Problem
60//===============================================================
61// crank.igs with solid from Salome
62void crank(int level)
63{
64  // instanciate ug grid object
65  typedef Dune::UGGrid<3> GridType;
66  GridType grid(400);
67
68  // vectors for boundary and material conditions
69  std::vector<int> boundaryIndexToPhysicalEntity;
70  std::vector<int> elementIndexToPhysicalEntity;
71
72  // read a gmsh file
73  std::string gridName = "./grids/crank.msh";
74  Dune::GmshReader<GridType> gmshreader;
75  gmshreader.read(grid, gridName, boundaryIndexToPhysicalEntity,
76      elementIndexToPhysicalEntity, true, false);
77
```

```
78   // refine grid
79   grid.globalRefine(level);
80
81   // get a grid view
82   typedef GridType::LeafGridView GV;
83   const GV& gv = grid.leafView();
84
85   // material conditions
86   typedef CrankDiffusion<GV,double,std::vector<int> > M;
87   M m(gv, elementIndexToPhysicalEntity);
88
89   // boundary conditions
90   typedef CrankBCType<GV,std::vector<int> > B;
91   B b(gv, boundaryIndexToPhysicalEntity);
92   typedef CrankBCExtension<GV,double,std::vector<int> > G;
93   G g(gv, boundaryIndexToPhysicalEntity);
94
95   // boundaries fluxes
96   typedef CrankFlux<GV,double,std::vector<int> > J;
97   J j(gv, boundaryIndexToPhysicalEntity);
98
99   // call driver with parameters
100  gridName.erase(0, gridName.rfind("/")+1);
101  gridName.erase(gridName.find(".",0), gridName.length());
102  cadsample_P1(gv, m, b, g, j, gridName);
103 }
104
105 //===========================================================
106 // Main program with grid setup
107 //===========================================================
108 int main(int argc, char** argv)
109 {
110   // scan arguments
111   if (argc!=2)
112   {
113     std::cout << "usage: ./cadsample <level>" << std::endl;
114     return 1;
115   }
116
117   // refinement level
118   int level = 0;
119   sscanf(argv[1],"%d",&level);
120
121   // run simulation
122   crank(level);
123
124   return 0;
125 }
```

Notes:

- The GmshReader takes two vectors to store volume and surface map data from the Gmsh file.
- In function crank, the parameter classes for the problem are created. In the constructor they take the according vectors.
  Example: CrankBCType evaluates the mapping Dune::Intersection with boundary to physical group of geometrical boundary.
- All parameter objects are passed to the driving function cadsample_P1 which forwards them to the operator.

Listing 17.

```
#ifndef _CADSAMPLE_PARAMETER_HH_
#define _CADSAMPLE_PARAMETER_HH_

// layout for codim0 data
template <int dim>
struct P0Layout
{
  bool contains(Dune::GeometryType gt)
  {
    if ( gt.dim() == dim ) return true;
    return false;
  }
};

//==========================================================
// crank parameter classes
//==========================================================

// function defining scalar diffusion parameter
template<typename GV, typename RF, typename PGMap>
class CrankDiffusion
  : public Dune::PDELab::GridFunctionBase<
        Dune::PDELab::GridFunctionTraits<GV,RF,1,Dune::FieldVector<RF,1> >,
        CrankDiffusion<GV,RF,PGMap> >
{
public:

  typedef Dune::PDELab::GridFunctionTraits<GV,RF,1,Dune::FieldVector<RF,1> > Traits;
  typedef Dune::PDELab::GridFunctionBase<Dune::PDELab::GridFunctionTraits<GV,RF,1,
      Dune::FieldVector<RF,1> >, CrankDiffusion<GV,RF,PGMap> > BaseT;

  // constructor
  CrankDiffusion(const GV& gv_, const PGMap& pg_) : gv(gv_), mapper(gv), pg(pg_) {}

  // evaluate scalar diffusion parameter
  inline void evaluate (const typename Traits::ElementType& e,
                        const typename Traits::DomainType& x,
                        typename Traits::RangeType& y) const
  {
    // retrieve element index and corresponding material index on level 0
    typename GV::template Codim<0>::EntityPointer ep(e);
    while (ep->level() != 0) ep = ep->father();
    const int ei             = mapper.map(e);
    const int physgroup_index = pg[ei];

    // evaluate physical group map and set values accordingly
    switch ( physgroup_index )
    {
      case 1  : y = 1.0;  break;
      default : y = 1.0;  break; // only one material here
    }
  }

  inline const typename Traits::GridViewType& getGridView() { return gv; }

private:

  const GV& gv;
  const Dune::MultipleCodimMultipleGeomTypeMapper<GV,P0Layout> mapper;
  const PGMap& pg;
};

/** \brief boundary grid function selecting boundary conditions
```

```
64  * 0 means Neumann
65  * 1 means Dirichlet
66  */
67 template<typename GV, typename PGMap>
68 class CrankBCType : public Dune::PDELab::BoundaryGridFunctionBase<
69         Dune::PDELab::BoundaryGridFunctionTraits<GV,int,1,
70         Dune::FieldVector<int,1> >,CrankBCType<GV,PGMap> >
71 {
72 public:
73
74   typedef Dune::PDELab::BoundaryGridFunctionTraits<
75           GV,int,1,Dune::FieldVector<int,1> > Traits;
76
77   //! construct from grid view
78   CrankBCType (const GV& gv_, const PGMap& pg_) : gv(gv_), pg(pg_) {}
79
80   //! return bc type at point on intersection
81   template<typename I>
82   inline void evaluate (I& i, const typename Traits::DomainType& xlocal,
83                         typename Traits::RangeType& y) const
84   {
85     // use with global ccordinates
86     const int dim = Traits::GridViewType::Grid::dimension;
87     typedef typename Traits::GridViewType::Grid::ctype ctype;
88     Dune::FieldVector<ctype,dim> x = i.geometry().global(xlocal);
89     if ( (x[2]<20.+1E-6) || (x[2] > 59.979-1e-6 ) )
90       y = 1; // Dirichlet
91     else
92       y = 0; // Neumann
93     return;
94
95     // evaluate with maps
96     //int physgroup_index = pg[i.boundarySegmentIndex()];
97     //switch ( physgroup_index )
98     //{
99     //  case 2  : y = 0; break;
100    //  case 3  : y = 0; break;
101    //  case 4  : y = 1; break;
102    //  case 5  : y = 1; break;
103    //  default : y = 0; break; // Neumann
104    //}
105    //return;
106  }
107
108  //! get a reference to the grid view
109  inline const GV& getGridView () {return gv;}
110
111 private:
112
113   const GV&     gv;
114   const PGMap& pg;
115 };
116
117 /**
118  * \brief A function that defines Dirichlet boundary conditions AND its extension to the interior
119  */
120 template<typename GV, typename RF, typename PGMap>
121 class CrankBCExtension
122   : public Dune::PDELab::GridFunctionBase<Dune::PDELab::
123             GridFunctionTraits<GV,RF,1,Dune::FieldVector<RF,1> >,
124             CrankBCExtension<GV,RF,PGMap> >
125 {
126 public :
127
```

```
128   typedef Dune::PDELab::GridFunctionTraits<GV,RF,1,Dune::FieldVector<RF,1> > Traits;
129   typedef typename GV::IntersectionIterator IntersectionIterator;
130
131   //! construct from grid view
132   CrankBCExtension(const GV& gv_, const PGMap& pg_) : gv(gv_), mapper(gv), pg(pg_) {}
133
134   //! evaluate extended function on element
135   inline void evaluate (const typename Traits::ElementType& e,
136                         const typename Traits::DomainType& xlocal,
137                         typename Traits::RangeType& y) const
138   {
139     // evaluate with global ccordinates
140     const int dim = Traits::GridViewType::Grid::dimension;
141     typedef typename Traits::GridViewType::Grid::ctype ctype;
142     Dune::FieldVector<ctype,dim> x = e.geometry().global(xlocal);
143     if ( x[2] < 20.0+1E-6 )
144       y = 1.0;
145     else
146       y = 0.0;
147     return;
148   }
149
150   //! get a reference to the grid view
151   inline const GV& getGridView() { return gv; }
152
153 private :
154
155   const GV& gv;
156   const Dune::MultipleCodimMultipleGeomTypeMapper<GV,P0Layout> mapper;
157   const PGMap& pg;
158 };
159
160 // function for defining radiation and Neumann boundary conditions
161 template<typename GV, typename RF, typename PGMap>
162 class CrankFlux
163   : public Dune::PDELab::BoundaryGridFunctionBase<
164              Dune::PDELab::BoundaryGridFunctionTraits<GV,RF,1,
165              Dune::FieldVector<RF,1> >, CrankFlux<GV,RF,PGMap> >
166 {
167 public:
168
169   typedef Dune::PDELab::BoundaryGridFunctionTraits<
170            GV,RF,1,Dune::FieldVector<RF,1> > Traits;
171
172   // constructor
173   CrankFlux(const GV& gv_, const PGMap& pg_) : gv(gv_), pg(pg_) {}
174
175   // evaluate flux boundary condition
176   template<typename I>
177   inline void evaluate(I& i, const typename Traits::DomainType& xlocal,
178                        typename Traits::RangeType& y) const
179   {
180     // could be handled as in the case of the BCType class!
181     y = 0.0;
182     return;
183   }
184
185 private:
186
187   const GV&     gv;
188   const PGMap& pg;
189 };
190
191 #endif
```

Notes:

- The parameter classes evaluate the physical maps instead of global coordinates now.
- E.g., take the diffusion coefficient. Based on the element index it evaluates the physical group map for volumes.
- Similar evaluates hold true for the other parameter classes.

Listing 18.

```
1 // driver now parametrized by M, B, G, J                                    // NEW
2 template<typename GV, typename M, typename B, typename G, typename J>
3 void cadsample_P1 (const GV& gv, const M& m, const B& b, const G& g,
4                    const J& j, std::string gridName)
5 {
6   // <<<1>>> Choose domain and range field type
7   typedef typename GV::Grid::ctype Coord;
8   typedef double Real;
9   const int dim = GV::dimension;
10
11  // <<<2a>>> Make grid function space
12  typedef Dune::PDELab::P1LocalFiniteElementMap<Coord,Real,dim> FEM;
13  FEM fem;
14  typedef Dune::PDELab::ConformingDirichletConstraints CON;
15  CON con;
16  typedef Dune::PDELab::ISTLVectorBackend<1> VBE;
17  typedef Dune::PDELab::GridFunctionSpace<GV,FEM,CON,VBE> GFS;
18  GFS gfs(gv,fem);
19
20  // <<<2b>>> Compute constraints on function space
21  typedef typename GFS::template ConstraintsContainer<Real>::Type CC;
22  CC cc;
23  Dune::PDELab::constraints(b,gfs,cc);
24  std::cout << "constrained␣dofs=" << cc.size()
25            << "␣of␣" << gfs.globalSize() << std::endl;
26
27  // <<<3>>> Make FE function extending Dirichlet boundary conditions
28  typedef typename GFS::template VectorContainer<Real>::Type V;
29  V x(gfs,0.0);
30  Dune::PDELab::interpolate(g,gfs,x);
31
32  // <<<4>>> Make grid operator space
33  typedef CADLocalOperator<M,B,J> LOP;                                       // NEW
34  LOP lop(m, b, j);
35  typedef Dune::PDELab::ISTLBCRSMatrixBackend<1,1> MBE;
36  typedef Dune::PDELab::GridOperatorSpace<GFS,GFS,LOP,CC,CC,MBE> GOS;
37  GOS gos(gfs,cc,gfs,cc,lop);
38
39  // <<<5>>> Select a linear solver backend
40#if HAVE_SUPERLU
41  typedef Dune::PDELab::ISTLBackend_SEQ_SuperLU LS;
42  LS ls(true);
43#else
44  typedef Dune::PDELab::ISTLBackend_SEQ_BCGS_SSOR LS;
45  LS ls(5000,true);
46#endif
47
48  // <<<6>>> assemble and solve linear problem
49  Dune::PDELab::StationaryLinearProblemSolver<GOS,LS,V> slp(gos,x,ls,1e-10);
50  slp.apply();
51
52  // <<<7>>> graphical output
53  {
54    typedef Dune::PDELab::DiscreteGridFunction<GFS,V> DGF;
```

```
55      DGF xdgf(gfs,x);
56      Dune::VTKWriter<GV> vtkwriter(gv,Dune::VTKOptions::conforming);
57      vtkwriter.addVertexData(new Dune::PDELab::VTKGridFunctionAdapter<DGF>(xdgf,"solution"));
58      vtkwriter.write(gridName.c_str(), Dune::VTKOptions::binaryappended);
59    }
60 }
```

Notes:

- The driver is nearly unchanged to the stationary examples.
- Passes the parameter objects to the operator and constraints.

Listing 19.

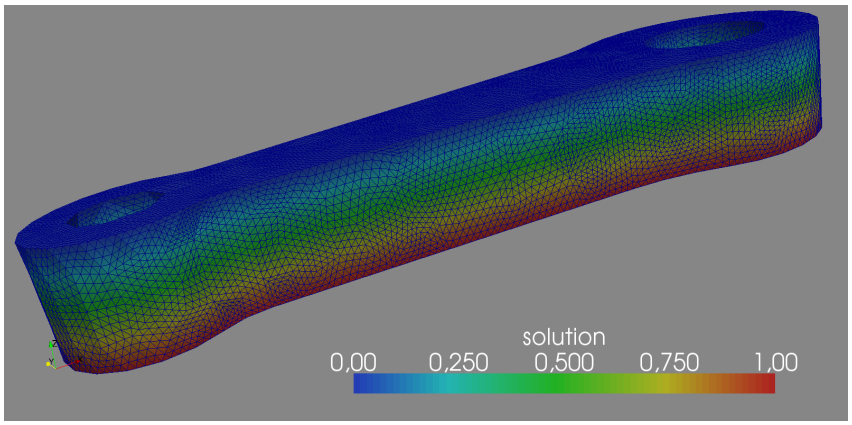
```
1#include<dune/grid/common/genericreferenceelements.hh>
2#include<dune/grid/common/quadraturerules.hh>
3#include<dune/pdelab/common/geometrywrapper.hh>
4#include<dune/pdelab/localoperator/pattern.hh>
5#include<dune/pdelab/localoperator/flags.hh>
6
7/** a local operator for solving the equation
8 *
9 *   - \Delta m(x) u + a*u    = f    in \Omega
10 *                      u    = g    on \Gamma_D\subseteq\partial\Omega
11 *   - \nabla m(x) u \cdot n = j    on \Gamma_N = \partial\Omega\setminus\Gamma_D
12 *
13 * with conforming finite elements on all types of grids in any dimension
14 * Note the source term is 0 here.
15 *
16 * \tparam M a function indicating the diffusion coefficient
17 * \tparam B a function indicating the type of boundary condition
18 * \tparam J a function indicating the source term
19 */
20 template<typename M, typename B, typename J>                    // NEW
21 class CADLocalOperator :
22
23   public Dune::PDELab::NumericalJacobianApplyVolume<CADLocalOperator<M, B, J> >,
24   public Dune::PDELab::NumericalJacobianVolume<CADLocalOperator<M, B, J> >,
25   public Dune::PDELab::NumericalJacobianApplyBoundary<CADLocalOperator<M, B, J> >,
26   public Dune::PDELab::NumericalJacobianBoundary<CADLocalOperator<M, B, J> >,
27   public Dune::PDELab::FullVolumePattern,
28   public Dune::PDELab::LocalOperatorDefaultFlags
29 {
30
31 public:
32
33   // pattern assembly flags
34   enum { doPatternVolume = true };
35
36   // residual assembly flags
37   enum { doAlphaVolume = true };
38   enum { doAlphaBoundary = true };
39
40   // constructor parametrized by material and boundary classes          // NEW
41   CADLocalOperator (const M& m_, const B& b_, const J& j_, unsigned int intorder_=2)
42     : m(m_), b(b_), j(j_), intorder(intorder_)
43   {}
44
45   // volume integral depending on test and ansatz functions
46   template<typename EG, typename LFSU, typename X, typename LFSV, typename R>
47   void alpha_volume (const EG& eg, const LFSU& lfsu, const X& x, const LFSV& lfsv, R& r) const
48   {
49     // extract some types
```

```
50    typedef typename LFSU::Traits::LocalFiniteElementType::
51      Traits::LocalBasisType::Traits::DomainFieldType DF;
52    typedef typename LFSU::Traits::LocalFiniteElementType::
53      Traits::LocalBasisType::Traits::RangeFieldType RF;
54    typedef typename LFSU::Traits::LocalFiniteElementType::
55      Traits::LocalBasisType::Traits::JacobianType JacobianType;
56    typedef typename LFSU::Traits::LocalFiniteElementType::
57      Traits::LocalBasisType::Traits::RangeType RangeType;
58    typedef typename LFSU::Traits::SizeType size_type;
59
60    // dimensions
61    const int dim = EG::Geometry::dimension;
62    const int dimw = EG::Geometry::dimensionworld;
63
64    // select quadrature rule
65    Dune::GeometryType gt = eg.geometry().type();
66    const Dune::QuadratureRule<DF,dim>& rule = Dune::QuadratureRules<DF,dim>::rule(gt,intorder);
67
68    // loop over quadrature points
69    for (typename Dune::QuadratureRule<DF,dim>::const_iterator
70           it=rule.begin(); it!=rule.end(); ++it)
71    {
72      // evaluate basis functions on reference element
73      std::vector<RangeType> phi(lfsu.size());
74      lfsu.localFiniteElement().localBasis().evaluateFunction(it->position(),phi);
75
76      // compute u at integration point
77      RF u=0.0;
78      for (size_type i=0; i<lfsu.size(); i++)
79        u += x[i]*phi[i];
80
81      // evaluate gradient of basis functions on reference element
82      std::vector<JacobianType> js(lfsu.size());
83      lfsu.localFiniteElement().localBasis().evaluateJacobian(it->position(),js);
84
85      // transform gradients from reference element to real element
86      const Dune::FieldMatrix<DF,dimw,dim>
87        jac = eg.geometry().jacobianInverseTransposed(it->position());
88      std::vector<Dune::FieldVector<RF,dim> > gradphi(lfsu.size());
89      for (size_type i=0; i<lfsu.size(); i++)
90        jac.mv(js[i][0],gradphi[i]);
91
92      // compute gradient of u
93      Dune::FieldVector<RF,dim> gradu(0.0);
94      for (size_type i=0; i<lfsu.size(); i++)
95        gradu.axpy(x[i],gradphi[i]);
96
97      // evaluate parameters;
98      Dune::FieldVector<RF,dim>
99        globalpos = eg.geometry().global(it->position());
100     RF f = 0;
101     RF a = 0;
102     typename M::Traits::RangeType y;
103     m.evaluate(eg.entity(), it->position(), y);                    // NEW
104     gradu *= (double) y;
105
106     // integrate grad u * grad phi_i + a*u*phi_i - f phi_i
107     RF factor = it->weight()*eg.geometry().integrationElement(it->position());
108     for (size_type i=0; i<lfsu.size(); i++)
109       r[i] += ( gradu*gradphi[i] + a*u*phi[i] - f*phi[i] )*factor;
110   }
111 }
112
113 // boundary integral
```

```
114   template<typename IG, typename LFSU, typename X, typename LFSV, typename R>
115   void alpha_boundary (const IG& ig, const LFSU& lfsu_s, const X& x_s,
116                        const LFSV& lfsv_s, R& r_s) const
117   {
118     // some types
119     typedef typename LFSV::Traits::LocalFiniteElementType::
120       Traits::LocalBasisType::Traits::DomainFieldType DF;
121     typedef typename LFSV::Traits::LocalFiniteElementType::
122       Traits::LocalBasisType::Traits::RangeFieldType RF;
123     typedef typename LFSV::Traits::LocalFiniteElementType::
124       Traits::LocalBasisType::Traits::RangeType RangeType;
125     typedef typename LFSV::Traits::SizeType size_type;
126
127     // dimensions
128     const int dim = IG::dimension;
129
130     // select quadrature rule for face
131     Dune::GeometryType gtface = ig.geometryInInside().type();
132     const Dune::QuadratureRule<DF,dim-1>& rule = Dune::QuadratureRules<DF,dim-1>::rule(gtface,intorder)
133
134     // loop over quadrature points and integrate normal flux
135     for (typename Dune::QuadratureRule<DF,dim-1>::const_iterator it=rule.begin(); it!=rule.end(); ++it)
136       {
137         // evaluate boundary condition type
138         typename B::Traits::RangeType bctype;
139         b.evaluate(ig,it->position(),bctype);
140
141         // skip rest if we are on Dirichlet boundary
142         if (bctype>0) continue;
143
144         // position of quadrature point in local coordinates of element
145         Dune::FieldVector<DF,dim> local = ig.geometryInInside().global(it->position());
146
147         // evaluate basis functions at integration point
148         std::vector<RangeType> phi(lfsv_s.size());
149         lfsu_s.localFiniteElement().localBasis().evaluateFunction(local,phi);
150
151         // evaluate u (e.g. flux may depend on u)
152         RF u=0.0;
153         for (size_type i=0; i<lfsu_s.size(); i++)
154           u += x_s[i]*phi[i];
155
156         // evaluate flux boundary condition
157         typename J::Traits::RangeType y;                                    // NEW
158         j.evaluate(ig, it->position(), y);
159
160         // integrate j
161         RF factor = it->weight()*ig.geometry().integrationElement(it->position());
162         for (size_type i=0; i<lfsv_s.size(); i++)
163           r_s[i] += y * phi[i] * factor;
164       }
165   }
166
167 private:
168
169   const M& m;
170   const B& b;
171   const J& j;
172   unsigned int intorder;
173 };
```

Notes:

- The operator is now parametrized with the boundary type selection, boundary extension and boundary flux classes.
- Source term vanishes but could also be a paramter to the operator.
- Evaluation of the parameter in operator via enities and intersecions. Compare the parameter classes!



This picture shows the results of the simulation (stationary diffusion, no flux boundaries, P1-FEM).

4.6 Some other applicable Open Source CAD-Tools

Salome

Salome is a more user-friendly CAE-environment which can also export CAD-models and meshes suitable for Gmsh-Import.

- Geometry Kernel: OpenCascade, exposes more of the functionality than Gmsh.
- Can import and export IGES-, STEP-, BREP-, ACIS-files.
- Can import and export STL and UNV-meshes.
- Has a generic Mesher interface (Available: Netgen, GHS3D, ...)
- User friendly GUI for creating and processing CAD-models.

Other Open Source CAD-Tools

Some other CAD-Tools which are able to export geometry models or mesh files suitable for Gmsh import:

Tool	Licence	Geom Import	Geom Import	Mesh Export
Gmsh	LGPL	geo	iges, step, brep, acis	msh
Salome	LGPL	iges, step, brep, acis	iges, step, brep, acis	unv, stl
gCAD3D	Freeware	step, iges	step, iges	
Blender	own (BL)	CSG	iges	
PythonOCC	GPL	Python-Wrapper for OpenCascade		

5 Solving Instationary Problems

5.1 One Step Methods

Approach for Instationary Problems

- Method of lines approach:
  - Semi-discretization in space.
  - Solve large ODE problem.
- One step methods for ODEs
  - Diagonally implicit Runge-Kutta methods.
  - Explicit Runge-Kutta methods.
  - Includes Explicit/implicit Euler, Crank-Nicolson, fractional step θ, Alexanders S-stable methods [Ale77], Shu's explicit TVD Runge-Kutta methods [Shu88]...
- Multistep methods (e.g. BDF) could be implemented easily in current approach.
- Space-time methods are not yet supported
  - Require extension of grid function space.

Problem and Finite Element Formulation

Consider the following model problem:

$$\begin{aligned} \partial_t u - \Delta u + au &= f && \text{in } \Omega \times \Sigma, \Sigma = (t_0, t_0 + T), \\ u(\cdot, t) &= g(\cdot, t) && \text{on } \Gamma_D, \\ -\nabla u(\cdot, t) \cdot n &= j(\cdot, t) && \text{on } \Gamma_N = \partial\Omega \setminus \Gamma_D, \\ u(\cdot, t_0) &= u_0(\cdot) && \text{at } t = t_0. \end{aligned}$$

Semi-discretization in space. Find $u_h \in L_2(t_0, t_0 + T; w_h + \tilde{U}_h^k)$:

$$\frac{d}{dt} \underbrace{\int_\Omega u_h v \, dx}_{m(u_h(t), v; t)} + \underbrace{\int_\Omega \nabla u_h \cdot \nabla v + a u_h v - f v \, dx + \int_{\Gamma_N} j v \, ds}_{r(u_h, v; t)} = 0 \qquad \begin{array}{l} \forall v \in \tilde{U}_h^k, \\ t \in \Sigma. \end{array}$$

$r(u, v; t)$ is known residual form, except it may depend on time.

$m(u, v; t)$ is a new form but can be described by the same interface.

Implicit Euler Method

Subdivide time interval

$$\overline{\Sigma} = \{t^0\} \cup (t^0, t^1] \cup \ldots \cup (t^{N-1}, t^N]$$

with $t^0 = t_0$, $t^N = t_0 + T$, $t^{n-1} < t^n$ for $1 \leq n \leq N$.

Set $k^n = t^{n+1} - t^n$.

Difference quotient in time. Find $u_h \in w_h(t^{n+1}) + \tilde{U}_h^k(t^{n+1})$:

$$\frac{1}{k^n}\left\{m\left(u_h^{n+1}, v; t^{n+1}\right) - m\left(u_h^n, v; t^n\right)\right\} + r\left(u_h^{n+1}, v; t^{n+1}\right) = 0 \qquad \forall v \in \tilde{U}_h^k(t^{n+1}).$$

Solve (non-) linear problem per time step:

$$\underbrace{m\left(u_h^{n+1}, v; t^{n+1}\right) + k^n r\left(u_h^{n+1}, v; t^{n+1}\right) - m\left(u_h^n, v; t^n\right)}_{\tilde{r}(u_h^{n+1}, v)} = 0 \ \forall v \in \tilde{U}_h^k(t^{n+1}).$$

General One Step Methods I

The general scheme reads:

1. $u_h^{(0)} = u_h^n$.
2. For $i = 1, \ldots, s \in \mathbb{N}$, find $u_h^{(i)} \in w_h(t^n + d_i k^n) + \tilde{U}_h^k(t^{n+1})$:

$$\sum_{j=0}^{s}\Big[a_{ij} m_h\left(u_h^{(j)}, v; t^n + d_j k^n\right) + b_{ij} k^n r_h\left(u_h^{(j)}, v; t^n + d_j k^n\right)\Big] = 0 \qquad \forall v \in \tilde{U}_h^k(t^{n+1}).$$

3. $u_h^{n+1} = u_h^{(s)}$.

Assumption: Type of boundary condition does not change in $(t^n, t^{n+1}]$.

General One Step Methods II

- An s-stage scheme is given by the parameters

$$A = \begin{bmatrix} a_{10} & \ldots & a_{1s} \\ \vdots & & \vdots \\ a_{s0} & \ldots & a_{ss} \end{bmatrix}, \quad B = \begin{bmatrix} b_{10} & \ldots & b_{1s} \\ \vdots & & \vdots \\ b_{s0} & \ldots & b_{ss} \end{bmatrix}, \quad d = (d_0, \ldots, d_s)^T.$$

- *Explicit* schemes: $a_{ij} = 0$ for $j > i$ and $b_{ij} = 0$ for $j \geq i$.
- *Diagonally implicit* schemes: $a_{ij} = b_{ij} = 0$ for $j > i$.
- Fully implicit schemes are not considered.
- Normalization: $a_{ii} = 1$.
- You can easily add new schemes.

Some Examples

- One step θ scheme:
$$A = \begin{bmatrix} -1 & 1 \end{bmatrix}, \quad B = \begin{bmatrix} 1-\theta & \theta \end{bmatrix}, \quad d = (0,1)^T.$$
Explicit/implicit Euler ($\theta \in \{0,1\}$), Crank-Nicolson ($\theta = 1/2$).
- Heun's second order explicit method
$$A = \begin{bmatrix} -1 & 1 & 0 \\ -1/2 & -1/2 & 1 \end{bmatrix}, \quad B = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1/2 & 0 \end{bmatrix}, \quad d = (0,1,1)^T.$$
- Alexander's second order strongly S-stable method:
$$A = \begin{bmatrix} -1 & 1 & 0 \\ -1 & 0 & 1 \end{bmatrix}, \quad B = \begin{bmatrix} 0 & \alpha & 0 \\ 0 & 1-\alpha & \alpha \end{bmatrix}, \quad d = (0,\alpha,1)^T$$
with $\alpha = 1 - \sqrt{2}/2$.

Implicit vs. Explicit Methods

- Implicit methods require a "strong" solver in each stage.
- Explicit methods for $m(u,v;t)$ *linear* in u combined with suitable spatial discretization (e.g. FV) lead to
$$\mathbf{D}\mathbf{u}^{(i)} = \mathbf{s} + k^n \mathbf{q},$$
where
  - $\mathbf{D}$ is a (block-) diagonal or even identity matrix.
  - k^n is determined by stability limit *dynamically* while assembling $\mathbf{s}$, $\mathbf{q}$.
- Specialized algorithm for such explicit methods:
  1. Assemble $\mathbf{s}$ and $\mathbf{q}$ separately.
  2. Compute optimal k^n while assembling.
  3. Vector update $\mathbf{b} = \mathbf{s} + k^n \mathbf{q}$.
  4. "Solve" diagonal System $\mathbf{D}\mathbf{u}^{(i)} = \mathbf{b}$.

5.2 Example 3

Example 3 Overview

Example 3 solves the following problem

$$\begin{aligned} \partial_t u - \Delta u + au &= f && \text{in } \Omega \times \Sigma, \Sigma = (t_0, t_0+T], \\ u(\cdot,t) &= g(\cdot,t) && \text{on } \Gamma_D(t), \\ -\nabla u(\cdot,t)\cdot n &= j(\cdot,t) && \text{on } \Gamma_N(t) = \partial\Omega \setminus \Gamma_D(t), \\ u(\cdot,t_0) &= u_0(\cdot) && \text{at } t = t_0. \end{aligned}$$

and consists of the files

- example03.cc – main program.
- example03_Q2.hh – driver to solve problem on grid view.
- example03_bctype.hh – defines $\partial\Omega = \Gamma_D(t) \cup \Gamma_N(t)$.
- example03_bcextension.hh – defines boundary values.
- example03_operator.hh – $r(u, v; t)$.
- example03_toperator.hh – $m(u, v; t)$.

Driver for Solving Instationary Linear Problem About example03_Q2.hh

Major extensions from the stationary example03 are:

- Boundary condition type and boundary condition extension depend on time.
  - Keep interface, add method setTime().
- Two local operators are needed now
  - One for $r(u, v; t)$.
  - Another for $m(u, v; t)$.
  - Same idea: keep interface, add setTime() method.
- Class InstationaryGridOperatorSpace takes two local operators.
- Class OneStepMethod implements time-stepping scheme.
- Time loop is under user control.

Listing 20 (File examples/example03_Q2.hh).

```
1 template<class GV>
2 void example03_Q2 (const GV& gv, double dt, double tend)
3 {
4   // <<<1>>> Choose domain and range field type
5   typedef typename GV::Grid::ctype Coord;
6   typedef double Real;
7   const int dim = GV::dimension;
8   Real time = 0.0;                                          // make a time variable
9
10  // <<<2>>> Make grid function space
11  typedef Dune::PDELab::Q22DLocalFiniteElementMap<Coord,Real> FEM;
12  FEM fem;
13  typedef Dune::PDELab::ConformingDirichletConstraints CON;
14  CON con;
15  typedef Dune::PDELab::ISTLVectorBackend<1> VBE;
16  typedef Dune::PDELab::GridFunctionSpace<GV,FEM,CON,VBE> GFS;
17  GFS gfs(gv,fem);
18  typedef BCType<GV> B;
19  B b(gv);
20  b.setTime(time);                                          // b.c. depends on time now
21  typedef typename GFS::template ConstraintsContainer<Real>::Type CC;
22  CC cc;
23  Dune::PDELab::constraints(b,gfs,cc);
24
25  // <<<3>>> Make FE function with initial value / Dirichlet b.c.
26  typedef typename GFS::template VectorContainer<Real>::Type U;
27  U uold(gfs,0.0);                                          // solution at t^n
```

```
28   typedef BCExtension<GV,Real> G;                                    // defines boundary condition,
29   G g(gv);                                                           // extension and initial cond.
30   g.setTime(time);                                                   // b.c. depends on time now
31   Dune::PDELab::interpolate(g,gfs,uold);
32
33   // <<<4>>> Make instationary grid operator space
34   typedef Example03LocalOperator<B> LOP;
35   LOP lop(b,4);                                                      // local operator r
36   typedef Example03TimeLocalOperator TLOP;
37   TLOP tlop(4);                                                      // local operator m
38   typedef Dune::PDELab::ISTLBCRSMatrixBackend<1,1> MBE;
39   typedef Dune::PDELab::InstationaryGridOperatorSpace<Real,U,GFS,GFS,LOP,TLOP,CC,CC,MBE> IGOS;
40   IGOS igos(gfs,cc,gfs,cc,lop,tlop);                                 // new grid operator space
41
42   // <<<5>>> Select a linear solver backend
43   typedef Dune::PDELab::ISTLBackend_SEQ_BCGS_SSOR LS;
44   LS ls(5000,false);
45
46   // <<<6>>> Solver for linear problem per stage
47   typedef Dune::PDELab::StationaryLinearProblemSolver<IGOS,LS,U> PDESOLVER;
48   PDESOLVER pdesolver(igos,ls,1e-10);
49
50   // <<<7>>> time-stepper
51   Dune::PDELab::Alexander2Parameter<Real> method;                    // defines coefficients
52   Dune::PDELab::OneStepMethod<Real,IGOS,PDESOLVER,U,U> osm(method,igos,pdesolver);
53   osm.setVerbosityLevel(2);                                          // time stepping scheme
54
55   // <<<8>>> graphics for initial guess
56   Dune::PDELab::FilenameHelper fn("example03_Q2");                   // append number to file name
57   {
58     typedef Dune::PDELab::DiscreteGridFunction<GFS,U> DGF;
59     DGF udgf(gfs,uold);
60     Dune::SubsamplingVTKWriter<GV> vtkwriter(gv,3);
61     vtkwriter.addVertexData(new Dune::PDELab::VTKGridFunctionAdapter<DGF>(udgf,"solution"));
62     vtkwriter.write(fn.getName(),Dune::VTKOptions::binaryappended);
63     fn.increment();                                                  // increase file number
64   }
65
66   // <<<9>>> time loop
67   U unew(gfs,0.0);                                                   // solution to be computed
68   while (time<tend-1e-8) {
69       // do time step
70       b.setTime(time+dt);                                            // compute constraints
71       cc.clear();                                                    // for this time step
72       Dune::PDELab::constraints(b,gfs,cc);
73       osm.apply(time,dt,uold,g,unew);                                // do one time step
74
75       // graphics
76       typedef Dune::PDELab::DiscreteGridFunction<GFS,U> DGF;
77       DGF udgf(gfs,unew);
78       Dune::SubsamplingVTKWriter<GV> vtkwriter(gv,3);
79       vtkwriter.addVertexData(new Dune::PDELab::VTKGridFunctionAdapter<DGF>(udgf,"solution"));
80       vtkwriter.write(fn.getName(),Dune::VTKOptions::binaryappended);
81       fn.increment();
82
83       uold = unew;                                                   // advance time step
84       time += dt;
85     }
86 }
```

Listing 21 (File examples/example03_bctype.hh).

```
1 /** \brief boundary grid function selecting boundary conditions 0=Neumann, 1=Dirichlet */
2 template<typename GV>
3 class BCType : public Dune::PDELab::BoundaryGridFunctionBase<
```

```
4 Dune::PDELab::BoundaryGridFunctionTraits<GV,int,1,Dune::FieldVector<int,1> >,BCType<GV> >
5 {
6   const GV& gv; double time;
7 public:
8   typedef Dune::PDELab::BoundaryGridFunctionTraits<GV,int,1,Dune::FieldVector<int,1> > Traits;
9
10  //! construct from grid view
11  BCType (const GV& gv_) : gv(gv_) {}
12
13  //! return bc type at point on intersection
14  template<typename I>
15  inline void evaluate (I& i, const typename Traits::DomainType& xlocal,
16                        typename Traits::RangeType& y) const {
17    Dune::FieldVector<typename GV::Grid::ctype,GV::dimension> x = i.geometry().global(xlocal);
18    if (x[0]>1.0-1e-6) y = 0; else  y = 1; return;
19  }
20
21  //! get a reference to the grid view
22  inline const GV& getGridView () {return gv;}
23
24  //! set time for subsequent evaluation
25  void setTime (double t) {time = t;}
26 };
```

Listing 22 (File examples/example03_bcextension.hh).

```
1 /** \brief A function that defines Dirichlet boundary conditions AND its extension to the interior
2  */
3 template<typename GV, typename RF>
4 class BCExtension
5   : public Dune::PDELab::GridFunctionBase<Dune::PDELab::
6             GridFunctionTraits<GV,RF,1,Dune::FieldVector<RF,1> >, BCExtension<GV,RF> >
7 {
8   const GV& gv;
9   RF time;
10 public:
11  typedef Dune::PDELab::GridFunctionTraits<GV,RF,1,Dune::FieldVector<RF,1> > Traits;
12
13  //! construct from grid view
14  BCExtension (const GV& gv_) : gv(gv_) {}
15
16  //! evaluate extended function on element
17  inline void evaluate (const typename Traits::ElementType& e,
18                        const typename Traits::DomainType& xlocal,
19                        typename Traits::RangeType& y) const
20  {
21    const int dim = Traits::GridViewType::Grid::dimension;
22    typedef typename Traits::GridViewType::Grid::ctype ctype;
23    Dune::FieldVector<ctype,dim> x = e.geometry().global(xlocal);
24    if (x[0]<1E-6 && x[1]>0.25-1e-6 && x[1]<0.5+1e-6)
25      y = sin(time);
26    else
27      y = 0.0;
28    return;
29  }
30
31  //! get a reference to the grid view
32  inline const GV& getGridView () {return gv;}
33
34  //! set time for subsequent evaluation
35  void setTime (double t) {time = t;}
36 };
```

Local Operator for Spatial Part About example03_operator.hh

- Local operators have the same interface as in the stationary case extend by some additional methods:
  - setTime(T t) – set time for subsequent evaluations.
  - preStep(T time, T dt, **int** stages) – called once at beginning of time step.
  - postStep() – called once at end of time step.
  - preStage(T time, **int** i) – called once at begin of stage.
  - postStage() – called once at end of stage.
  - T suggestTimestep(T dt) – optimal time step for explicit methods.
- If operator does not depend on time import default implementation from base class.
- Allows easy reuse of existing stationary operators.

Listing 23 (File examples/example03_operator.hh).

```
1#include"example02_operator.hh"
2#include<dune/pdelab/localoperator/idefault.hh>
3
4/** a local operator for solving the equation
5 *
6 *  \partial_t u - \Delta u + a*u = f    in \Omega
7 *                              u = g    on \Gamma_D\subseteq\partial\Omega
8 *             - \nabla u \cdot n = j    on \Gamma_N = \partial\Omega\setminus\Gamma_D
9 *                              u = u_0  at t=t_0
10 *
11 * (spatial part!) with conforming finite elements on all types of grids in any dimension
12 *
13 * \tparam B a function indicating the type of boundary condition
14 */
15template<class B>
16class Example03LocalOperator :
17  public Example02LocalOperator<B>,
18  public Dune::PDELab::InstationaryLocalOperatorDefaultMethods<double> // default methods
19{
20  B& b;
21public:
22  Example03LocalOperator (B& b_, unsigned int intorder_=2)
23    : Example02LocalOperator<B>(b_,intorder_), b(b_) {}
24  void preStep (double time, double dt, int stages) {
25    b.setTime(time); // enable change of boundary condition type
26    Dune::PDELab::InstationaryLocalOperatorDefaultMethods<double>::preStep(time,dt,stages);
27  }
28};
```

Listing 24 (File examples/example03_toperator.hh).

```
1#include<dune/grid/common/genericreferenceelements.hh>
2#include<dune/grid/common/quadraturerules.hh>
3#include<dune/pdelab/common/geometrywrapper.hh>
4#include<dune/pdelab/localoperator/pattern.hh>
5#include<dune/pdelab/localoperator/flags.hh>
6#include<dune/pdelab/localoperator/idefault.hh>
7
8/** a local operator for the mass operator (L_2 integral)
9 *
10 * \f{align*}{
```

```
11  \int_\Omega uv dx
12  * \f}
13  */
14 class Example03TimeLocalOperator
15   : public Dune::PDELab::NumericalJacobianApplyVolume<Example03TimeLocalOperator>,
16     public Dune::PDELab::NumericalJacobianVolume<Example03TimeLocalOperator>,
17     public Dune::PDELab::FullVolumePattern,
18     public Dune::PDELab::LocalOperatorDefaultFlags,
19     public Dune::PDELab::InstationaryLocalOperatorDefaultMethods<double>
20 {
21 public:
22   // pattern assembly flags
23   enum { doPatternVolume = true };
24
25   // residual assembly flags
26   enum { doAlphaVolume = true };
27
28   Example03TimeLocalOperator (unsigned int intorder_=2)
29     : intorder(intorder_), time(0.0)
30   {}
31
32   //! set time for subsequent evaluation
33   void setTime (double t) {time = t;}
34
35   // volume integral depending on test and ansatz functions
36   template<typename EG, typename LFSU, typename X, typename LFSV, typename R>
37   void alpha_volume (const EG& eg, const LFSU& lfsu, const X& x, const LFSV& lfsv, R& r) const
38   {
39     // domain and range field type
40     typedef typename LFSU::Traits::LocalFiniteElementType::
41       Traits::LocalBasisType::Traits::DomainFieldType DF;
42     typedef typename LFSU::Traits::LocalFiniteElementType::
43       Traits::LocalBasisType::Traits::RangeFieldType RF;
44     typedef typename LFSU::Traits::LocalFiniteElementType::
45       Traits::LocalBasisType::Traits::RangeType RangeType;
46     typedef typename LFSU::Traits::SizeType size_type;
47
48     // dimensions
49     const int dim = EG::Geometry::dimension;
50     const int dimw = EG::Geometry::dimensionworld;
51
52     // select quadrature rule
53     Dune::GeometryType gt = eg.geometry().type();
54     const Dune::QuadratureRule<DF,dim>& rule = Dune::QuadratureRules<DF,dim>::rule(gt,intorder);
55
56     // loop over quadrature points
57     for (typename Dune::QuadratureRule<DF,dim>::const_iterator it=rule.begin(); it!=rule.end(); ++it)
58       {
59         // evaluate basis functions
60         std::vector<RangeType> phi(lfsu.size());
61         lfsu.localFiniteElement().localBasis().evaluateFunction(it->position(),phi);
62
63         // evaluate u
64         RF u=0.0;
65         for (size_type i=0; i<lfsu.size(); i++)
66           u += x[i]*phi[i];
67
68         // u*phi_i
69         RF factor = it->weight() * eg.geometry().integrationElement(it->position());
70         for (size_type i=0; i<lfsu.size(); i++)
71           r[i] += u*phi[i]*factor;
72       }
73   }
74 private:
```

```
75   unsigned int intorder;
76   double time;
77 };
```

6 Solving Systems

6.1 Composite Function Spaces

Model Problem in Mixed Formulation

Rewrite elliptic model problem as a system of first order equations:

$$\begin{aligned} \sigma + \nabla u &= 0 && \text{in } \Omega, \\ \nabla \cdot \sigma &= f && \text{in } \Omega, \\ u &= g && \text{on } \Gamma_D \subseteq \partial\Omega, \\ \sigma \cdot n &= j && \text{on } \Gamma_N = \partial\Omega \setminus \Gamma_D. \end{aligned}$$

Weak formulation. Define $H(\mathrm{div};\Omega)$ and its subspace:

$$S = \{\sigma \in (L_2(\Omega))^d \mid \nabla \cdot \sigma \in L_2(\Omega)\}, \qquad \tilde{S} = \{\sigma \in S \mid \text{“}\sigma \cdot n = 0\text{” on } \Gamma_N\}.$$

Find $(\sigma, u) \in (w + \tilde{S}) \times L_2(\Omega)$ (w extension of j !) s. t.

$$\begin{aligned} \int_\Omega \sigma \cdot v \, dx - \int_\Omega u \, \nabla \cdot v \, dx &= -\int_{\Gamma_D} g v \cdot n \, ds && \forall v \in \tilde{S} \\ -\int_\Omega \nabla \cdot \sigma \, q \, dx &= -\int_\Omega f q \, ds && \forall q \in L_2(\Omega) \end{aligned}$$

$H(\mathbf{div};\Omega)$ Conforming Finite Elements

Raviart-Thomas space of lowest order on triangles is

$$S_h = \left\{ \sigma \in (L_2(\Omega))^2 \;\middle|\; \sigma|_{\Omega_e} = \begin{pmatrix} a_e \\ b_e \end{pmatrix} + c_e \begin{pmatrix} x \\ y \end{pmatrix} \quad \forall e \in E_h^0 \right\}$$

with subspace $\tilde{S}_h = \{\sigma_h \in S_h \mid \text{“}\sigma_h \cdot n = 0\text{” on } \Gamma_N\}$.

Define the residual form

$$\begin{aligned} r^{\text{mixed}}((\sigma,u),(v,q)) = &\int_\Omega \sigma \cdot v \, dx - \int_\Omega u \, \nabla \cdot v \, dx - \int_\Omega \nabla \cdot \sigma \, q \, dx \\ &+ \int_{\Gamma_D} g v \cdot n \, ds + \int_\Omega f q \, ds. \end{aligned}$$

Discrete problem in residual form reads

$$(\sigma_h, u_h) \in (w_h + \tilde{S}_h) \times W_h : \quad r_h^{\text{mixed}}((\sigma_h,u_h),(v,q)) = 0 \quad \forall (v,q) \in \tilde{S}_h \times W_h.$$

Systems of PDEs lead to tensor product function spaces !

Function Space Composition

Given $k > 1$ function spaces $U_0, \ldots, U_{k-1}$ we define the composite function space

$$U = U_0 \times U_1 \times \ldots \times U_{k-1}.$$

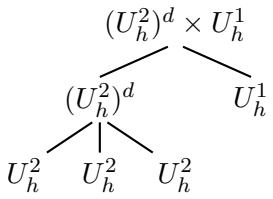
If all component spaces are the same we can also write

$$U = V^k$$

This can be done *recursively*. E.g. for solving the Stokes equation in dimension d using Taylor-Hood elements we would require

$$U_h^{\mathrm{TH}} = \left(U_h^2\right)^d \times U_h^1$$

Composite function spaces have tree structure:



Implementation of Function Space Composition

- Dune::PDELab::PowerGridFunctionSpace<GFS,k,M> builds a new grid function space that is k-times the product of GFS.
- D ...:: CompositeGridFunctionSpace<M,GFS0,...,GFS8> builds a new grid function space out of up to 9 existing grid function spaces.
- M controls construction of local to global map.
- This can be applied recursively leading to a type tree.
- Leaves of the tree are scalar or vector-valued finite element spaces.
- The local function spaces given to local operators reflects this tree structure.
- Dune::PDELab::GridFunctionSubSpace allows the selection of subspaces.
- Grid functions can be composed in a similar way.

Now the Taylor-Hood interpolation example.

Listing 25 (File examples/thinterpolate.hh).

```
1#include<dune/grid/io/file/vtk/vtkwriter.hh>
2#include<dune/pdelab/common/vtkexport.hh>
3#include<dune/pdelab/common/function.hh>
4#include<dune/pdelab/gridfunctionspace/gridfunctionspace.hh>
5#include<dune/pdelab/gridfunctionspace/gridfunctionspaceutilities.hh>
6#include<dune/pdelab/gridfunctionspace/interpolate.hh>
7#include<dune/pdelab/finiteelementmap/q1fem.hh>
8#include<dune/pdelab/finiteelementmap/q22dfem.hh>
9
10template<class GV>
```

```
11 void thinterpolate (const GV& gv)
12 {
13   // types
14   typedef typename GV::Grid::ctype D;
15   typedef double R;
16   const int dim = GV::dimension;
17
18   //  make Q_1 grid function space
19   typedef Dune::PDELab::Q1LocalFiniteElementMap<D,R,dim> Q1FEM;
20   Q1FEM q1fem;                          // Q1 finite elements
21   typedef Dune::PDELab::GridFunctionSpace<GV,Q1FEM> Q1GFS;
22   Q1GFS q1gfs(gv,q1fem);                // Q1 space
23
24   // make Q_2 grid function spaces
25   typedef Dune::PDELab::Q22DLocalFiniteElementMap<D,R> Q22DFEM;
26   Q22DFEM q22dfem;                      // Q2 finite elements, no 3D :-(
27   typedef Dune::PDELab::GridFunctionSpace<GV,Q22DFEM> Q2GFS;
28   Q2GFS q2gfs(gv,q22dfem);              // Q2 space
29
30   // make velocity grid function space
31   typedef Dune::PDELab::PowerGridFunctionSpace<Q2GFS,dim> VGFS;
32   VGFS vgfs(q2gfs);                     // velocity space
33
34   // make Taylor-Hood grid function space
35   typedef Dune::PDELab::CompositeGridFunctionSpace<
36     Dune::PDELab::GridFunctionSpaceLexicographicMapper,
37     VGFS,Q1GFS> THGFS;
38   THGFS thgfs(vgfs,q1gfs);              // Taylor-Hood space
39
40   // make coefficent vector
41   typedef typename THGFS::template VectorContainer<R>::Type X;
42   X x(thgfs,0.0);                       // one x for all dofs !
43
44   // interpolate from analytic function
45   typedef U<GV,R> Pressure;
46   Pressure p(gv);                       // pressure component
47   typedef V<GV,R> Velocity;
48   Velocity v(gv);                       // velocity component
49   typedef Dune::PDELab::CompositeGridFunction<Velocity,Pressure> THF;
50   THF thf(v,p);
51   Dune::PDELab::interpolate(thf,thgfs,x);
52
53   // select subspaces
54   typedef Dune::PDELab::GridFunctionSubSpace<THGFS,0> VSUB;
55   VSUB vsub(thgfs);                     // velocity subspace
56   typedef Dune::PDELab::GridFunctionSubSpace<VSUB,0> V0SUB;
57   V0SUB v0sub(vsub);                     // first velocity component
58   typedef Dune::PDELab::GridFunctionSubSpace<THGFS,1> PSUB;
59   PSUB psub(thgfs);                     // pressure subspace
60
61   // make discrete function objects
62   typedef Dune::PDELab::VectorDiscreteGridFunction<VSUB,X> VDGF;
63   VDGF vdgf(vsub,x);
64   typedef Dune::PDELab::DiscreteGridFunction<V0SUB,X> V0DGF;
65   V0DGF v0dgf(v0sub,x);
66   typedef Dune::PDELab::DiscreteGridFunction<PSUB,X> PDGF;
67   PDGF pdgf(psub,x);
68
69   // output grid functions with VTKWriter
70   Dune::VTKWriter<GV> vtkwriter(gv,Dune::VTKOptions::conforming);
71   vtkwriter.addVertexData(
72     new Dune::PDELab::VTKGridFunctionAdapter<VDGF>(vdgf,"velocity"));
73   vtkwriter.addVertexData(
74     new Dune::PDELab::VTKGridFunctionAdapter<V0DGF>(v0dgf,"velo␣0"));
```

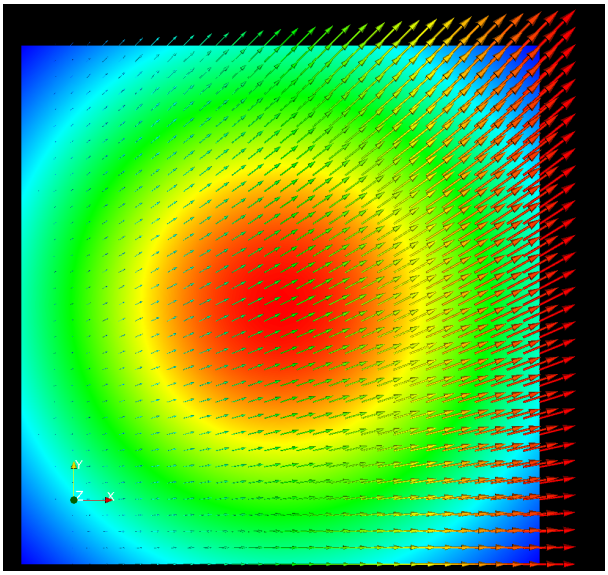


Figure 3: Visualization of velocity field and pressure in the Taylor-Hood example.

```
75   vtkwriter.addVertexData(
76     new Dune::PDELab::VTKGridFunctionAdapter<PDGF>(pdgf,"pressure"));
77   vtkwriter.write("thinterpolate",Dune::VTKOptions::ascii);
78 }
```

Figure 3 visualizes the result of this example.

For completeness here is the definition of the analytic velocity field for interpolation.

Listing 26 (File examples/thvelocity.hh)**.**

```
1#include<dune/pdelab/common/function.hh>
2template<typename GV, typename R>
3class V : public Dune::PDELab::AnalyticGridFunctionBase<
4   Dune::PDELab::AnalyticGridFunctionTraits<GV,R,GV::dimension>,
5   V<GV,R> > {
6public:
7   typedef Dune::PDELab::
8     AnalyticGridFunctionTraits<GV,R,GV::dimension> Traits;
9   typedef Dune::PDELab::
10    AnalyticGridFunctionBase<Traits,V<GV,R> > BaseT;
11
12  V (const GV& gv) : BaseT(gv) {}
13
14  inline void evaluateGlobal (const typename Traits::DomainType& x,
15                              typename Traits::RangeType& y) const {
16    y[0]=x[0]; for (int i=1; i<GV::dimension; i++) y[i]=x[i]*y[i-1];
17  }
18};
```

6.2 Example 5

Problem Definition

We solve a two-component diffusion-reaction problem with FitzHugh-Nagumo reaction[1]:

$$\begin{aligned} \partial_t u_0 - d_0 \Delta u_0 - f(u_0) + \sigma u_1 &= 0 && \text{in } \Omega = (0,2)^2, \\ \tau \partial_t u_1 - d_1 \Delta u_1 - u_0 + u_1 &= 0 && \text{in } \Omega, \\ \nabla u_0 \cdot n = \nabla u_1 \cdot n &= 0 && \text{on } \partial\Omega, \\ u_0(\cdot, t_0) &= U_0(\cdot), \\ u_1(\cdot, t_0) &= U_1(\cdot), \end{aligned}$$

with $f(u) = \lambda u - u^3 - \kappa$. We will combine

- Conforming finite elements: Q_1, Q_2.
- Various implicit time discretizations.
- Newton's method.

Example 5 Overview

Example 5 solves the diffusion reaction problem with conforming finite elements.

It consists of the following files:

- example05.cc – the file to be compiled, main function.
- example05_Q1Q1.hh – driver using Q_1 elements.
- example05_Q2Q2.hh – driver using Q_2 elements.
- example05_operator.hh – local operator for spatial part.
- example05_toperator.hh – local operator for temporal part.
- example05_initial.hh – set up initial conditions.

Main Features of the Driver About example05_Q1Q1.hh

- Make a composit grid function space with two components.
- Use GridFunctionSpaceBlockwiseMapper to order both degrees of freedom of the system consecutively.
- Local operator of the system takes a number of parameters.
- Graphical output requires subspaces to extract the components.
- Start with small time steps that are increased later.

Listing 27 (File examples/example05_Q1Q1.hh).

```
1 template<class GV>
2 void example05_Q1Q1 (const GV& gv, double dtstart, double dtmax, double tend) {
3   // <<<1>>> Choose domain and range field type
4   typedef typename GV::Grid::ctype Coord;
5   typedef double Real;
```

[1] http://en.wikipedia.org/wiki/Reaction-diffusion_system

```
6  const int dim = GV::dimension;
7  Real time = 0.0;
8
9  // <<<2>>> Make grid function space for the system
10 typedef Dune::PDELab::Q1LocalFiniteElementMap<Coord,Real,dim> FEM0;
11 FEM0 fem0;
12 typedef Dune::PDELab::NoConstraints CON;                    // pure Neumann: no constraints
13 typedef Dune::PDELab::ISTLVectorBackend<2> VBE;             // block size 2
14 typedef Dune::PDELab::GridFunctionSpace<GV,FEM0,CON,VBE> GFS0;
15 GFS0 gfs0(gv,fem0);
16
17 typedef Dune::PDELab::Q1LocalFiniteElementMap<Coord,Real,dim> FEM1;
18 FEM1 fem1;                                                  // might use Q2 as well
19 typedef Dune::PDELab::GridFunctionSpace<GV,FEM1,CON,VBE> GFS1;
20 GFS1 gfs1(gv,fem1);
21
22 typedef Dune::PDELab::CompositeGridFunctionSpace<          // compose function space
23 Dune::PDELab::GridFunctionSpaceBlockwiseMapper,GFS0,GFS1> GFS;// point block ordering
24 GFS gfs(gfs0,gfs1);
25 typedef typename GFS::template ConstraintsContainer<Real>::Type CC;
26
27 typedef Dune::PDELab::GridFunctionSubSpace<GFS,0> U0SUB;    // subspaces for later use
28 U0SUB u0sub(gfs);
29 typedef Dune::PDELab::GridFunctionSubSpace<GFS,1> U1SUB;
30 U1SUB u1sub(gfs);
31
32 // <<<3>>> Make FE function with initial value
33 typedef typename GFS::template VectorContainer<Real>::Type U;
34 U uold(gfs,0.0);
35 typedef U0Initial<GV,Real> U0InitialType;
36 U0InitialType u0initial(gv);
37 typedef U1Initial<GV,Real> U1InitialType;
38 U1InitialType u1initial(gv);
39 typedef Dune::PDELab::CompositeGridFunction<U0InitialType,U1InitialType> UInitialType;
40 UInitialType uinitial(u0initial,u1initial);
41 Dune::PDELab::interpolate(uinitial,gfs,uold);
42
43 // <<<4>>> Make instationary grid operator space
44 Real d_0 = 0.00028, d_1 = 0.005, lambda = 1.0, sigma = 1.0, kappa = -0.05, tau = 0.1;
45 typedef Example05LocalOperator LOP;
46 LOP lop(d_0,d_1,lambda,sigma,kappa,2);                      // spatial part
47 typedef Example05TimeLocalOperator TLOP;
48 TLOP tlop(tau,2);                                          // temporal part
49 typedef Dune::PDELab::ISTLBCRSMatrixBackend<2,2> MBE;
50 typedef Dune::PDELab::InstationaryGridOperatorSpace<Real,U,GFS,GFS,LOP,TLOP,CC,CC,MBE> IGOS;
51 IGOS igos(gfs,gfs,lop,tlop);
52
53 // <<<5>>> Select a linear solver backend
54 typedef Dune::PDELab::ISTLBackend_SEQ_BCGS_SSOR LS;
55 LS ls(5000,false);
56
57 // <<<6>>> Solver for non-linear problem per stage
58 typedef Dune::PDELab::Newton<IGOS,LS,U> PDESOLVER;
59 PDESOLVER pdesolver(igos,ls);
60 pdesolver.setReassembleThreshold(0.0);
61 pdesolver.setVerbosityLevel(2);
62 pdesolver.setReduction(1e-10);
63 pdesolver.setMinLinearReduction(1e-4);
64 pdesolver.setMaxIterations(25);
65 pdesolver.setLineSearchMaxIterations(10);
66
67 // <<<7>>> time-stepper
68 Dune::PDELab::Alexander2Parameter<Real> method;
69 Dune::PDELab::OneStepMethod<Real,IGOS,PDESOLVER,U,U> osm(method,igos,pdesolver);
```

```
70  osm.setVerbosityLevel(2);
71
72  // <<<8>>> graphics for initial guess
73  Dune::PDELab::FilenameHelper fn("example05_Q1Q1");
74  {
75    typedef Dune::PDELab::DiscreteGridFunction<U0SUB,U> U0DGF;
76    U0DGF u0dgf(u0sub,uold);
77    typedef Dune::PDELab::DiscreteGridFunction<U1SUB,U> U1DGF;
78    U1DGF u1dgf(u1sub,uold);
79    Dune::VTKWriter<GV> vtkwriter(gv,Dune::VTKOptions::conforming);
80    vtkwriter.addVertexData(new Dune::PDELab::VTKGridFunctionAdapter<U0DGF>(u0dgf,"u0"));
81    vtkwriter.addVertexData(new Dune::PDELab::VTKGridFunctionAdapter<U1DGF>(u1dgf,"u1"));
82    vtkwriter.write(fn.getName(),Dune::VTKOptions::binaryappended);
83    fn.increment();
84  }
85
86  // <<<9>>> time loop
87  U unew(gfs,0.0);
88  unew = uold;
89  double dt = dtstart;
90  while (time<tend-1e-8)
91    {
92      // do time step
93      osm.apply(time,dt,uold,unew);
94
95      // graphics
96      typedef Dune::PDELab::DiscreteGridFunction<U0SUB,U> U0DGF;
97      U0DGF u0dgf(u0sub,unew);
98      typedef Dune::PDELab::DiscreteGridFunction<U1SUB,U> U1DGF;
99      U1DGF u1dgf(u1sub,unew);
100     Dune::VTKWriter<GV> vtkwriter(gv,Dune::VTKOptions::conforming);
101     vtkwriter.addVertexData(new Dune::PDELab::VTKGridFunctionAdapter<U0DGF>(u0dgf,"u0"));
102     vtkwriter.addVertexData(new Dune::PDELab::VTKGridFunctionAdapter<U1DGF>(u1dgf,"u1"));
103     vtkwriter.write(fn.getName(),Dune::VTKOptions::binaryappended);
104     fn.increment();
105
106     uold = unew;
107     time += dt;
108     if (dt<dtmax-1e-8) dt = std::min(dt*1.1,dtmax);          // time step adaption
109   }
110 }
```

New Features of the Local Operators About example05_operator.hh, example05_toperator.hh

- Local function spaces (trial and test) are now composite as well.
- They reflect the structure of the grid function spaces.
- Components are extracted with template magic.
- localIndex() method on local function space maps a degree of freedom from this space to all degrees of freedom of the element.
- This local operator would work also when u_0 and u_1 are discretized with different finite element spaces.
- Same features are used in the temporal part.

Listing 28 (File examples/example05_operator.hh).

```
1#include<dune/grid/common/genericreferenceelements.hh>
2#include<dune/grid/common/quadraturerules.hh>
3#include<dune/pdelab/common/geometrywrapper.hh>
4#include<dune/pdelab/localoperator/pattern.hh>
5#include<dune/pdelab/localoperator/flags.hh>
6#include<dune/pdelab/localoperator/idefault.hh>
7
8/** A local operator for solving the two-component reaction-diffusion
9 *  system of Fitzhugh-Nagumo type. See also
10 *  http://en.wikipedia.org/wiki/Reaction-diffusion
11 *
12 *   - d_0 \Delta u_0 - (\lambda*u_0 - u_0^3 - \sigma* u_1 + \kappa) = 0   in \Omega
13 *   - d_1 \Delta u_1 - (u_0 - u_1)                                  = 0   in \Omega
14 *
15 * with natural boundary conditions
16 *   \nabla u_0 \cdot n = 0   on \partial\Omega
17 *   \nabla u_1 \cdot v = 0   on \partial\Omega
18 *
19 * with conforming finite elements on all types of grids in any dimension
20 */
21class Example05LocalOperator :
22  public Dune::PDELab::NumericalJacobianApplyVolume<Example05LocalOperator >,
23  public Dune::PDELab::NumericalJacobianVolume<Example05LocalOperator >,
24  public Dune::PDELab::FullVolumePattern,
25  public Dune::PDELab::LocalOperatorDefaultFlags,
26  public Dune::PDELab::InstationaryLocalOperatorDefaultMethods<double>
27{
28public:
29  // pattern assembly flags
30  enum { doPatternVolume = true };
31
32  // residual assembly flags
33  enum { doAlphaVolume = true };
34
35  // constructor stores parameters
36  Example05LocalOperator (double d_0_, double d_1_, double lambda_, double sigma_,
37                          double kappa_, unsigned int intorder_=2)
38    : d_0(d_0_), d_1(d_1_), lambda(lambda_), sigma(sigma_),
39      kappa(kappa_), intorder(intorder_)
40  {}
41
42  // volume integral depending on test and ansatz functions
43  template<typename EG, typename LFSU, typename X, typename LFSV, typename R>
44  void alpha_volume (const EG& eg, const LFSU& lfsu, const X& x, const LFSV& lfsv, R& r) const
45  {
46    // select the two components (assume Galerkin scheme U=V)
47    typedef typename LFSU::template Child<0>::Type LFSU0;        // extract components
48    const LFSU0& lfsu0 = lfsu.template getChild<0>();            // with template magic
49    typedef typename LFSU::template Child<1>::Type LFSU1;
50    const LFSU1& lfsu1 = lfsu.template getChild<1>();
51
52    // domain and range field type (assume both components have same RF)
53    typedef typename LFSU0::Traits::LocalFiniteElementType::
54      Traits::LocalBasisType::Traits::DomainFieldType DF;
55    typedef typename LFSU0::Traits::LocalFiniteElementType::
56      Traits::LocalBasisType::Traits::RangeFieldType RF;
57    typedef typename LFSU0::Traits::LocalFiniteElementType::
58      Traits::LocalBasisType::Traits::JacobianType JacobianType;
59    typedef typename LFSU0::Traits::LocalFiniteElementType::
60      Traits::LocalBasisType::Traits::RangeType RangeType;
61    typedef typename LFSU::Traits::SizeType size_type;
62
63    // dimensions
64    const int dim = EG::Geometry::dimension;
```

```
65      const int dimw = EG::Geometry::dimensionworld;
66
67      // select quadrature rule
68      Dune::GeometryType gt = eg.geometry().type();
69      const Dune::QuadratureRule<DF,dim>& rule = Dune::QuadratureRules<DF,dim>::rule(gt,intorder);
70
71      // loop over quadrature points
72      for (typename Dune::QuadratureRule<DF,dim>::const_iterator
73             it=rule.begin(); it!=rule.end(); ++it)
74        {
75          // evaluate basis functions on reference element
76          std::vector<RangeType> phi0(lfsu0.size());
77          lfsu0.localFiniteElement().localBasis().evaluateFunction(it->position(),phi0);
78          std::vector<RangeType> phi1(lfsu1.size());
79          lfsu1.localFiniteElement().localBasis().evaluateFunction(it->position(),phi1);
80
81          // compute u_0, u_1 at integration point
82          RF u_0=0.0;
83          for (size_type i=0; i<lfsu0.size(); i++)
84        u_0 += x[lfsu0.localIndex(i)]*phi0[i];                  // localIndex() maps dof within
85          RF u_1=0.0;                                               // leaf space to all dofs
86          for (size_type i=0; i<lfsu1.size(); i++)                  // within given element
87        u_1 += x[lfsu1.localIndex(i)]*phi1[i];
88
89          // evaluate gradient of basis functions on reference element
90          std::vector<JacobianType> js0(lfsu0.size());
91          lfsu0.localFiniteElement().localBasis().evaluateJacobian(it->position(),js0);
92          std::vector<JacobianType> js1(lfsu1.size());
93          lfsu1.localFiniteElement().localBasis().evaluateJacobian(it->position(),js1);
94
95          // transform gradients from reference element to real element
96          const Dune::FieldMatrix<DF,dimw,dim>
97            jac = eg.geometry().jacobianInverseTransposed(it->position());
98          std::vector<Dune::FieldVector<RF,dim> > gradphi0(lfsu0.size());
99          for (size_type i=0; i<lfsu0.size(); i++)
100           jac.mv(js0[i][0],gradphi0[i]);
101         std::vector<Dune::FieldVector<RF,dim> > gradphi1(lfsu1.size());
102         for (size_type i=0; i<lfsu1.size(); i++)
103           jac.mv(js1[i][0],gradphi1[i]);
104
105         // compute gradient of u_0, u_1
106         Dune::FieldVector<RF,dim> gradu0(0.0);
107         for (size_type i=0; i<lfsu0.size(); i++)
108           gradu0.axpy(x[lfsu0.localIndex(i)],gradphi0[i]);
109         Dune::FieldVector<RF,dim> gradu1(0.0);
110         for (size_type i=0; i<lfsu1.size(); i++)
111           gradu1.axpy(x[lfsu1.localIndex(i)],gradphi1[i]);
112
113         // integrate both components
114         RF factor = it->weight()*eg.geometry().integrationElement(it->position());
115         // eq. 0: - d_0 \Delta u_0 - (\lambda*u_0 - u_0^3 - \sigma* u_1 + \kappa) = 0
116         for (size_type i=0; i<lfsu0.size(); i++)
117           r[lfsu0.localIndex(i)] += (d_0*(gradu0*gradphi0[i])
118             -(lambda*u_0-u_0*u_0*u_0-sigma*u_1+kappa)*phi0[i])*factor;
119         // eq. 1: - d_1 \Delta u_1 - (u_0 - u_1) = 0
120         for (size_type i=0; i<lfsu1.size(); i++)
121           r[lfsu1.localIndex(i)] += (d_1*(gradu1*gradphi1[i])
122             -(u_0-u_1)*phi1[i])*factor;
123       }
124   }
125
126 private:
127   unsigned int intorder;
128   double d_0, d_1, lambda, sigma, kappa;
```

```
129 };
```

Listing 29 (File examples/example05_toperator.hh).

```
 1#include<dune/grid/common/genericreferenceelements.hh>
 2#include<dune/grid/common/quadraturerules.hh>
 3#include<dune/pdelab/common/geometrywrapper.hh>
 4#include<dune/pdelab/localoperator/pattern.hh>
 5#include<dune/pdelab/localoperator/flags.hh>
 6#include<dune/pdelab/localoperator/idefault.hh>
 7
 8/** \brief A local operator for the mass operator (L_2 integral) in the system */
 9class Example05TimeLocalOperator
10  : public Dune::PDELab::NumericalJacobianApplyVolume<Example05TimeLocalOperator>,
11    public Dune::PDELab::NumericalJacobianVolume<Example05TimeLocalOperator>,
12    public Dune::PDELab::FullVolumePattern,
13    public Dune::PDELab::LocalOperatorDefaultFlags,
14    public Dune::PDELab::InstationaryLocalOperatorDefaultMethods<double>
15{
16public:
17  // pattern assembly flags
18  enum { doPatternVolume = true };
19
20  // residual assembly flags
21  enum { doAlphaVolume = true };
22
23  // constructor remembers parameters
24  Example05TimeLocalOperator (double tau_, unsigned int intorder_=2)
25    : tau(tau_), intorder(intorder_) {}
26
27  // volume integral depending on test and ansatz functions
28  template<typename EG, typename LFSU, typename X, typename LFSV, typename R>
29  void alpha_volume (const EG& eg, const LFSU& lfsu, const X& x, const LFSV& lfsv, R& r) const
30  {
31    // select the two components (assume Galerkin scheme U=V)
32    typedef typename LFSU::template Child<0>::Type LFSU0;
33    const LFSU0& lfsu0 = lfsu.template getChild<0>();
34    typedef typename LFSU::template Child<1>::Type LFSU1;
35    const LFSU1& lfsu1 = lfsu.template getChild<1>();
36
37    // domain and range field type (assume both components have same RF)
38    typedef typename LFSU0::Traits::LocalFiniteElementType::
39      Traits::LocalBasisType::Traits::DomainFieldType DF;
40    typedef typename LFSU0::Traits::LocalFiniteElementType::
41      Traits::LocalBasisType::Traits::RangeFieldType RF;
42    typedef typename LFSU0::Traits::LocalFiniteElementType::
43      Traits::LocalBasisType::Traits::JacobianType JacobianType;
44    typedef typename LFSU0::Traits::LocalFiniteElementType::
45      Traits::LocalBasisType::Traits::RangeType RangeType;
46    typedef typename LFSU::Traits::SizeType size_type;
47
48    // dimensions
49    const int dim = EG::Geometry::dimension;
50    const int dimw = EG::Geometry::dimensionworld;
51
52    // select quadrature rule
53    Dune::GeometryType gt = eg.geometry().type();
54    const Dune::QuadratureRule<DF,dim>& rule = Dune::QuadratureRules<DF,dim>::rule(gt,intorder);
55
56    // loop over quadrature points
57    for (typename Dune::QuadratureRule<DF,dim>::const_iterator it=rule.begin(); it!=rule.end(); ++it)
58      {
59        // evaluate basis functions on reference element
60        std::vector<RangeType> phi0(lfsu0.size());
61        lfsu0.localFiniteElement().localBasis().evaluateFunction(it->position(),phi0);
```

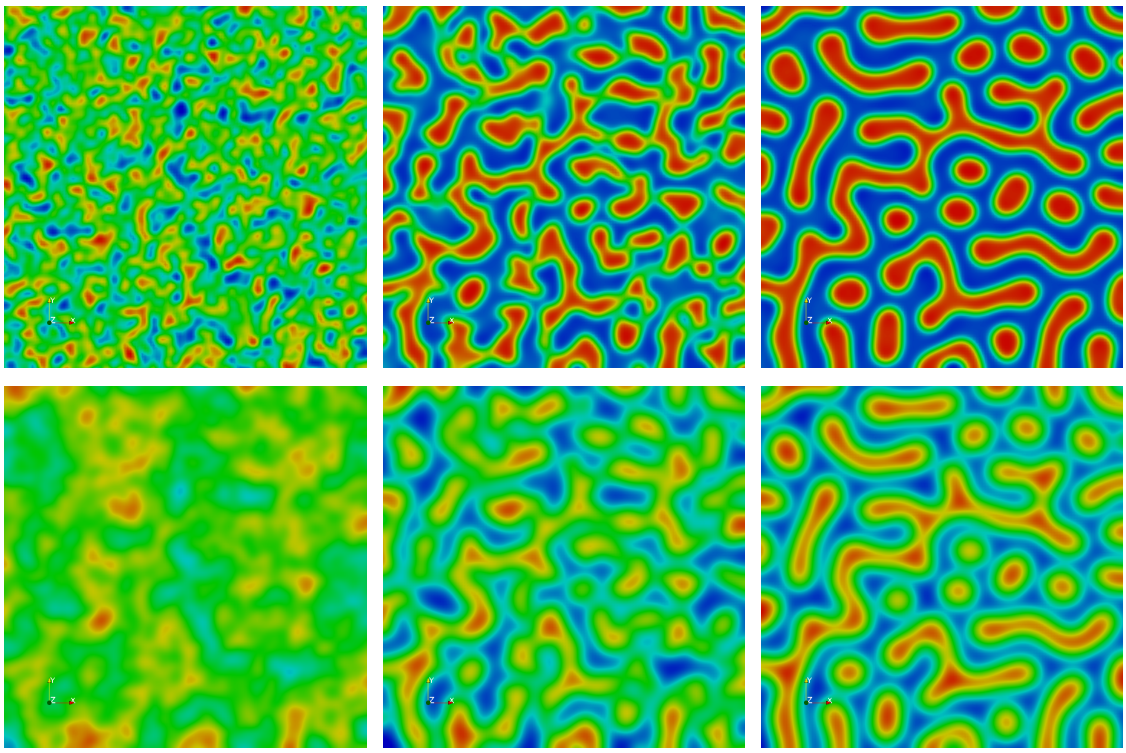


Figure 4: Results for example 5 computed with Q_1 elements on a 128×128 grid at times $0.5, 5$ and 50 (left to right) and components 0 and 1 (top to bottom). Parameters of the FitzHugh-Nagumo model were $d_0 = 0.00028$, $d_1 = 0.005$, $\lambda = 1$, $\sigma = 1$, $\kappa = -0.05$ and $\tau = 0.1$.

```
62          std::vector<RangeType> phi1(lfsu1.size());
63          lfsu1.localFiniteElement().localBasis().evaluateFunction(it->position(),phi1);
64
65          // compute u_0, u_1 at integration point
66          RF u_0=0.0;
67          for (size_type i=0; i<lfsu0.size(); i++) u_0 += x[lfsu0.localIndex(i)]*phi0[i];
68          RF u_1=0.0;
69          for (size_type i=0; i<lfsu1.size(); i++) u_1 += x[lfsu1.localIndex(i)]*phi1[i];
70
71          // integration
72          RF factor = it->weight() * eg.geometry().integrationElement(it->position());
73          for (size_type i=0; i<lfsu0.size(); i++)
74            r[lfsu0.localIndex(i)] += u_0*phi0[i]*factor;
75          for (size_type i=0; i<lfsu1.size(); i++)
76            r[lfsu1.localIndex(i)] += tau*u_1*phi1[i]*factor;
77        }
78    }
79  private:
80    double tau;
81    unsigned int intorder;
82  };
```

Figure 4 shows visualizations of the results computed with example05.

6.3 Example 6

Example 6 Overview

We demonstrate orthogonality of concepts by parallelizing example 5.

Example 6 consist of the following files

- example06.cc – the file to be compiled, main function.
- example06_Q1Q1.hh – parallel driver for *overlapping* grids.
- example06_bctype.hh – dummy boundary condition type for overlapping constraints class.

Main Features of the Parallel Driver About example06_Q1Q1.hh

- Needs OverlappingConformingDirichletConstraints to assemble Dirichlet constraints at artificial processor boundaries.
- Needs boundary condition type function to call constraints() function.
- Operators and initial conditions are taken from example 5.
- Select a parallel solver backend for overlapping grids.
- Invoke driver with a parallel overlapping grid (see example06.cc).
- Thats it.

Listing 30 (File examples/example06_Q1Q1.hh).

```
1 template<class GV>
2 void example06_Q1Q1 (const GV& gv, double dtstart, double dtmax, double tend)
3 {
4   // <<<1>>> Choose domain and range field type
5   typedef typename GV::Grid::ctype Coord;
6   typedef double Real;
7   const int dim = GV::dimension;
8   Real time = 0.0;
9
10  // <<<2>>> Make grid function space for the system
11  typedef Dune::PDELab::Q1LocalFiniteElementMap<Coord,Real,dim> FEM0;
12  FEM0 fem0;
13  typedef Dune::PDELab::OverlappingConformingDirichletConstraints CON; // new constraints class
14  CON con;
15  typedef Dune::PDELab::ISTLVectorBackend<2> VBE;
16  typedef Dune::PDELab::GridFunctionSpace<GV,FEM0,CON,VBE> GFS0;
17  GFS0 gfs0(gv,fem0,con);
18
19  typedef Dune::PDELab::Q1LocalFiniteElementMap<Coord,Real,dim> FEM1;
20  FEM1 fem1;
21  typedef Dune::PDELab::GridFunctionSpace<GV,FEM1,CON,VBE> GFS1;
22  GFS1 gfs1(gv,fem1,con);
23
24  typedef Dune::PDELab::CompositeGridFunctionSpace<
25    Dune::PDELab::GridFunctionSpaceBlockwiseMapper,
26      GFS0,GFS1> GFS;
27  GFS gfs(gfs0,gfs1);
28
29  typedef BCType<GV> U0BC;
30  U0BC u0bc(gv);
```

```
31  typedef Dune::PDELab::PowerGridFunction<U0BC,2> UBC;
32  UBC ubc(u0bc);
33  typedef typename GFS::template ConstraintsContainer<Real>::Type CC;
34  CC cc;                                                    // constraints needed due
35  Dune::PDELab::constraints(ubc,gfs,cc);                    // to artificial boundaries
36
37  typedef Dune::PDELab::GridFunctionSubSpace<GFS,0> U0SUB;
38  U0SUB u0sub(gfs);
39  typedef Dune::PDELab::GridFunctionSubSpace<GFS,1> U1SUB;
40  U1SUB u1sub(gfs);
41
42  // <<<3>>> Make FE function with initial value
43  typedef typename GFS::template VectorContainer<Real>::Type U;
44  U uold(gfs,0.0);
45  typedef U0Initial<GV,Real> U0InitialType;
46  U0InitialType u0initial(gv);
47  typedef U1Initial<GV,Real> U1InitialType;
48  U1InitialType u1initial(gv);
49  typedef Dune::PDELab::CompositeGridFunction<U0InitialType,U1InitialType> UInitialType;
50  UInitialType uinitial(u0initial,u1initial);
51  Dune::PDELab::interpolate(uinitial,gfs,uold);
52
53  // <<<4>>> Make instationary grid operator space
54  Real d_0 = 0.00028, d_1 = 0.005;
55  Real lambda = 1.0, sigma = 1.0, kappa = -0.05, tau = 0.1;
56  typedef Example05LocalOperator LOP;
57  LOP lop(d_0,d_1,lambda,sigma,kappa,2);
58  typedef Example05TimeLocalOperator TLOP;
59  TLOP tlop(tau,2);
60  typedef Dune::PDELab::ISTLBCRSMatrixBackend<2,2> MBE;
61  typedef Dune::PDELab::InstationaryGridOperatorSpace<Real,U,GFS,GFS,LOP,TLOP,CC,CC,MBE> IGOS;
62  IGOS igos(gfs,cc,gfs,cc,lop,tlop);
63
64  // <<<5>>> Select a linear solver backend
65  typedef Dune::PDELab::ISTLBackend_OVLP_BCGS_SSORk<GFS,CC> LS; // select parallel backend !
66  LS ls(gfs,cc,5000,5,1);
67
68  // <<<6>>> Solver for non-linear problem per stage
69  typedef Dune::PDELab::Newton<IGOS,LS,U> PDESOLVER;
70  PDESOLVER pdesolver(igos,ls);
71  pdesolver.setReassembleThreshold(0.0);
72  pdesolver.setVerbosityLevel(2);
73  pdesolver.setReduction(1e-10);
74  pdesolver.setMinLinearReduction(1e-4);
75  pdesolver.setMaxIterations(25);
76  pdesolver.setLineSearchMaxIterations(10);
77
78  // <<<7>>> time-stepper
79  Dune::PDELab::FractionalStepParameter<Real> method;
80  Dune::PDELab::OneStepMethod<Real,IGOS,PDESOLVER,U,U> osm(method,igos,pdesolver);
81  osm.setVerbosityLevel(2);
82
83  // <<<8>>> graphics for initial guess
84  Dune::PDELab::FilenameHelper fn("example06_Q1Q1");
85  {
86    typedef Dune::PDELab::DiscreteGridFunction<U0SUB,U> U0DGF;
87    U0DGF u0dgf(u0sub,uold);
88    typedef Dune::PDELab::DiscreteGridFunction<U1SUB,U> U1DGF;
89    U1DGF u1dgf(u1sub,uold);
90    Dune::VTKWriter<GV> vtkwriter(gv,Dune::VTKOptions::conforming);
91    vtkwriter.addVertexData(new Dune::PDELab::VTKGridFunctionAdapter<U0DGF>(u0dgf,"u0"));
92    vtkwriter.addVertexData(new Dune::PDELab::VTKGridFunctionAdapter<U1DGF>(u1dgf,"u1"));
93    vtkwriter.write(fn.getName(),Dune::VTKOptions::binaryappended);
94    fn.increment();
```

```
95   }
96
97   // <<<9>>> time loop
98   U unew(gfs,0.0);
99   unew = uold;
100  double dt = dtstart;
101  while (time<tend-1e-8)
102    {
103      // do time step
104      osm.apply(time,dt,uold,unew);
105
106      // graphics
107      typedef Dune::PDELab::DiscreteGridFunction<U0SUB,U> U0DGF;
108      U0DGF u0dgf(u0sub,unew);
109      typedef Dune::PDELab::DiscreteGridFunction<U1SUB,U> U1DGF;
110      U1DGF u1dgf(u1sub,unew);
111      Dune::VTKWriter<GV> vtkwriter(gv,Dune::VTKOptions::conforming);
112      vtkwriter.addVertexData(new Dune::PDELab::VTKGridFunctionAdapter<U0DGF>(u0dgf,"u0"));
113      vtkwriter.addVertexData(new Dune::PDELab::VTKGridFunctionAdapter<U1DGF>(u1dgf,"u1"));
114      vtkwriter.write(fn.getName(),Dune::VTKOptions::binaryappended);
115      fn.increment();
116
117      uold = unew;
118      time += dt;
119      if (dt<dtmax-1e-8)
120        dt = std::min(dt*1.1,dtmax);
121    }
122 }
```

Further Examples for Local Operators

in dune/pdelab/localoperator you find

- diffusiondg.hh – Stationary diffusion equation with variable coefficients, discontinuous Galerkin methods.
- diffusionmixed.hh – dito, $RT0$ mixed method.
- diffusionmfd.hh – dito, mimetic finite difference method.
- twophaseccfv.hh – water-gas flow in porous media using pressure-pressure formulation, cell-centered finite volumes.
- transportccfv.hh – Stationary and instationary convection-diffusion equation with cell-centered finite volumes.

...and more.

7 Building a Finite Element Space

7.1 General Construction

Introduction

- We consider affine finite element spaces:
  - Global functions are pieced together from local functions on each element.
  - Local functions are constructed by transformation from a reference element.
- Global functions need not be continuous and can be vector-valued.
- The functions on the reference element are collected in the module dune−localfunctions.
- Constructing the global function spaces from this is part of PDELab.

Finite-dimensional Function Spaces

$\Omega \subset \mathbb{R}^n$, $n \geq 1$, is a domain, $\mathbb{T}_h$ a grid partitioning the domain Ω.

Definition 31 (Finite-dimensional function space)**.**

$$U_h(\mathbb{T}_h) = \left\{ u_h(x) : \bigcup_{e \in E_h^0} \Omega_e \to \mathbb{K}^m \;\middle|\; u_h(x) = \sum_{e \in E_h^0} \sum_{i=0}^{k(e)-1} (\mathbf{u})_{g(e,i)}\, \pi_e(\hat{x})\, \hat{\phi}_{e,i}(\hat{x})\, \chi_e(x);\ \hat{x} = \mu_e^{-1}(x) \right\}$$

defines a general finite-dimensional function space of element-wise continuous functions. □

Building Blocks

- $\mathbb{K} = \mathbb{R}$ or $\mathbb{K} = \mathbb{C}$. $m \geq 1$ denotes vector-valued function spaces.
- $\hat{\Omega}_e$ is the *reference element* associated with element $e \in E_h^0$. $\mu_e : \overline{\hat{\Omega}}_e \to \overline{\Omega}_e$ maps the reference element to Ω.
- $\hat{\Phi}_e = \left\{ \hat{\phi}_{e,i} : \overline{\hat{\Omega}}_e \to \mathbb{R}^{m'} \,|\, 0 \leq i < k(e) \right\}$ is the set of *local basis functions* for element e.
- $\pi_e(\hat{x}) \in \mathbb{R}^{m \times m'}$ is a transformation. A non-trivial example is the Piola transformation [BF91]
$$\pi_e(\hat{x}) = \frac{1}{\det \nabla \mu_e(\hat{x})} \nabla \mu_e(\hat{x})$$
where $\nabla \mu_e$ denotes the Jacobian of the map μ_e. For most finite element spaces π_e is just the identity and we have $m' = m$.
- $g : L \to \mathbb{N}$, $L = \left\{ (e,i) \in E_h^0 \times \mathbb{N} \,|\, 0 \leq i < k(e) \right\}$ is the *local to global map* and $\mathcal{I}_{U_h} = \operatorname{im} g$ is the associated *global index set*.
- $\mathbf{u} \in \mathbf{U} = \mathbb{K}^{\mathcal{I}_{U_h}}$ is a coefficient vector.

Global Basis Functions

For $j \in \mathcal{I}_{U_h}$ we define the *global basis function*

$$U_h \ni \phi_j(x) = \sum_{(e,i)\in L(j)} \pi_e(\mu_e^{-1}(x))\, \hat{\phi}_{e,i}(\mu_e^{-1}(x))\, \chi_e(x)$$

and $L(j) = \{(e,i) \in L \,|\, g(e,i) = j\}$. With that we have

$$\Phi_{U_h} = \{\phi_i \,|\, i \in \mathcal{I}_{U_h}\}, \qquad U_h = \text{span}\ \Phi_{U_h}. \tag{14}$$

and the finite element isomorphism

$$\text{FE}_{\Phi_{U_h}} : \mathbf{U} \to U_h, \qquad \text{FE}_{\Phi_{U_h}}(\mathbf{u}) = \sum_{i\in\mathcal{I}_{U_h}} (\mathbf{u})_i \phi_i\ . \tag{15}$$

Definition 31 allows for functions that are *discontinuous* at element boundaries.

If limits on the skeleton $\Gamma_h = \Omega \setminus \bigcup_{e\in E_h^0} \Omega_e$ coincide, a function may be extended to $\left(C^0(\Omega)\right)^m$.

7.2 Local Finite Element Space

A global finite element space is built up from a collection of local finite element spaces for each element.

Local Finite Element Space

A finite element space on the reference element (compare [Cia02]) implements Dune::FiniteElementInterface:

1. The type of reference element, given by Dune::GeometryType.
2. The local basis functions $\hat{\Phi} = \{\hat{\phi}_i : \mathbb{D}^n \to \mathbb{K}^m \,|\, 0 \le i < k\}$ and possibly derivatives in a class implementing e.g C0BasisInterface.
3. Information that allows to construct the local to global map $g : (e,i) \mapsto j$ in a class implementing Dune:: LocalCoefficientsInterface .
4. A method that allows to compute coefficients z_i such that

$$\sum_{0\le i<k} z_i \hat{\phi}_i = \hat{u} \qquad \hat{u} \in \text{span}\hat{\Phi}$$

in a class implementing from Dune:: InterpolationInterface . For $\hat{u} \notin \text{span}\hat{\Phi}$ the method provides a projection.

The dune module dune−localfunctions provides a collection of local finite element spaces.

Function Traits

The local basis is required to contain a class Traits that gives the types for $\mathbb{D}, \mathbb{D}^n, \mathbb{K}, \mathbb{K}^m$, the numbers n, m and a type for the Jacobian:

```
1  template<class DF, int n, class D, class RF, int m, class R,
2           class J, int dorder=0>
3  struct LocalBasisTraits
4  {
5    typedef DF DomainFieldType;   typedef D DomainType;
6    typedef RF RangeFieldType;    typedef R RangeType;
7
8    enum { dimDomain = n }; enum { dimRange = m };
9
10   typedef J JacobianType;
11
12   enum { diffOrder=dorder };
13 };
```

Later on, other classes representing functions will use the same traits classes.

Q_1 Local Basis

We now consider the bilinear elements Q_1 as an example.

The basis functions are given by

$$\hat{\phi}_2(\hat{x}) = (1-\hat{x}_0)\hat{x}_1, \qquad \hat{\phi}_3(\hat{x}) = \hat{x}_0\hat{x}_1,$$
$$\hat{\phi}_0(\hat{x}) = (1-\hat{x}_0)(1-\hat{x}_1), \qquad \hat{\phi}_1(\hat{x}) = \hat{x}_0(1-\hat{x}_1).$$

The numbering of the basis functions corresponds to the reference quadrilateral.

Q_1 Local Basis Implementation

- The following listing gives an implementation of the Q_1 local basis functions in 2D.
- We provide also gradients (C^1 function).
- There is also an interface for higher derivatives (not shown).
- Evaluation always provides the values of *all* basis functions or gradients at *one* point (in coordinates of the reference element).
- Implementations typically use Dune::FieldVector<T,n> to represent short vectors.
- JacobianType is Dune::FieldMatrix<R,1,2>.
- In the jacobian evaluation in lines 26 to 29 we have

$$\text{out[i][j][k]} = \partial_{\hat{x}_k}\left(\hat{\phi}_i\right)_j(\hat{x}).$$

Listing 32 (File examples/q1localbasis.hh).

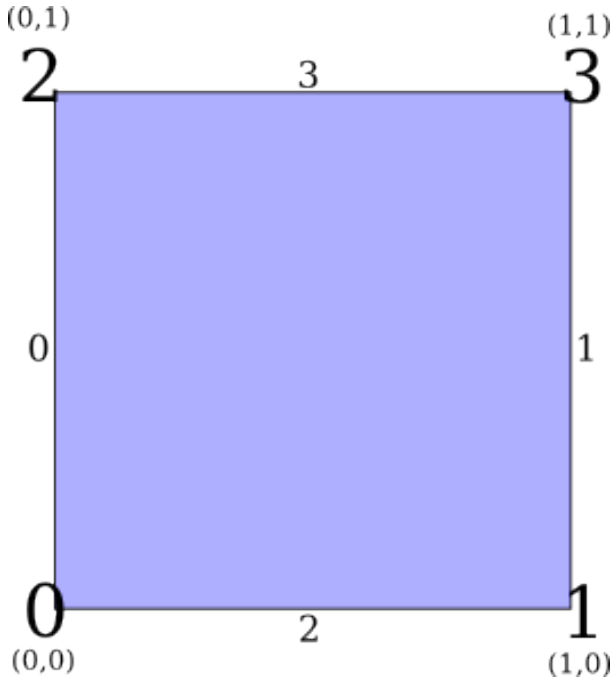


Figure 5: DUNE reference quadrilateral.

```
1 #include<dune/common/fvector.hh>
2 #include<dune/localfunctions/common/localbasis.hh>
3 template<class D, class R>
4 class Q1LocalBasis
5 {
6 public:
7   typedef Dune::LocalBasisTraits<D,2,Dune::FieldVector<D,2>,R,1,
8     Dune::FieldVector<R,1>,Dune::FieldMatrix<R,1,2>, 1> Traits;
9
10  unsigned int size () const { return 4; }
11
12  //! \brief Evaluate all shape functions
13  inline void evaluateFunction (
14                 const typename Traits::DomainType& in,
15           std::vector<typename Traits::RangeType>& out) const {
16    out.resize(4);
17    out[0] = (1-in[0])*(1-in[1]); out[1] = (  in[0])*(1-in[1]);
18    out[2] = (1-in[0])*(  in[1]); out[3] = (  in[0])*(  in[1]);
19  }
20
21  //! \brief Evaluate Jacobian of all shape functions
22  inline void
23  evaluateJacobian (const typename Traits::DomainType& in,
24           std::vector<typename Traits::JacobianType>& out) const {
25    out.resize(4);
26    out[0][0][0] = in[1]-1; out[0][0][1] = in[0]-1;
27    out[1][0][0] = 1-in[1]; out[1][0][1] = -in[0];
28    out[2][0][0] =  -in[1]; out[2][0][1] = 1-in[0];
29    out[3][0][0] =   in[1]; out[3][0][1] = in[0];
30  }
31
```

```
32   //! \brief Polynomial order of the shape functions
33   unsigned int order () const  {
34     return 1;
35   }
36 };
```

Q_1 Local Coefficients

- Piecing together local functions to global functions (globalization) may involve some form of continuity.
- Part of this is *identifying degrees of freedom*: Associate basis functions with (sub-)entities of the reference element.
- *On the reference element* each (number of a) basis function is mapped to
  - a subentity given by a number and a codimension.
  - an offset with in that entity if several indices are mapped to the same subentity (offset is zero if only one local index is mapped to the subentity).
  - Dune::LocalKey represents such a triple.
- Globalization may also involve orientation of geometric entities.
- From this information a local-to-global map g can be constructed *generically*.
- This is *not* part of dune−localfunctions!

Listing 33 (File examples/q1localcoefficients.hh).

```
1 #include<dune/localfunctions/common/localkey.hh>
2 class Q1LocalCoefficients
3 {
4 public:
5   Q1LocalCoefficients () : li(4)  {
6     for (int i=0; i<4; i++) li[i] = Dune::LocalKey(i,2,0);
7   }
8
9   //! number of coefficients
10  int size () const { return 4; }
11
12  //! map index i to local key
13  const Dune::LocalKey& localKey (int i) const  {
14    return li[i];
15  }
16
17 private:
18   std::vector<Dune::LocalKey> li;
19 };
```

Q_1 Local Interpolation

- Local interpolation takes a function $\hat{u}(\hat{x})$ *on the reference element* and provides a projection onto the space spanned by the local basis.
- It does so by providing the coefficients with respect to the basis (inverse of local finite element isomorphism).

- For Lagrange basis functions this is point-wise evaluation.
- For other basis functions this might involve the solution of a local system (e.g. L_2-projection).
- The local interpolation interface is provided by Dune::LocalInterpolationInterface.
- The given function needs to map from Traits::DomainType to Traits::RangeType like the basis.

Listing 34 (File examples/q1localinterpolation.hh).

```
1  // interface: <dune/localfunctions/common/interpolation.hh>
2  template<class LB>
3  class Q1LocalInterpolation
4  {
5  public:
6
7    //! \brief Local interpolation of a function
8    template<typename F, typename C>
9    void interpolate (const F& f, std::vector<C>& out) const {
10     typename LB::Traits::DomainType x;
11     typename LB::Traits::RangeType y;
12
13     out.resize(4);
14     x[0] = 0.0; x[1] = 0.0; f.evaluate(x,y); out[0] = y;
15     x[0] = 1.0; x[1] = 0.0; f.evaluate(x,y); out[1] = y;
16     x[0] = 0.0; x[1] = 1.0; f.evaluate(x,y); out[2] = y;
17     x[0] = 1.0; x[1] = 1.0; f.evaluate(x,y); out[3] = y;
18   }
19 };
```

Q_1 Local Finite Element

- Finally, we need to collect local basis, local coefficients and local interpolation in a local finite element.
- Local finite element also provides the reference element.
- The interface is given in Dune::FiniteElementInterface.

Listing 35 (File examples/q1localfiniteelement.hh).

```
1  #include<dune/common/geometrytype.hh>
2  #include<dune/localfunctions/common/localfiniteelementtraits.hh>
3
4  #include"q1localbasis.hh"
5  #include"q1localcoefficients.hh"
6  #include"q1localinterpolation.hh"
7
8  template<class D, class R>
9  class Q1LocalFiniteElement
10 {
11   Q1LocalBasis<D,R> basis;
12   Q1LocalCoefficients coefficients;
13   Q1LocalInterpolation<Q1LocalBasis<D,R> > interpolation;
14   Dune::GeometryType gt;
15 public:
16   typedef Dune::LocalFiniteElementTraits<Q1LocalBasis<D,R>,
17                 Q1LocalCoefficients,
18                 Q1LocalInterpolation<Q1LocalBasis<D,R> > > Traits;
19
```

```
20    Q1LocalFiniteElement () { gt.makeQuadrilateral(); }
21
22    const typename Traits::LocalBasisType& localBasis () const
23    {
24      return basis;
25    }
26
27    const typename Traits::LocalCoefficientsType& localCoefficients () const
28    {
29      return coefficients;
30    }
31
32    const typename Traits::LocalInterpolationType& localInterpolation () const
33    {
34      return interpolation;
35    }
36
37    Dune::GeometryType type () const { return gt; }
38
39    Q1LocalFiniteElement* clone () const {
40      return new Q1LocalFiniteElement(*this);
41    }
42  };
```

List of Elements

Currently (March 2010), dune−localfunctions implements the following elements:

- P_0 for any reference element in any dimension.
- Piecewise linear Lagrange elements P_1 in $1, 2, 3d$.
- Piecewise multi-linear Lagrange elements Q_1 in $1, 2, 3d$.
- P_k on triangles and tetrahedra
- Q_2 on quadrilaterals.
- Lowest order Raviart-Thomas elements on triangles, quadrilaterals, hexahedra.
- P_k discontinuous on any element in $1, 2, 3d$ (monomial and orthogonal basis).
- Rotated bilinear element in $2d$.
- Whitney elements in $2, 3d$.
- Some hierarchical macro-elements.

7.3 Building a Global Finite Element Space from Local Spaces

Local Finite Element Map

- PDELab uses classes from dune−localfunctions to piece local functions together forming global functions.
- Each codim 0 entity may have a different local basis (i.e. local finite element).
- A *local finite element map* provides this basis for each entity.

- Can handle hybrid meshes and *hp*-methods.
- To ease implementation there is a variant of the local finite element in dune−localfunctions *with virtual functions*.
- The local finite element map is responsible for ensuring the required continuity.
- The case where each codim 0 entity has the same basis is particularly easy. See next code example.

Listing 36 (File examples/q1localfiniteelementmap.hh).

```
1 #include<dune/pdelab/finiteelementmap/finiteelementmap.hh>
2 #include"q1localfiniteelement.hh"
3
4 template<class D, class R>
5 class Q1LocalFiniteElementMap
6   : public Dune::PDELab::
7       SimpleLocalFiniteElementMap< Q1LocalFiniteElement<D,R> >
8 {};
```

Q_1 Grid Function Space

- The local finite element map is a parameter to the class template GridFunctionSpace.
- The GridFunctionSpace is responsible for:
  - Building up the local-to-global-map g.
  - Provides a type for coefficient vectors ("vector container").
  - Provides information about the local finite element and degrees of freedoms on each element ("local function space").
- The remaining lines construct a function object that can be evaluated in local coordinates on the reference element (explained below). It is used by the Dune::SubsamplingVTKWriter.

Listing 37 (File examples/q1gridfunctionspace.hh).

```
1 #include<dune/grid/io/file/vtk/subsamplingvtkwriter.hh>
2 #include<dune/pdelab/common/vtkexport.hh>
3 #include<dune/pdelab/gridfunctionspace/gridfunctionspace.hh>
4 #include<dune/pdelab/gridfunctionspace/gridfunctionspaceutilities.hh>
5 #include"q1localfiniteelementmap.hh"
6
7 template<typename GV> void q1GridFunctionSpace (const GV& gv) {
8   typedef typename GV::Grid::ctype D;// domain type
9   typedef double R;                 // range type
10
11  Q1LocalFiniteElementMap<D,R> fem;  // maps entity to finite element
12
13  typedef Dune::PDELab::GridFunctionSpace<GV,
14    Q1LocalFiniteElementMap<D,R> > GFS;
15  GFS gfs(gv,fem);                  // make grid function space
16
17  typedef typename GFS::template VectorContainer<R>::Type X;
18  X x(gfs,0.0);                     // make coefficient vector
19  x[4] = 1.0;                       // set a component
20
21  typedef Dune::PDELab::DiscreteGridFunction<GFS,X> DGF;
22  DGF dgf(gfs,x);                   // make a grid function
```

```
23
24   Dune::SubsamplingVTKWriter<GV> vtkwriter(gv,3);   // plot result
25   vtkwriter.addVertexData(new Dune::PDELab::
26                           VTKGridFunctionAdapter<DGF>(dgf,"q1"));
27   vtkwriter.write("q1gridfunctionspace",Dune::VTKOptions::ascii);
28 }
```

Finally, in the main program a Dune::Grid is instantiated and the generic function is called.

Listing 38 (File examples/q1gridfunctionspacemain.cc)**.**

```
1 #ifdef HAVE_CONFIG_H
2 #include "config.h"
3 #endif
4 #include<dune/common/mpihelper.hh>
5 #include<dune/grid/yaspgrid.hh>
6 #include"q1gridfunctionspace.hh"
7 int main(int argc, char** argv)
8 {
9   try{
10    //Maybe initialize Mpi
11    Dune::MPIHelper::instance(argc, argv);
12
13    // make grid
14    Dune::FieldVector<double,2> L(1.0);
15    Dune::FieldVector<int,2> N(2);
16    Dune::FieldVector<bool,2> B(false);
17    Dune::YaspGrid<2> grid(L,N,B,0);
18    q1GridFunctionSpace(grid.leafView());
19
20    return 0;
21  }
22  catch (Dune::Exception &e){ std::cerr << "Dune␣reported␣error:␣" << e << std::endl;
23    return 1;
24  }
25  catch (...){ std::cerr << "Unknown␣exception␣thrown!" << std::endl;
26    return 1;
27  }
28 }
```

Figure 6 shows the result visualized with ParaView.

7.4 Grid Functions

Functions

Often one wants to prescribe functions for initial or boundary conditions.

PDELab offers several classes to represent functions:

- Dune::PDELab::FunctionInterface is the interface for general functions

$$u : \mathbb{D}^n \to \mathbb{K}^m, \qquad x \mapsto y.$$

- Dune::PDELab::GridFunctionInterface is the interface for functions defined on a grid:

$$u : E_h^0 \times \mathbb{D}^n \to \mathbb{K}^m, \qquad (e, \hat{x}) \mapsto y.$$

  $\hat{x}$ is on the *reference element*.

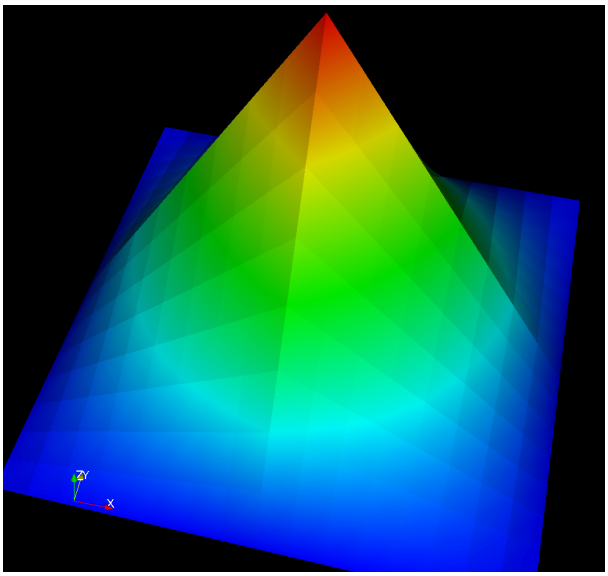


Figure 6: Q_1 global basis function visualized with ParaView.

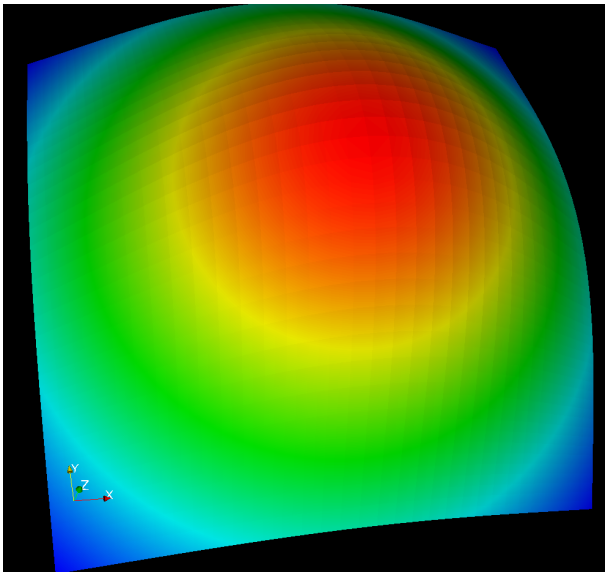


Figure 7: Interpolation of $\exp(-3\|x - c\|^2)$ with Q_1 elements.

- Dune::PDELab::BoundaryGridFunctionInterface is the interface for *grid functions* living on the boundary:
$$u : (E_h^1 \text{“}\cap\partial\Omega\text{”}) \times \mathbb{D}^{n'} \to \mathbb{K}^m, \qquad (f, \hat{x}) \mapsto y.$$
- Functions offer the same Traits as the local basis.

Useful Adapters and Base Classes

There are various useful adapters and base classes:

Dune::PDELab::DiscreteGridFunction: Takes a grid function space and a coefficient vector and makes a grid function out of it.

Dune::PDELab::VTKGridFunctionAdapter: Takes a grid function and makes a VTK function out of it (this is the input for VTKWriter). These two classes have been used already above.

Dune::PDELab::AnalyticGridFunctionBase: Implements grid function interface from global function by deriving from it. This class will be used shortly.

Less Useful Adapters

Dune::PDELab::FunctionToGridFunctionAdapter: Takes a global function and makes a grid function out of it.

Dune::PDELab::GlobalFunctionToLocalFunctionAdapter: Takes a global function and an element and provides evaluation w.r.t. reference element.

Dune::PDELab::GridFunctionToLocalFunctionAdapter: Takes grid function and element, acts as function on the reference element.

Dune::PDELab::SelectComponentAdapter. Takas vector-valued function and component number and provides a scalar function.

D ...:: BoundaryGridFunctionSelectComponentAdapter: Same for boundary grid functions.

Dune::PDELab::PiolaBackwardAdapter: Takes global vector-valued function, makes grid function tranformed back to reference element.

For more details see dune/pdelab/common/function.hh.

Listing 39 (File examples/analyticfunction.hh).

```
1 #include<math.h>
2 #include<dune/pdelab/common/function.hh>
3
4 template<typename GV, typename RF>
5 class U : public Dune::PDELab::AnalyticGridFunctionBase<
6   Dune::PDELab::AnalyticGridFunctionTraits<GV,RF,1>,
7   U<GV,RF> > {
8 public:
9   typedef Dune::PDELab::AnalyticGridFunctionTraits<GV,RF,1> Traits;
10  typedef Dune::PDELab::AnalyticGridFunctionBase<Traits,U<GV,RF> > B;
11
12  U (const GV& gv) : B(gv) {}
13  inline void evaluateGlobal (const typename Traits::DomainType& x,
14                              typename Traits::RangeType& y) const
```

```
15   {
16     typename Traits::DomainType center(0.0);
17     center -= x;
18     y = exp((-1.0/(1500*1500))*center.two_norm2());
19   }
20 };
```

Generic Interpolation

Interpolation of a finite element function from a given function is now generic.

Listing 40 (File examples/q1interpolate.hh).

```
1 #include<dune/grid/io/file/vtk/subsamplingvtkwriter.hh>
2 #include<dune/pdelab/common/vtkexport.hh>
3 #include<dune/pdelab/gridfunctionspace/gridfunctionspace.hh>
4 #include<dune/pdelab/gridfunctionspace/gridfunctionspaceutilities.hh>
5 #include<dune/pdelab/gridfunctionspace/interpolate.hh>
6
7 template<typename GV>
8 void q1interpolate (const GV& gv)
9 {
10   typedef typename GV::Grid::ctype D; // domain type
11   typedef double R;                   // range type
12
13   Q1LocalFiniteElementMap<D,R> fem;   // maps entity to finite element
14
15   typedef Dune::PDELab::GridFunctionSpace<GV,
16     Q1LocalFiniteElementMap<D,R> > GFS;
17   GFS gfs(gv,fem);                    // make grid function space
18
19   typedef typename GFS::template VectorContainer<R>::Type X;
20   X x(gfs,0.0);                       // make coefficient vector
21
22   U<GV,R> u(gv);                      // make analytic function object
23   Dune::PDELab::interpolate(u,gfs,x); // interpolate x from u
24
25   typedef Dune::PDELab::DiscreteGridFunction<GFS,X> DGF;
26   DGF dgf(gfs,x);                     // make a grid function
27
28   Dune::SubsamplingVTKWriter<GV> vtkwriter(gv,1); // plot result
29   vtkwriter.addVertexData(new Dune::PDELab::
30                           VTKGridFunctionAdapter<DGF>(dgf,"q1"));
31   vtkwriter.write("q1interpolate",Dune::VTKOptions::ascii);
32 }
```

7.5 Interpolation Error Example

Interpolation Error Example

Now, we do a little example that shows how one can work with global finite element functions.

For a given function u and its interpolant $u_h \in U_h(\mathbb{T}_h)$ we want to compute the L_2 interpolation

error

$$\|u-u_h\|_{L_2(\Omega)} = \int_\Omega (u-u_h)^2\,dx = \sum_{e\in E_h^0}\int_{\Omega_e}(u-u_h)^2\,dx$$

$$= \sum_{e\in E_h^0}\int_{\hat\Omega_e}(u(\mu_e(\hat x))-u_h(\mu_e(\hat x)))^2\det\nabla\mu_e(\hat x)\,d\hat x$$

$$= \sum_{e\in E_h^0}\sum_{j=0}^{q(e)-1} w_{e,j}\,(u(\mu_e(\hat x_{e,j}))-u_h(\mu_e(\hat x_{e,j})))^2\det\nabla\mu_e(\hat x_{e,j})\,d\hat x + \text{error}$$

$$= \sum_{e\in E_h^0}\sum_{j=0}^{q(e)-1} w_{e,j}\left(u(\mu_e(\hat x_{e,j}))-\sum_{i=0}^{k(e)-1}(\mathbf{u})_{g(e,i)}\hat\phi_{e,i}(\hat x_{e,j})\right)^2\det\nabla\mu_e(\hat x_{e,j})\,d\hat x + \text{error}.$$

In the following code example the function l2interpolationerror is parametrized by

- U: Type for a function.
- GFS: Type for a grid function space.
- X: Type for a coefficient vector.

The local function space GFS::LocalFunctionSpace in line 19 is a type exported by a grid function space which

- is bound to an element e later in line 28,
- provides the local finite element for that element e ,
- provides the local to global map $g(e,\cdot)$ for that element,
- can read, write and add degrees of freedom of element e.

Dune::QuadratureRule in line 31 provides quadrature rules for many element types, dimensions and orders.

Listing 41 (File examples/l2interpolationerror.hh).

```
1 #include<dune/common/geometrytype.hh>
2 #include<dune/grid/common/quadraturerules.hh>
3 template<class U, class GFS, class X>
4 double l2interpolationerror (const U& u, const GFS& gfs, X& x,
5                               int qorder=1) {
6   // constants and types
7   typedef typename GFS::Traits::GridViewType GV;
8   const int dim = GV::Grid::dimension;
9   typedef typename GV::Traits::template Codim<0>::Iterator
10    ElementIterator;
11  typedef typename GFS::Traits::LocalFiniteElementType::
12    Traits::LocalBasisType::Traits FTraits;
13  typedef typename FTraits::DomainFieldType D;
14  typedef typename FTraits::RangeFieldType R;
15  typedef typename FTraits::RangeType RangeType;
16
17  // make local function space
18  typedef typename GFS::LocalFunctionSpace LFS;
```

```
19   LFS lfs(gfs);
20   std::vector<R> xl(lfs.maxSize());          // local coefficients
21   std::vector<RangeType> b(lfs.maxSize()); // shape function values
22
23   // loop over grid view
24   double sum = 0.0;
25   for (ElementIterator eit = gfs.gridview().template begin<0>();
26        eit!=gfs.gridview().template end<0>(); ++eit)
27     {
28       lfs.bind(*eit);        // bind local function space to element
29       lfs.vread(x,xl);       // get local coefficients
30       Dune::GeometryType gt = eit->geometry().type();
31       const Dune::QuadratureRule<D,dim>&
32         rule = Dune::QuadratureRules<D,dim>::rule(gt,qorder);
33
34       for (typename Dune::QuadratureRule<D,dim>::const_iterator
35              qit=rule.begin();
36            qit!=rule.end(); ++qit)
37         {
38           // evaluate finite element function at integration point
39           RangeType u_fe(0.0);
40           lfs.localFiniteElement().localBasis().evaluateFunction(
41             qit->position(),b);
42           for (int i=0; i<lfs.size(); i++)
43             u_fe.axpy(xl[i],b[i]);
44
45           // evaluate the given grid function at integration point
46           RangeType u_given;
47           u.evaluate(*eit,qit->position(),u_given);
48
49           // accumulate error
50           u_fe -= u_given;
51           sum += u_fe.two_norm2()*qit->weight()*
52             eit->geometry().integrationElement(qit->position());
53         }
54     }
55   return sqrt(sum);
56 }
```

Next comes the driver that uses the generic L_2 interpolation error function.

Listing 42 (File examples/q1interpolationerror.hh).

```
1 #include<dune/grid/io/file/vtk/subsamplingvtkwriter.hh>
2 #include<dune/pdelab/common/vtkexport.hh>
3 #include<dune/pdelab/gridfunctionspace/gridfunctionspace.hh>
4 #include<dune/pdelab/gridfunctionspace/gridfunctionspaceutilities.hh>
5 #include<dune/pdelab/gridfunctionspace/interpolate.hh>
6 #include"l2interpolationerror.hh"
7
8 template<typename GV>
9 void q1interpolationerror (const GV& gv)
10 {
11   typedef typename GV::Grid::ctype D; // domain type
12   typedef double R;                   // range type
13
14   Q1LocalFiniteElementMap<D,R> fem;   // maps entity to finite element
15
16   typedef Dune::PDELab::GridFunctionSpace<GV,
17     Q1LocalFiniteElementMap<D,R> > GFS;
18   GFS gfs(gv,fem);                    // make grid function space
19
20   typedef typename GFS::template VectorContainer<R>::Type X;
21   X x(gfs,0.0);                       // make coefficient vector
```

```
22
23   U<GV,R> u(gv);                          // make analytic function object
24   Dune::PDELab::interpolate(u,gfs,x); // make x interpolate u
25
26   std::cout.precision(8);
27   std::cout << "interpolation␣error:␣"
28             << std::setw(8) << gv.size(0) << "␣elements␣"
29             << std::scientific << l2interpolationerror(u,gfs,x,4) << std::endl;
30 }
```

Q_1 Interpolation Error Results

Evaluating the interpolaton error for our Q_1 basis for different levels of refinement produces the following result:

```
interpolation error:        1 elements 4.63081768e-01
interpolation error:        4 elements 1.02612039e-01
interpolation error:       16 elements 3.03000305e-02
interpolation error:       64 elements 7.77518775e-03
interpolation error:      256 elements 1.95618596e-03
interpolation error:     1024 elements 4.89819655e-04
interpolation error:     4096 elements 1.22503221e-04
interpolation error:    16384 elements 3.06288243e-05
interpolation error:    65536 elements 7.65739477e-06
interpolation error:   262144 elements 1.91436048e-06
interpolation error:  1048576 elements 4.78590858e-07
```

Obviously, we get second order approximation.

Modification for Q_2

Now it is very easy to compute the interpolation error for other function spaces.

Just replace Q1LocalFiniteElementMap with Dune::PDELab::Q22DLocalFiniteElementMap in lines 15 and 18!

Listing 43 (File examples/q2interpolationerror.hh).

```
1 #include<dune/grid/io/file/vtk/subsamplingvtkwriter.hh>
2 #include<dune/pdelab/common/vtkexport.hh>
3 #include<dune/pdelab/gridfunctionspace/gridfunctionspace.hh>
4 #include<dune/pdelab/gridfunctionspace/gridfunctionspaceutilities.hh>
5 #include<dune/pdelab/gridfunctionspace/interpolate.hh>
6 #include<dune/pdelab/finiteelementmap/q22dfem.hh>
7 #include"l2interpolationerror.hh"
8
9 template<typename GV>
10 void q2interpolationerror (const GV& gv)
11 {
12   typedef typename GV::Grid::ctype D; // domain type
13   typedef double R;                   // range type
14
15   Dune::PDELab::Q22DLocalFiniteElementMap<D,R> fem; // Q_2 now !
16
17   typedef Dune::PDELab::GridFunctionSpace<GV,
18     Dune::PDELab::Q22DLocalFiniteElementMap<D,R> > GFS;
19   GFS gfs(gv,fem);                    // make grid function space
20
21   typedef typename GFS::template VectorContainer<R>::Type X;
22   X x(gfs,0.0);                       // make coefficient vector
```

```
23
24   U<GV,R> u(gv);                          // make analytic function object
25   Dune::PDELab::interpolate(u,gfs,x); // make x interpolate u
26
27   std::cout.precision(8);
28   std::cout << "interpolation␣error:␣"
29             << std::setw(8) << gv.size(0) << "␣elements␣"
30             << std::scientific << l2interpolationerror(u,gfs,x,4) << std::endl;
31 }
```

Q_2 Interpolation Error Result

And we see third order approximation:

```
interpolation error:         1 elements 4.65290291e-02
interpolation error:         4 elements 1.20509875e-02
interpolation error:        16 elements 1.36558258e-03
interpolation error:        64 elements 1.71127393e-04
interpolation error:       256 elements 2.14102072e-05
interpolation error:      1024 elements 2.67692161e-06
interpolation error:      4096 elements 3.34635709e-07
interpolation error:     16384 elements 4.18301070e-08
interpolation error:     65536 elements 5.22878350e-09
interpolation error:    262144 elements 6.53598587e-10
interpolation error:   1048576 elements 8.16998215e-11
```

8 More on Constrained Function Spaces

8.1 How to Construct Constrained Spaces

As we have seen we often have the situation that problems have to be solved in a subspace $\tilde{U}_h \subset U_h$ or even an affine subspace $w_h + \tilde{U}_h$, where $w_h \in U_h$.

Basis transformation

Definition 44 (Basis Transformation)**.** Consider an alternative basis $\Phi'_{U_h} = \{\phi'_i \,|\, i \in \mathcal{I}_{U_h}\}$ of U_h obtained by

$$\phi'_i = \sum_{j \in \mathcal{I}_{U_h}} (\mathbf{T}_{U_h})_{i,j}\, \phi_j, \qquad i \in \mathcal{I}_{U_h}. \tag{16}$$

$\mathbf{T}_{U_h}$ is the *transformation matrix.* □

For $\mathbf{U}' = \mathbb{K}^{\mathcal{I}_{U_h}}$ we have the isomorphism $\mathrm{FE}_{\Phi'_{U_h}}(\mathbf{u}') = \sum_{i \in \mathcal{I}_{U_h}} (\mathbf{u}')_i \phi'_i$ and get

$$\begin{aligned} u_h = \mathrm{FE}_{\Phi'_{U_h}}(\mathbf{u}') &= \sum_{j \in \mathcal{I}_{\tilde{U}_h}} (\mathbf{u}')_j \phi'_j = \sum_{j \in \mathcal{I}_{U_h}} (\mathbf{u}')_j \left(\sum_{i \in \mathcal{I}_{U_h}} (\mathbf{T}_{U_h})_{j,i}\, \phi_i \right) \\ &= \sum_{i \in \mathcal{I}_{U_h}} \left(\sum_{j \in \mathcal{I}_{U_h}} \left(\mathbf{T}_{U_h}^T\right)_{i,j} (\mathbf{u}')_j \right) \phi_i = \mathrm{FE}_{\Phi_{U_h}}\left(\mathbf{T}_{U_h}^T \mathbf{u}'\right). \end{aligned} \tag{17}$$

Splitting and Subspaces Subspaces are introduced by a splitting of the index set into unconstrained and constrained indices:

$$\mathcal{I}_{U_h} = \tilde{\mathcal{I}}_{U_h} \cup \bar{\mathcal{I}}_{U_h}, \qquad \tilde{\mathcal{I}}_{U_h} \cap \bar{\mathcal{I}}_{U_h} = \emptyset.$$

With respect to this splitting we define the subspaces

$$\tilde{U}'_h = \mathrm{span}\,\{\phi'_i \,|\, i \in \tilde{\mathcal{I}}_{U_h}\}, \qquad \bar{U}'_h = \mathrm{span}\,\{\phi'_i \,|\, i \in \bar{\mathcal{I}}_{U_h}\}$$

and the corresponding coefficient spaces

$$\tilde{\mathbf{U}}' = \mathbb{K}^{\tilde{\mathcal{I}}_{U_h}}, \qquad \bar{\mathbf{U}}' = \mathbb{K}^{\bar{\mathcal{I}}_{U_h}}.$$

$\tilde{U}_h := \tilde{U}'_h \subseteq U_h$ is the desired subspace of U_h where we want to solve the constrained problem.

The transformation matrix $\mathbf{T}_{U_h}$ is written in block form w.r.t. the splitting:

$$\mathbf{T}_{U_h} = \begin{pmatrix} \mathbf{T}_{\tilde{U}_h,\tilde{U}_h} & \mathbf{T}_{\tilde{U}_h,\bar{U}_h} \\ \mathbf{T}_{\bar{U}_h,\tilde{U}_h} & \mathbf{T}_{\bar{U}_h,\bar{U}_h} \end{pmatrix}.$$

We show below that it suffices to consider transformations of the form

$$\mathbf{T}_{U_h} = \begin{pmatrix} \mathbf{I} & \mathbf{T}_{\tilde{U}_h,\bar{U}_h} \\ \mathbf{0} & \mathbf{I} \end{pmatrix} \tag{18}$$

which means transformations have the form

$$\phi_i' = \phi_i + \sum_{j \in \bar{\mathcal{I}}_{U_h}} \left(\mathbf{T}_{\tilde{U}_h,\bar{U}_h}\right)_{i,j} \phi_j, \qquad i \in \tilde{\mathcal{I}}_{U_h}.$$

The $\tilde{\mathcal{I}}_{U_h} \times \bar{\mathcal{I}}_{U_h}$ matrix $\mathbf{T}_{\tilde{U}_h,\bar{U}_h}$ will usually be very sparse (or even zero).

Restrictions Below we will make use of the following restriction operators

$$\begin{aligned} &\mathbf{R}_{\tilde{\mathbf{U}}',\mathbf{U}'} : \mathbf{U}' \to \tilde{\mathbf{U}}', && (\mathbf{R}_{\tilde{\mathbf{U}}',\mathbf{U}'}\mathbf{u}')_i = (\mathbf{u}')_i \quad \forall i \in \tilde{\mathcal{I}}_{U_h}, \\ &\mathbf{R}_{\bar{\mathbf{U}}',\mathbf{U}'} : \mathbf{U}' \to \bar{\mathbf{U}}', && (\mathbf{R}_{\bar{\mathbf{U}}',\mathbf{U}'}\mathbf{u}')_i = (\mathbf{u}')_i \quad \forall i \in \bar{\mathcal{I}}_{U_h}. \end{aligned}$$

Note that $\mathbf{Q}_{\tilde{\mathbf{U}}} = \mathbf{R}^T_{\tilde{\mathbf{U}}',\mathbf{U}'}\mathbf{R}_{\tilde{\mathbf{U}}',\mathbf{U}'}$ is an orthogonal projection.

It can be used to project a function from U_h to $\tilde{U}_h$ as follows:

$$P_h : U_h \to \tilde{U}_h, \qquad P_h = FE_{\Phi_h'} \circ \mathbf{Q}_{\tilde{\mathbf{U}}} \circ FE^{-1}_{\Phi_h'}.$$

The projection P_h will play a major role below.

8.2 Examples of Constrained Spaces

Dirichlet Boundary Conditions U_h^1 : piecewise-linear, conforming finite element functions.
$\tilde{U}_h^1 = \{u \in U_h^1 \,|\, \text{“}u(x) = 0\text{” for } x \in \Gamma_D\}$.
Then just set

$$\bar{\mathcal{I}}_{U_h} = \{i \in \mathcal{I}_{U_h} \,|\, x_{z_i} \in \Gamma_D\}, \qquad \tilde{\mathcal{I}}_{U_h} = \mathcal{I}_{U_h} \setminus \bar{\mathcal{I}}_{U_h}$$

and

$$\phi_i' = \phi_i \qquad i \in \tilde{\mathcal{I}}_{U_h}.$$

Obviously, $\mathbf{T}_{\tilde{U}_h,\bar{U}_h} = \mathbf{0}$ in this case.

Hanging Nodes Consider a mesh obtained from non-conforming refinement.

U_h^1 now is the space of piecewise linear functions with degrees of freedom in *all* vertices of the mesh. To obtain the subspace $\tilde{U}_h^1 = U_h^1 \cap C^0(\Omega)$ we set

$$\begin{aligned} \bar{\mathcal{I}}_{U_h} &= \{i \in \mathcal{I}_{U_h} \,|\, z_i \text{ is a hanging node}\}, \\ \tilde{\mathcal{I}}_{U_h} &= \mathcal{I}_{U_h} \setminus \bar{\mathcal{I}}_{U_h}. \end{aligned}$$

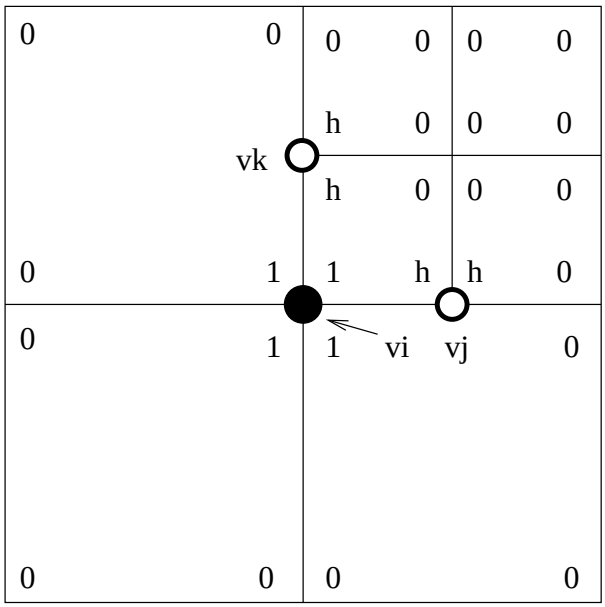


Figure 8: Non-conforming refinement and piecewise linear finite elements.

For the vertex i in the figure we obtain, e. g., the basis function

$$\phi_i' = \phi_i + \frac{1}{2}\phi_j + \frac{1}{2}\phi_k.$$

Pure Neumann Problem We wish to solve

$$\begin{aligned} -\Delta u &= f && \text{in } \Omega \subseteq \mathbb{R}^n, \\ -\nabla u \cdot \nu &= j && \text{on } \partial\Omega \end{aligned}$$

where $\int_{\partial\Omega} j\, ds = \int_\Omega f\, dx$.

Again, consider U_h^1, the piecewise-linear, conforming finite element functions.

Then u is only defined up to a constant and we wish to solve in

$$\tilde{U}_h^1 = \left\{ u \in U_h^1 \,\middle|\, \int_\Omega u\, dx = 0 \right\}.$$

Set $\bar{\mathcal{I}}_{U_h} = \{0\}$, $\tilde{\mathcal{I}}_{U_h} = \mathcal{I}_{U_h} \setminus \{0\}$ and

$$\phi_i' = \phi_i - \frac{\int_\Omega \phi_i\, dx}{\int_\Omega \phi_0\, dx}\phi_0, \qquad \forall i \in \tilde{\mathcal{I}}_{U_h}.$$

Other Situations Where Constraints Occur In addition to Dirichlet, hanging nodes and pure Neumann constraints we can also treat

- Varying polynomial degree in conforming finite elements (p-refinement).
- Combination of p-refinement, non-conforming refinement and essential boundary conditions.
- Non-conforming refinement in mixed finite elements.
- Periodic boundary conditions.
- Constraints in parallel overlapping Schwarz methods.

8.3 Implementation of Dirichlet Constraints

We now show how constraints can be added to a grid function space.

Constraints are a property of a grid function space.

In addition to the local finite elements we have to parametrize Dune::PDELab::GridFunctionSpace with a type that can provide the sparse transformation matrix $\mathbf{T}_{\tilde{U}_h,\bar{U}_h}$ from (18).

Information should be provided only locally. One column of $\mathbf{T}_{\tilde{U}_h,\bar{U}_h}$ may only involve degrees of freedom of two (intersecting) elements.

A constraints class provides rows of $\mathbf{T}^T_{\tilde{U}_h,\bar{U}_h}$ and may have methods

- volume: Constrain degrees of freedom associated with volume (useful in parallelization).
- skeleton: Constrain degrees of freedom associated with interior intersections (useful for hanging nodes).
- boundary: Constrain degrees of freedom on boundary intersections (for boundary conditions).

Flags **do**... control which methods must be provided.

Listing 45 (File examples/q1constraints.hh)**.**

```
1  class Q1Constraints {
2  public:
3    enum{doBoundary=true};enum{doProcessor=false};
4    enum{doSkeleton=false};enum{doVolume=false};
5
6    template<typename B, typename I, typename LFS, typename T>
7    void boundary (const B& b, const I& ig, const LFS& lfs, T& trafo) const
8    {
9      typename B::Traits::DomainType ip(0.5); // test edge midpoint
10     typename B::Traits::RangeType bctype;   // return value
11     b.evaluate(ig,ip,bctype);               // eval condition type
12
13     if (bctype<=0) return;                  // done
14
15     typename T::RowType empty;              // need not interpolate
16     if (ig.indexInInside()==0) { trafo[0]=empty; trafo[2]=empty; }
17     if (ig.indexInInside()==1) { trafo[1]=empty; trafo[3]=empty; }
18     if (ig.indexInInside()==2) { trafo[0]=empty; trafo[1]=empty; }
19     if (ig.indexInInside()==3) { trafo[2]=empty; trafo[3]=empty; }
```

```
20   }
21 };
```

To use assembling of constraints we define a boundary grid function that gives the *boundary condition type* for a point on the boundary.

Note that RangeFieldType is set to **int** in line 7.

Listing 46 (File examples/boundaryconditiontypefunction.hh).

```
1 #include<dune/common/fvector.hh>
2 #include<dune/pdelab/common/function.hh>
3
4 template<typename GV>
5 class B
6   : public Dune::PDELab::BoundaryGridFunctionBase<Dune::PDELab::
7   BoundaryGridFunctionTraits<GV,int,1,Dune::FieldVector<int,1> >,
8                                                          B<GV> >
9 {
10
11 public:
12   typedef Dune::PDELab::BoundaryGridFunctionTraits<GV,int,1,
13     Dune::FieldVector<int,1> > Traits;
14   typedef Dune::PDELab::BoundaryGridFunctionBase<Traits,B<GV> > BaseT;
15
16   B (const GV& gv_) : gv(gv_) {}
17
18   template<typename I>
19   inline void evaluate (const I& ig,
20                         const typename Traits::DomainType& x,
21                         typename Traits::RangeType& y) const {
22     Dune::FieldVector<typename GV::Grid::ctype,GV::dimension>
23       xg = ig.geometry().global(x);
24
25     y = 0; // no Dirichlet
26     if (xg[0]<1E-6 && xg[1]>0.25 && xg[1]<0.75 ) return;
27     y = 1; // Dirichlet
28   }
29
30   inline const GV& getGridView () {
31     return gv;
32   }
33 private:
34   const GV& gv;
35 };
```

In the following listing we redo the interpolation example with a constrained finite element space.

In line 17 the grid function space is parametrized with the constraints class.

In line 20 the grid function space exports a type to hold the transformation matrix $\mathbf{T}^T_{\tilde{U}_h,\bar{U}_h}$.

In line 22 the function giving the boundary condition type is instantiated.

Finally, in line 23 the constraints are assembled and stored.

Function Dune::PDELab::set_nonconstrained_dofs in line 30 allows to set all nonconstrained degrees of freedom to a given value.

Function Dune::PDELab::set_constrained_dofs does the same for the constrained degrees of freedom.

Listing 47 (File examples/q1constrainedinterpolate.hh).

```
1 #include<dune/grid/io/file/vtk/subsamplingvtkwriter.hh>
2 #include<dune/pdelab/common/vtkexport.hh>
3 #include<dune/pdelab/gridfunctionspace/gridfunctionspace.hh>
4 #include<dune/pdelab/gridfunctionspace/gridfunctionspaceutilities.hh>
5 #include<dune/pdelab/gridfunctionspace/interpolate.hh>
6 #include<dune/pdelab/gridfunctionspace/constraints.hh>
7
8 template<typename GV>
9 void q1interpolate (const GV& gv)
10 {
11   typedef typename GV::Grid::ctype D; // domain type
12   typedef double R;                   // range type
13
14   Q1LocalFiniteElementMap<D,R> fem;  // map entity to finite element
15
16   typedef Dune::PDELab::GridFunctionSpace<GV,
17     Q1LocalFiniteElementMap<D,R>,Q1Constraints> GFS;
18   GFS gfs(gv,fem);                    // make grid function space
19
20   typedef typename GFS::template ConstraintsContainer<R>::Type T;
21   T t;                                // container for transformation
22   B<GV> b(gv);                        // boundary condition function
23   Dune::PDELab::constraints(b,gfs,t);// fill container
24
25   typedef typename GFS::template VectorContainer<R>::Type X;
26   X x(gfs,0.0);                       // make coefficient vector
27
28   U<GV,R> u(gv);                      // analytic function object
29   Dune::PDELab::interpolate(u,gfs,x);// interpolate x from u
30   Dune::PDELab::set_nonconstrained_dofs(t,0.0,x); // clear interior
31
32   typedef Dune::PDELab::DiscreteGridFunction<GFS,X> DGF;
33   DGF dgf(gfs,x);                      // make a grid function
34
35   Dune::SubsamplingVTKWriter<GV> vtkwriter(gv,1); // plot result
36   vtkwriter.addVertexData(new Dune::PDELab::
37                           VTKGridFunctionAdapter<DGF>(dgf,"q1"));
38   vtkwriter.write("q1constrainedinterpolate",Dune::VTKOptions::ascii);
39 }
```

Figure 9 shows the result obtained after setting the nonconstrained degrees of freedom to zero.

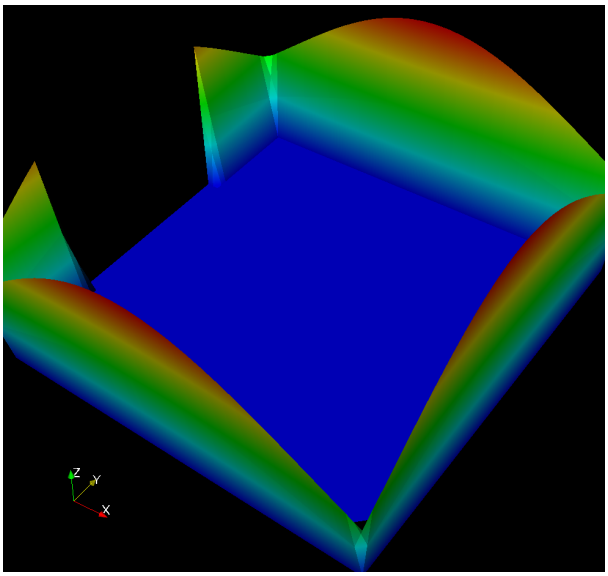


Figure 9: Interpolation of $\exp(-3\|x-c\|^2)$ with Q_1 elements with subsequent modification of nonconstrained degrees of freedom.

9 Algebraic Formulation

9.1 Solution of the Unconstrained Problem

Unconstrained Problem in Original Basis We recall the unconstrained problem in weighted residual form:

$$u_h \in U_h \ : \qquad r_h(u_h, v) = 0 \qquad \forall v \in V_h. \tag{19}$$

Solving it in the original basis reduces to the solution of a nonlinear algebraic problem:

$$\begin{aligned} \mathbf{u} \in \mathbf{U} : \qquad & r_h\left(\mathrm{FE}_{\Phi_{U_h}}(\mathbf{u}), \psi_i\right) = 0, \quad i \in \mathcal{I}_{V_h} \\ \Leftrightarrow\ & \mathcal{R}(\mathbf{u}) = \mathbf{0} \end{aligned} \tag{20}$$

where we introduced the nonlinear residual map $\mathcal{R} : \mathbf{U} = \mathbb{K}^{\mathcal{I}_{V_h}} \to \mathbb{K}^{\mathcal{I}_{V_h}}$ which is defined as

$$\left(\mathcal{R}(\mathbf{u})\right)_i = r_h(\mathrm{FE}_{U_h}(\mathbf{u}), \psi_i). \tag{21}$$

$\Phi_{V_h} = \{\psi_i \,|\, i \in \mathcal{I}_{V_h}\}$ is the basis of V_h.

Unconstrained Problem in Transformed Basis We may also solve the unconstrained problem in the transformed basis for trial and test space:

$$\begin{aligned} \mathbf{u}' \in \mathbf{U}' : \qquad & r_h\left(\mathrm{FE}_{\Phi'_{U_h}}(\mathbf{u}'), \psi'_i\right) = 0, \quad i \in \mathcal{I}_{V_h} \\ \Leftrightarrow\ & r_h\left(\mathrm{FE}_{\Phi_{U_h}}(\mathbf{T}_{U_h}^T \mathbf{u}'), \sum_{j \in \mathcal{I}_{V_h}} (\mathbf{T}_{V_h})_{i,j}\, \psi_j\right) = 0, \quad i \in \mathcal{I}_{V_h} \\ \Leftrightarrow\ & \sum_{j \in \mathcal{I}_{V_h}} (\mathbf{T}_{V_h})_{i,j}\, r_h\left(\mathrm{FE}_{\Phi_{U_h}}(\mathbf{T}_{U_h}^T \mathbf{u}'), \psi_j\right) = 0, \quad i \in \mathcal{I}_{V_h} \\ \Leftrightarrow\ & \mathbf{T}_{V_h} \mathcal{R}\left(\mathbf{T}_{U_h}^T \mathbf{u}'\right) = \mathbf{0}. \end{aligned} \tag{22}$$

Used linearity of residual form with respect to the second argument.

Requires simple matrix multiplication.

Newton solver Use Newton's method to solve the algebraic problem.

Let a current iterate $\mathbf{u}'_k$ be given.

We seek an update $\mathbf{z}'_k$ such that $\mathbf{u}'_{k+1} = \mathbf{u}'_k + \mathbf{z}'_k$ and linearize:

$$\mathbf{T}_{V_h} \mathcal{R}\left(\mathbf{T}_{U_h}^T \mathbf{u}'_{k+1}\right) \approx \mathbf{T}_{V_h} \mathcal{R}\left(\mathbf{T}_{U_h}^T \mathbf{u}'_k\right) + \mathbf{T}_{V_h} \nabla \mathcal{R}\left(\mathbf{T}_{U_h}^T \mathbf{u}'_k\right) \mathbf{T}_{U_h}^T \mathbf{z}'_k = \mathbf{0}.$$

A linear system for the update is

$$\mathbf{T}_{V_h} \nabla \mathcal{R}\left(\mathbf{T}_{U_h}^T \mathbf{u}'_k\right) \mathbf{T}_{U_h}^T \mathbf{z}'_k = -\mathbf{T}_{V_h} \mathcal{R}\left(\mathbf{T}_{U_h}^T \mathbf{u}'_k\right). \tag{23}$$

$\nabla\mathcal{R}(\mathbf{u}_k)$ denotes the Jacobian matrix of the map $\mathcal{R}$.

Multiplying the update equation with $\mathbf{T}_{U_h}^T$ from the left yields

$$\mathbf{T}_{U_h}^T \mathbf{u}'_{k+1} = \mathbf{T}_{U_h}^T \mathbf{u}'_k + \mathbf{T}_{U_h}^T \mathbf{z}'_k. \tag{24}$$

Setting $\mathbf{u}_k := \mathbf{T}_{U_h}^T \mathbf{u}'_k$ allows us now to write the Newton scheme with respect to the original basis.

Algorithm 48 (Newton's method for unconstrained problem)**.** Given the initial guess $\mathbf{u}_0$ iterate until convergence

1. Compute residual: $\mathbf{r}_k = \mathcal{R}(\mathbf{u}_k)$.
2. Transform residual: $\mathbf{r}'_k = \mathbf{T}_{V_h} \mathbf{r}_k$.
3. Solve update equation: $\mathbf{T}_{V_h} \nabla\mathcal{R}(\mathbf{u}_k) \mathbf{T}_{U_h}^T \mathbf{z}'_k = \mathbf{r}'_k$.
4. Transform update: $\mathbf{z}_k = \mathbf{T}_{U_h}^T \mathbf{z}'_k$.
5. Update: $\mathbf{u}_{k+1} = \mathbf{u}_k - \mathbf{z}_k$. □

Two applications of the basis transformation, for the residual and the update, are necessary in steps ii) and iv).

These transformations are cheap due to the structure of the transformation.

In step (iii) the *transformed* Jacobian system is required.

All these transformations are done generically by PDELab!

9.2 Solution of Constrained Problem

We now turn to the constrained problem.

Reformulation in unconstrained space We recall the constrained problem in weighted residual form:

$$u_h \in w_h + \tilde{U}_h \ : \qquad r_h(u_h, v) = 0 \quad \forall v \in \tilde{V}_h. \tag{25}$$

This problem can be reformulated in the unconstrained space by adding a constrained equation:

Proposition 49. Let $P_h : U_h \to \tilde{U}_h$ be a projection (i. e. $P_h^2 = P_h$) and assume that the affine shift is such that $P_h w_h = 0$. Then

$$u_h \in U_h \ : \qquad \begin{cases} r_h(u_h, v) = 0 \quad \forall v \in \tilde{V}_h \\ (I - P_h) u_h = w_h \end{cases} \tag{26}$$

is equivalent to (25).

Proof. Assume that (25) holds and $P_h w_h = 0$. Since u_h solves (25) the first equation in (26) clearly holds. Moreover, we have $u_h = w_h + \tilde{u}_h$ with $\tilde{u}_h \in \tilde{U}_h$ which allows us to write $u_h = w_h + P_h v_h$ for some $v_h \in U_h$. When we can prove that $v_h = u_h$ we obtain the desired

$(I - P_h)u_h = w_h$. We now show that $v_h = u_h$: Applying P_h to both sides of the identity $u_h = w_h + P_h v_h$ yields $P_h u_h = P_h w_h + P_h^2 v_h$. Using $P_h w_h = 0$ and $P_h^2 = P_h$ yields $P_h u_h = P_h v_h$. Thus we may identify v_h and u_h as v_h was arbitrary. Assume now that (26) holds. The first equation of (26) is the same as (25). From the second equation we conclude $u_h = w_h + P_h u_h$, i. e. $u_h \in w_h + \tilde{U}_h$.

□

Reformulated Problem in Coefficient Space We now seek to solve problem (26) in coefficient space.

The projection P_h is taken from the follwing commutative diagram:

$$\begin{array}{ccc} U_h & \xrightarrow{P_h = \mathrm{FE}_{\Phi'_{U_h}} \mathbf{R}^T_{\tilde{\mathbf{U}}',\mathbf{U}'} \mathbf{R}_{\tilde{\mathbf{U}}',\mathbf{U}'} \mathrm{FE}^{-1}_{\Phi'_{U_h}}} & \tilde{U}_h \\ \mathrm{FE}_{\Phi'_{U_h}} \uparrow & & \uparrow \mathrm{FE}_{\Phi'_{U_h}} \mathbf{R}^T_{\tilde{\mathbf{U}}'\mathbf{U}'} \\ \mathbf{U}' & \xrightarrow{\mathbf{R}_{\tilde{\mathbf{U}}',\mathbf{U}'}} & \tilde{\mathbf{U}}' \end{array}$$

Proposition 50. Using this definition of P_h the reformulated constrained problem (26) in coefficient space reads

$$\mathbf{u}' \in \mathbf{U}' : \qquad \left\{ \begin{array}{rcl} \mathbf{S}_{\tilde{\mathbf{V}}'} \mathcal{R}\left(\mathbf{T}^T_{U_h} \mathbf{u}'\right) & = & \mathbf{0} \\ \mathbf{R}_{\bar{\mathbf{U}}',\mathbf{U}'} \mathbf{u}' & = & \mathbf{w}' \end{array} \right. \tag{27}$$

with $\mathbf{S}_{\tilde{\mathbf{V}}'} = \mathbf{R}_{\tilde{\mathbf{V}}',\mathbf{V}'} + \mathbf{T}_{\tilde{V}_h,\bar{V}_h} \mathbf{R}_{\bar{\mathbf{V}}',\mathbf{V}'}$ and $w_h = FE_{\Phi'_{U_h}}(\mathbf{R}^T_{\bar{\mathbf{U}}',\mathbf{U}'} \mathbf{w}')$. □

The idea in this formulation is that with respect to the transformed basis the affine shift (for Dirichlet boundary conditions) can be "encoded" in the constrained degrees of freedom $\mathbf{R}_{\bar{\mathbf{U}}',\mathbf{U}'} \mathbf{u}'$. This is possible because the subspace $\tilde{U}_h$ is the image of the unconstrained degrees of freedom $\mathbf{R}_{\tilde{\mathbf{U}}',\mathbf{U}'} \mathbf{u}'$ and the decomposition is orthogonal (i. e. $\mathbf{R}^T_{\bar{\mathbf{U}}',\mathbf{U}'} \mathbf{R}_{\bar{\mathbf{U}}',\mathbf{U}'}$ and $\mathbf{R}^T_{\tilde{\mathbf{U}}',\mathbf{U}'} \mathbf{R}_{\tilde{\mathbf{U}}',\mathbf{U}'}$ are orthogonal projections).

The second equation is seen as follows:

$$\begin{array}{ll} & (I - P_h)u_h = w_h \\ \Leftrightarrow & \left(\mathrm{FE}_{\Phi'_{U_h}} \mathrm{FE}^{-1}_{\Phi'_{U_h}} - \mathrm{FE}_{\Phi'_{U_h}} \mathbf{R}^T_{\tilde{\mathbf{U}}',\mathbf{U}'} \mathbf{R}_{\tilde{\mathbf{U}}',\mathbf{U}'} \mathrm{FE}^{-1}_{\Phi'_{U_h}} \right) \mathrm{FE}_{\Phi'_{U_h}} \mathbf{u}' = \mathrm{FE}_{\Phi'_{U_h}} \mathbf{R}^T_{\bar{\mathbf{U}}',\mathbf{U}'} \mathbf{w}' \\ \Leftrightarrow & \left(\mathbf{I} - \mathbf{R}^T_{\tilde{\mathbf{U}}',\mathbf{U}'} \mathbf{R}_{\tilde{\mathbf{U}}',\mathbf{U}'} \right) \mathbf{u}' = \mathbf{R}^T_{\bar{\mathbf{U}}',\mathbf{U}'} \mathbf{w}' \\ \Leftrightarrow & \mathbf{R}^T_{\bar{\mathbf{U}}',\mathbf{U}'} \mathbf{R}_{\bar{\mathbf{U}}',\mathbf{U}'} \mathbf{u}' = \mathbf{R}^T_{\bar{\mathbf{U}}',\mathbf{U}'} \mathbf{w}' \\ \Leftrightarrow & \mathbf{R}_{\bar{\mathbf{U}}',\mathbf{U}'} \mathbf{u}' = \mathbf{w}'. \end{array} \tag{28}$$

For the first equation in (26) we proceed as in (22)

$$\begin{aligned}
\mathbf{u}' \in \mathbf{U}' : \qquad & r_h\left(\mathrm{FE}_{\Phi'_{U_h}}(\mathbf{u}'), \psi'_i\right) = 0, \quad i \in \tilde{\mathcal{I}}_{V_h} \\
\Leftrightarrow\ & r_h\left(\mathrm{FE}_{\Phi_{U_h}}(\mathbf{T}_{U_h}^T \mathbf{u}'), \sum_{j \in \mathcal{I}_{V_h}} (\mathbf{T}_{V_h})_{i,j}\, \psi_j\right) = 0, \quad i \in \tilde{\mathcal{I}}_{V_h} \\
\Leftrightarrow\ & \sum_{j \in \mathcal{I}_{V_h}} (\mathbf{T}_{V_h})_{i,j}\, r_h\left(\mathrm{FE}_{\Phi_{U_h}}(\mathbf{T}_{U_h}^T \mathbf{u}'), \psi_j\right) = 0, \quad i \in \tilde{\mathcal{I}}_{V_h} \\
\Leftrightarrow\ & \underbrace{\left(\mathbf{R}_{\tilde{\mathbf{V}}',\mathbf{V}'} + \mathbf{T}_{\tilde{V}_h,\bar{V}_h} \mathbf{R}_{\bar{\mathbf{V}}',\mathbf{V}'}\right)}_{\mathbf{S}_{\tilde{\mathbf{V}}'}} \mathcal{R}\left(\mathbf{T}_{U_h}^T \mathbf{u}'\right) = \mathbf{S}_{\tilde{\mathbf{V}}'} \mathcal{R}\left(\mathbf{T}_{U_h}^T \mathbf{u}'\right) = \mathbf{0}.
\end{aligned} \tag{29}$$

Here we made use of the structure of the transformation (18) in the final line.

Newton's Method for Constrained Problem Newton's method applied to the constrained problem (27) is formulated in the following alorithm.

Algorithm 51 (Newton's method for constrained problem)**.** Let the initial guess $\mathbf{u}_0$ with $\mathrm{FE}_{\Phi_{U_h}}(\mathbf{u}_0) \in w_h + \tilde{U}_h$ be given. Iterate until convergence

1. Compute residual: $\mathbf{r}_k = \mathcal{R}(\mathbf{u}_k)$.
2. Transform residual: $\mathbf{r}'_k = \mathbf{S}_{\tilde{\mathbf{V}}'} \mathbf{r}_k$.
3. Solve update equation:

$$\begin{pmatrix} \mathbf{S}_{\tilde{\mathbf{V}}'} \nabla \mathcal{R}\left(\mathbf{T}_{U_h}^T \mathbf{u}'_k\right) \mathbf{S}_{\tilde{\mathbf{U}}'}^T & \mathbf{0} \\ \mathbf{0} & \mathbf{I} \end{pmatrix} \begin{pmatrix} \tilde{\mathbf{z}}'_k \\ \bar{\mathbf{z}}'_k \end{pmatrix} = \begin{pmatrix} \mathbf{r}'_k \\ \mathbf{0} \end{pmatrix}$$

   and set $\mathbf{z}'_k = \begin{pmatrix} \tilde{\mathbf{z}}'_k \\ \bar{\mathbf{z}}'_k \end{pmatrix}$.
4. Transform update: $\mathbf{z}_k = \mathbf{T}_{U_h}^T \mathbf{z}'_k$. (This is where interpolation to hanging nodes is done).
5. Update: $\mathbf{u}_{k+1} = \mathbf{u}_k - \mathbf{z}_k$. □

Let $\mathbf{u}'_k$ be given. Seek update $\mathbf{z}'_k$ s.t. $\mathbf{u}'_{k+1} = \mathbf{u}'_k + \mathbf{z}'_k$.

Inserting $\mathbf{u}'_{k+1}$ into the second equation of (27) yields

$$\begin{aligned}
& \mathbf{R}_{\bar{\mathbf{U}}',\mathbf{U}'} \mathbf{u}'_{k+1} = \mathbf{R}_{\bar{\mathbf{U}}',\mathbf{U}'} \mathbf{u}'_k + \mathbf{R}_{\bar{\mathbf{U}}',\mathbf{U}'} \mathbf{z}'_k = \bar{\mathbf{w}}' \\
\Leftrightarrow\ & \mathbf{R}_{\bar{\mathbf{U}}',\mathbf{U}'} \mathbf{z}'_k = \bar{\mathbf{w}}' - \mathbf{R}_{\bar{\mathbf{U}}',\mathbf{U}'} \mathbf{u}'_k = \mathbf{0} \\
\Leftrightarrow\ & \bar{\mathbf{z}}'_k = \mathbf{0}
\end{aligned} \tag{30}$$

where we introduced $\mathbf{z}'_k = \mathbf{R}_{\bar{\mathbf{U}}',\mathbf{U}'}^T \bar{\mathbf{z}}'_k$ and used $\mathbf{R}_{\bar{\mathbf{U}}',\mathbf{U}'} \mathbf{R}_{\bar{\mathbf{U}}',\mathbf{U}'}^T = \mathbf{I}$. Note that the affine shift is not changed during the iteration:

$$\mathbf{R}_{\bar{\mathbf{U}}',\mathbf{U}'} \mathbf{u}'_{k+1} = \mathbf{R}_{\bar{\mathbf{U}}',\mathbf{U}'} \mathbf{u}'_k + \underbrace{\mathbf{R}_{\bar{\mathbf{U}}',\mathbf{U}'} \mathbf{z}'_k}_{=\mathbf{0}} = \mathbf{R}_{\bar{\mathbf{U}}',\mathbf{U}'} \mathbf{u}'_k. \tag{31}$$

Thus it is sufficient to satisfy the affine shift in the initial guess $\mathbf{R}_{\bar{\mathbf{U}}',\mathbf{U}'}\mathbf{u}'_0 = \bar{\mathbf{w}}'$.

Now insert $\mathbf{u}'_{k+1}$ into the first equation of (27):

$$\begin{aligned}
\mathbf{S}_{\tilde{\mathbf{V}}'}\mathcal{R}\left(\mathbf{T}_{U_h}^T\mathbf{u}'_{k+1}\right) &= \mathbf{S}_{\tilde{\mathbf{V}}'}\mathcal{R}\left(\mathbf{T}_{U_h}^T\mathbf{u}'_k + \mathbf{T}_{U_h}^T\mathbf{z}'_k\right) \\
&= \mathbf{S}_{\tilde{\mathbf{V}}'}\mathcal{R}\left(\mathbf{T}_{U_h}^T\mathbf{u}'_k + \mathbf{T}_{U_h}^T\left(\mathbf{R}_{\tilde{\mathbf{U}}',\mathbf{U}'}^T\underbrace{\mathbf{R}_{\bar{\mathbf{U}}',\mathbf{U}'}\mathbf{z}'_k}_{=\mathbf{0},\text{ cf.(30)}} + \mathbf{R}_{\tilde{\mathbf{U}}',\mathbf{U}'}^T\mathbf{R}_{\tilde{\mathbf{U}}',\mathbf{U}'}\mathbf{z}'_k\right)\right) \\
&= \mathbf{S}_{\tilde{\mathbf{V}}'}\mathcal{R}\left(\mathbf{T}_{U_h}^T\mathbf{u}'_k + \mathbf{T}_{U_h}^T\mathbf{R}_{\tilde{\mathbf{U}}',\mathbf{U}'}^T\mathbf{R}_{\tilde{\mathbf{U}}',\mathbf{U}'}\mathbf{z}'_k\right) \\
&= \mathbf{S}_{\tilde{\mathbf{V}}'}\mathcal{R}\left(\mathbf{T}_{U_h}^T\mathbf{u}'_k + \underbrace{\left(\mathbf{R}_{\tilde{\mathbf{U}}',\mathbf{U}'}^T + \mathbf{R}_{\bar{\mathbf{U}}',\mathbf{U}'}^T\mathbf{T}_{\tilde{U}_h,\bar{U}_h}^T\right)}_{=:\,\mathbf{S}_{\tilde{\mathbf{U}}'}^T}\mathbf{R}_{\tilde{\mathbf{U}}',\mathbf{U}'}\mathbf{z}'_k\right) \\
&= \mathbf{S}_{\tilde{\mathbf{V}}'}\mathcal{R}\left(\mathbf{T}_{U_h}^T\mathbf{u}'_k + \mathbf{S}_{\tilde{\mathbf{U}}'}^T\mathbf{R}_{\tilde{\mathbf{U}}',\mathbf{U}'}\mathbf{R}_{\tilde{\mathbf{U}}',\mathbf{U}'}^T\tilde{\mathbf{z}}'_k\right) = \mathbf{S}_{\tilde{\mathbf{V}}'}\mathcal{R}\left(\mathbf{T}_{U_h}^T\mathbf{u}'_k + \mathbf{S}_{\tilde{\mathbf{U}}'}^T\tilde{\mathbf{z}}'_k\right)
\end{aligned} \tag{32}$$

where we introduced $\mathbf{z}'_k = \mathbf{R}_{\tilde{\mathbf{U}}',\mathbf{U}'}^T\tilde{\mathbf{z}}'_k$ and used $\mathbf{R}_{\tilde{\mathbf{U}}',\mathbf{U}'}\mathbf{R}_{\tilde{\mathbf{U}}',\mathbf{U}'}^T = \mathbf{I}$.

Linearization now gives

$$\mathbf{S}_{\tilde{\mathbf{V}}'}\mathcal{R}\left(\mathbf{T}_{U_h}^T\mathbf{u}'_k + \mathbf{S}_{\tilde{\mathbf{U}}'}^T\tilde{\mathbf{z}}'_k\right) \approx \mathbf{S}_{\tilde{\mathbf{V}}'}\mathcal{R}\left(\mathbf{T}_{U_h}^T\mathbf{u}'_k\right) + \mathbf{S}_{\tilde{\mathbf{V}}'}\nabla\mathcal{R}\left(\mathbf{T}_{U_h}^T\mathbf{u}'_k\right)\mathbf{S}_{\tilde{\mathbf{U}}'}^T\tilde{\mathbf{z}}'_k = \mathbf{0}.$$

Thus the equation for the update reads

$$\mathbf{S}_{\tilde{\mathbf{V}}'}\nabla\mathcal{R}\left(\mathbf{T}_{U_h}^T\mathbf{u}'_k\right)\mathbf{S}_{\tilde{\mathbf{U}}'}^T\tilde{\mathbf{z}}'_k = -\mathbf{S}_{\tilde{\mathbf{V}}'}\mathcal{R}\left(\mathbf{T}_{U_h}^T\mathbf{u}'_k\right).$$

10 List of Main Programs Provided

This list may not be complete.

`dg.cc` Some variants of the discontinuous Galerkin method for the diffusion equation.

`dnapl.cc` Two-phase flow in porous media with cell-centered finite volumes including transport of disolved components.

`dnaplfv.cc` Two-phase flow in porous media with cell-centered finite volumes.

`dnaplmfd.cc` Two-phase flow in porous media with mimetic finite difference method.

`heleshaw.cc` Water-gas example for two-phase flow. With transport of one component in the water phase.

`heleshawfv.cc` Water-gas example for two-phase flow.

`heleshawreaction.cc` Water-gas example for two-phase flow. With transport of one component in each phase.

`laplacedirichletccfv.cc` Cell-centered finite volume method on axi-parallel structured grids for the Laplace equation.

`laplacedirichletmain.cc` Solve the Dirichlet problem for the Laplace equation with many different function spaces on different grids.

`instationarytest.cc` Solve instationary convection-diffusion equation.

`nonlineardiffusion.cc` Solve stationary nonlinear convection-diffusion equation. Sequential, parallel overlapping and parallel nonoverlapping driver.

`poisson.cc` Solve the Poisson equation.

`q1gridfunctionspacemain.cc` Set up a Q_1 function space and display a global basis function.

`q1interpolatemain.cc` Interpolate a Q_1 function from a given analytic function.

`q1constrainedinterpolatemain.cc` Set up constraints for a function space and manipulate the constrained degrees of freedom.

`q1interpolationerrormain.cc` Compute the interpolation error in a Q_1 function.

`q2interpolationerrormain.cc` Compute the interpolation error in a Q_2 function.

`reentrantcorner.cc` Solve the reentrant corner problem.

`rt0main.cc` Solve the Poisson equation using lowest order Raviart-Thomas elements.

`thinterpolatemain.cc` Show the construction of composite function spaces taking the Taylor-Hood element Q_2/Q_1 as an example.

`transporttest.cc` Solve instationary convection-diffusion equation with cell-centered finite volumes.

References

[Ale77] R. Alexander. Diagonally implicit Runge-Kutta methods for stiff O. D. E.'s. *SIAM Journal on Numerical Analysis*, 14(6):1006–1021, 1977.

[BBD+08a] P. Bastian, M. Blatt, A. Dedner, C. Engwer, R. Klöfkorn, R. Kornhuber, M. Ohlberger, and O. Sander. A generic grid interface for parallel and adaptive scientific computing. part I: implementation and tests in DUNE. *Computing*, 82(2-3):121–138, 2008. DOI 10.1007/s00607-008-0003-x.

[BBD+08b] P. Bastian, M. Blatt, A. Dedner, C. Engwer, R. Klöfkorn, M. Ohlberger, and O. Sander. A generic grid interface for parallel and adaptive scientific computing. part I: abstract framework. *Computing*, 82(2-3):103–119, 2008. DOI 10.1007/s00607-008-0003-x.

[BF91] F. Brezzi and M. Fortin. *Mixed and hybrid finite element methods*. Springer-Verlag New York, 1991.

[Cia02] P. G. Ciarlet. *The finite element method for elliptic problems*. Classics in applied mathematics. SIAM, 2002.

[OBB98] J. T. Oden, I. Babuška, and C. E. Baumann. A discontinuous *hp* finite element method for diffusion problems. *Journal of Computational Physics*, 146:491–519, 1998.

[Shu88] C. W. Shu. Total-variation-diminishing time discretizations. *SIAM J. Sci. Stat. Comput.*, 9:1073, 1988.