DUNE PDELab Tutorial 02
Cell-Centered Finite Volume Method

DUNE/PDELab Team

August 31, 2021

Contents

1 Introduction **2**

2 PDE Problem **2**

3 Cell-centered Finite Volume Method **2**

4 Realization in PDELab **5**
4.1 Ini-File 5
4.2 Function `main` 6
4.3 Function `driver` 6
4.4 The `Problem` Class 7
4.5 Local Operator `NonlinearPoissonFV` 7
4.6 Running the Example 15

5 Outlook **16**

1 Introduction

This tutorial solves the same partial differential equation (PDE) as tutorial 01, namely a nonlinear Poisson equation, with the following differences:

1) Implements a cell-centered finite volume method with two-point flux approximation as an example of a non-conforming scheme.

2) Implements *all* possible methods of a local operator.

Depends On

Tutorial 00 and 01.

2 PDE Problem

Consider the following nonlinear Poisson equation (the same as in tutorial 01) with Dirichlet and Neumann boundary conditions:

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

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

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

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

3 Cell-centered Finite Volume Method

The application of the cell-centered finite volume method as presented here is restricted to *axiparallel meshes*. We assume that the domain Ω is covered by a mesh $\mathcal{T}_h = \{T_1, \dots, T_M\}$ consisting of elements which are closed sets satisfying

$$\bigcup_{T \in \mathcal{T}_h} T = \overline{\Omega}, \quad \forall T, T' \in \mathcal{T}_h, T \neq T' : \mathring{T} \cap \mathring{T}' = \emptyset. \tag{2}$$

In order to describe the method some further notation is needed. The nonempty intersections $F = T_F^- \cap T_F^+$ of codimension 1 form the interior skeleton $\mathcal{F}_h^i = \{F_1, \dots, F_N\}$. Each intersection is equipped with a unit normal vector ν_F pointing from T_F^- to T_F^+. The intersections of an element $F = T_F^- \cap \partial\Omega$ with the domain boundary form the set of boundary intersections $\mathcal{F}_h^{\partial\Omega} = \{F_1, \dots, F_L\}$ which can be further partitioned into Dirichlet boundary intersections $\mathcal{F}_h^{\Gamma_D}$ and Neumann boundary intersections $\mathcal{F}_h^{\Gamma_N}$. Each boundary intersection is equipped with a unit normal vector ν_F which coincides with the unit outer normal to the domain. Furthermore, x_T, x_F denotes the center point of an element or face. This notation is illustrated graphically in Figure 1.

For the cell-centered finite volume method the discrete function space involved is the space of piecewise constant functions on the mesh:

$$W_h = \{w \in L^2(\Omega) \,:\, w|_T = \text{const for all } T \in \mathcal{T}_h\}.$$

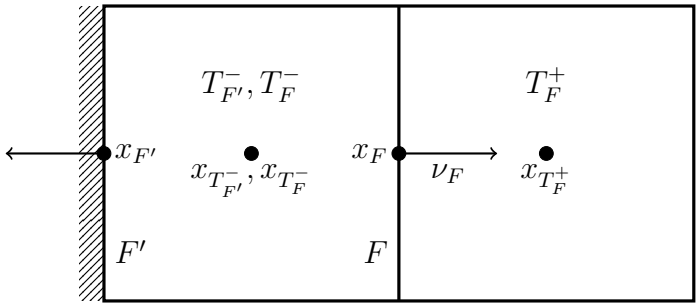

Figure 1: Illustration of quantities associated with elements and intersections.

In order to derive the residual form we proceed as follows: multiply equation (1) with a test function $v \in W_h$, i.e. *from the discrete space*, and use integration by parts:

$$\begin{aligned}
\int_\Omega f v \, dx &= \int_\Omega [-\Delta u + q(u)] v \, dx \\
&= \sum_{T \in \mathcal{T}_h} v \int_T -\Delta u + q(u) \, dx && (v \text{ const on } T) \\
&= \sum_{T \in \mathcal{T}_h} \left[\int_T q(u) v \, dx - \int_{\partial T} \nabla u \cdot \nu v \, ds \right] && (\text{Gauss' thm.}) \\
&= \sum_{T \in \mathcal{T}_h} \int_T q(u) v \, dx - \sum_{F \in \mathcal{F}_h^i} \int_F \nabla u \cdot \nu_F \big[v(x_{T_F^-}) - v(x_{T_F^+}) \big] \, ds \\
&\quad - \sum_{F \in \mathcal{F}_h^{\partial\Omega}} \int_F \nabla u \cdot \nu_F \, ds. && (\text{rearrange})
\end{aligned}$$

At this point, the normal derivative $\partial_{\nu_F} u = \nabla u \cdot \nu_F$ is approximated by a difference quotient

$$\nabla u \cdot \nu_F = \frac{u_h(x_{T_F^+}) - u_h(x_{T_F^-})}{\| x_{T_F^+} - x_{T_F^-} \|} + \text{ error}$$

and all integrals are approximated by the midpoint rule

$$\int_T f \, dx = f(x_T)|T| + \text{ error}$$

where $|T|$ is the measure of T.

Put together the cell-centered finite volume method can be stated in its abstract form suitable for implementation in PDELab:

$$\boxed{\text{Find } u_h \in W_h \text{ s.t.:} \quad r_h^{\text{CCFV}}(u_h, v) = 0 \quad \forall v \in W_h} \tag{3}$$

where the residual form is

$$
\begin{aligned}
r_h^{\mathrm{CCFV}}(u_h, v) = &\sum_{T\in\mathcal{T}_h} q(u_h(x_T))v(x_T)|T| - \sum_{T\in\mathcal{T}_h} f(x_T)v(x_T)|T| \\
&- \sum_{F\in\mathcal{F}_h^i} \frac{u_h(x_{T_F^+}) - u_h(x_{T_F^-})}{\|x_{T_F^+} - x_{T_F^-}\|}\big[v(x_{T_F^-}) - v(x_{T_F^+})\big]|F| \\
&+ \sum_{F\in\mathcal{F}_h^{\partial\Omega}\cap\Gamma_D} \frac{u_h(x_{T_F^-})}{\|x_F - x_{T_F^-}\|} v(x_{T_F^-})|F| \\
&- \sum_{F\in\mathcal{F}_h^{\partial\Omega}\cap\Gamma_D} \frac{g(x_F)}{\|x_F - x_{T_F^-}\|} v(x_{T_F^-})|F| + \sum_{F\in\mathcal{F}_h^{\partial\Omega}\cap\Gamma_N} j(x_F)v(x_{T_F^-})|F|.
\end{aligned}
\tag{4}
$$

In this case *five* different types of integrals are involved in the residual form:

1. Volume integral depending on trial and test function.
2. Volume integral depending on test function only.
3. Interior intersection integral depending on trial and test function.
4. Boundary intersection integral depending on trial and test function.
5. Boundary intersection integral depending on test function only.

Also note that no constraints on the function space are necessary in this case. Dirichlet as well as Neumann boundary conditions are built weakly into the residual form!

Finally, many types of discontinuous Galerkin finite element methods (DGFEM) lead to the same five types of integrals and can be applied on general unstructured conforming as well as nonconforming meshes.

General Residual Form

The residual form of the cell-centered finite volume method suggests that all residual forms could be composed of five different types of terms in the following way:

$$
\begin{aligned}
r(u, v) = &\sum_{T\in\mathcal{T}_h} \alpha_T^V(R_T u, R_T v) + \sum_{T\in\mathcal{T}_h} \lambda_T^V(R_T v) \\
&+ \sum_{F\in\mathcal{F}_h^i} \alpha_F^S(R_{T_F^-}u, R_{T_F^+}u, R_{T_F^-}v, R_{T_F^+}v) \\
&+ \sum_{F\in\mathcal{F}_h^{\partial\Omega}} \alpha_F^B(R_{T_F^-}u, R_{T_F^-}v) + \sum_{F\in\mathcal{F}_h^{\partial\Omega}} \lambda_F^B(R_{T_F^-}v).
\end{aligned}
\tag{5}
$$

Here, we define the restriction of a function $u \in U$ to an element by

$$(R_T u)(x) = u(x) \quad \forall x \in \mathring{T}.$$

Note that the restriction of a function to element T is only defined in the interior of T. On interior intersections F, functions may be two-valued and limits from

within the elements T_F^-, T_F^+ need to be defined (when U is the space of element-wise constants that is trivial).

The five terms comprise volume integrals (superscript V), interior skeleton integrals (superscript S) and boundary integrals (superscript B). Furthermore, the α-terms depend on trial and test functions whereas the λ-terms only depend on the test function and involve the data of the PDE.

Each of the five terms α_T^V, α_F^S, α_F^B, λ_T^V, λ_F^B corresponds to one method on the local operator. In addition to the evaluation of residuals also Jacobians and matrix-free application of Jacobians are needed. This gives rise to in total $5 + 3 + 3 = 11$ possible methods on a local operator given in the following table:

	volume	skeleton	boundary
residual	`alpha_volume` `lambda_volume`	`alpha_skeleton`	`alpha_boundary` `lambda_boundary`
Jacobian	`jacobian_volume`	`jacobian_skeleton`	`jacobian_boundary`
Jac. app.	`jacobian_apply_volume`	`jacobian_apply_skeleton`	`jacobian_apply_boundary`

4 Realization in PDELab

The structure of the code is already known from the previous tutorials. It consists of the following files:

1) The ini-file `tutorial02.ini` holds parameters read by various parts of the code which control the execution.

2) The main file `tutorial02.cc` includes the necessary C++, DUNE and PDELab header files and contains the `main` function where the execution starts. The purpose of the `main` function is to instantiate DUNE grid objects and call the `driver` function.

3) File `driver.hh` instantiates the necessary PDELab classes for solving a nonlinear stationary problem with the cell-centered finite volume method and solves the problem.

4) File `nonlinearpoissonfv.hh` contains the class `NonlinearPoissonFV` realizing a PDELab local operator implementing the cell-centered finite volume method on axi-parallel meshes.

5) File `problem.hh` contains a parameter class which encapsulates the user-definable part of the PDE problem as introduced in tutorial 01.

4.1 Ini-File

The ini-file uses the same sections as in tutorial 01 with the following exceptions:

- Only the structured grid in its `YaspGrid` implementation can be used in dimension 2 and 3. `OneDGrid` is used in dimension 1.
- No degree can be chosen.

4.2 Function `main`

The function `main` is very similar to the one in tutorials 00 and 01 and need not be repeated here.

4.3 Function `driver`

Also the function `driver` is very similar in structure to the one in tutorial 00 and 01. Here we just point out the differences. The cell-centered finite volume method is based on the space of piecewise constant functions on the mesh W_h. The following code segment constructs this function space using the class `P0LocalFiniteElementMap`:

```
// Make grid function space
typedef Dune::PDELab::P0LocalFiniteElementMap<DF,RF,dim> FEM;
FEM fem(Dune::GeometryTypes::cube(dim));
typedef Dune::PDELab::NoConstraints CON;
typedef Dune::PDELab::ISTL::VectorBackend<> VBE;
typedef Dune::PDELab::GridFunctionSpace<GV,FEM,CON,VBE> GFS;
GFS gfs(gv,fem);
gfs.name("Q0");
```

The constraints class `NoConstraints` is used to express that there are no constraints on the function space.

Now no constraints container type is exported by the grid function space. Instead the class `EmptyTransformation` is used in the grid operator:

```
// Make a global operator
typedef Dune::PDELab::ISTL::BCRSMatrixBackend<> MBE;
MBE mbe(2*dim+1); // guess nonzeros per row
typedef Dune::PDELab::EmptyTransformation CC;
typedef Dune::PDELab::GridOperator<
  GFS,GFS,  /* ansatz and test space */
  LOP,      /* local operator */
  MBE,      /* matrix backend */
  RF,RF,RF, /* domain, range, jacobian field type*/
  CC,CC     /* constraints for ansatz and test space */
  > GO;
GO go(gfs,gfs,lop,mbe);
```

Cell-wise data is passed to the `VTKWriter` using its method `addCellData`:

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

These are the only changes to the driver!

4.4 The `Problem` Class

The class `NonlinearPoissonFV` explained below uses the same problem class as the class `NonlinearPoissonFEM`. This means that the same problem can be easily solved using the two different methods.

4.5 Local Operator `NonlinearPoissonFV`

The class `NonlinearPoissonFV` implements the element-wise computations of the cell-centered finite volume method. In particular, it provides a full implementation of all possible methods on a local operator including analytic Jacobians. The class has the problem class as a template parameter:

```
template<typename Param>
class NonlinearPoissonFV :
  public Dune::PDELab::FullVolumePattern,
  public Dune::PDELab::FullSkeletonPattern,
  public Dune::PDELab::LocalOperatorDefaultFlags
```

The base class `FullSkeletonPattern` provides the local operator with a method coupling all degrees of freedom of two elements sharing an intersection. In combination with `FullVolumePattern` this provides the sparsity pattern of the matrix.

The only private data member is a reference to an object to the parameter class:

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

The public section begins with a definition of flags controlling assembly of the sparsity pattern

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

as well as element contributions:

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

These five flags specify that all five contributions will be provided.

The constructor just gets a reference of the parameter object:

```
  NonlinearPoissonFV (Param& param_)
```

Method `lambda_volume`

This method was already present in the finite element method and corresponds to sum number two on the right hand side of equation (4). The element contributions for the cell-centered finite volume method are particularly simple to implement. Here is the right hand side contribution:

```
template<typename EG, typename LFSV, typename R>
void lambda_volume (const EG& eg, const LFSV& lfsv,
                    R& r) const
{
  // center of reference element
  auto cellgeo = eg.geometry();
  auto cellcenterlocal =
    referenceElement(cellgeo).position(0,0);

  // accumulate residual
  auto f = param.f(eg.entity(),cellcenterlocal);
  r.accumulate(lfsv,0,-f*cellgeo.volume());
}
```

The variable `cellcenterlocal` is filled with the center of the reference element. Then the function f is evaluated and the integral is approximated with the midpoint rule. To that end `cellgeo.volume()` provides the measure of the element.

Note that throughout the whole class we assume that the basis functions of the space W_h are one on one element and zero on all others, i.e.

$$\phi_i(x) = \begin{cases} 1 & x \in T_i \\ 0 & \text{else} \end{cases} . \tag{6}$$

This means that basis functions will never be evaluated!

Method `lambda_boundary`

This method was also already present in the finite element method and corresponds to sums five *and* six on the right hand side of equation (4). It assembles contributions from Dirichlet and Neumann boundary conditions and has the interface

```
template<typename IG, typename LFSV, typename R>
void lambda_boundary (const IG& ig, const LFSV& lfsv_i,
                      R& r_i) const
```

First the center of the reference element of the intersection is extracted and the boundary condition type is evaluated:

```
  // face volume for integration
  auto facegeo = ig.geometry();
  auto facecenterlocal =
    referenceElement(facegeo).position(0,0);

  // evaluate boundary condition and quit on Dirichlet
  bool isdirichlet =
    param.b(ig.intersection(),facecenterlocal);
```

Now comes the part for the Dirichlet boundary conditions where we need to compute the distance from the face center to the element center, the value of the Dirichlet boundary condition and the measure of the face:

```
    if (isdirichlet)
      {
        // inside cell center
        auto insidecenterglobal=ig.inside().geometry().center();

        // face center in global coordinates
        auto facecenterglobal = facegeo.center();

        // compute distance of these two points
        insidecenterglobal -= facecenterglobal;
        auto distance = insidecenterglobal.two_norm();

        // face center in local coordinates of the element
        auto facecenterinelement=ig.geometryInInside().center();

        // evaluate Dirichlet condition
        auto g = param.g(ig.inside(),facecenterinelement);

        // face volume for integration
        auto face_volume = facegeo.volume();

        // contribution to residual
        r_i.accumulate(lfsv_i,0,-g/distance*face_volume);
      }
```

The Neumann part is much simpler:

```
    else
      {
        // contribution to residual from Neumann boundary
        auto j = param.j(ig.intersection(),facecenterlocal);
        r_i.accumulate(lfsv_i,0,j*facegeo.volume());
      }
```

Method `alpha_volume`

Now `alpha_volume` has also been present before and corresponds to the first sum on the right hand side of equation (4). Here it just contains the evaluation of the reaction term with the midpoint rule:

```
  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
  {
    // get cell value
    auto u = x(lfsu,0);

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

```
    // and accumulate
    r.accumulate(lfsv,0,q*eg.geometry().volume());
  }
```

Method `jacobian_volume`

Now we come to the first method that has not been implemented in previous examples. The method `jacobian_volume` will assemble the entries of the Jacobian coupling all degrees of the given element. As there is only one degree of freedom per element there is only one matrix entry to assemble. The matrix entries are returned in the container which is the last argument of the method:

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

First the derivative of the nonlinearity is evaluated

```
    auto u = x(lfsu,0);
    auto qprime = param.qprime(u);
```

and the matrix entry is written into the container

```
    mat.accumulate(lfsv,0,lfsu,0,qprime*eg.geometry().volume());
```

`mat.accumulate` has five arguments: the *matrix row* given by local test space and number of the test function, the *matrix column* given by local trial space and number of the trial function and, as last argument, the contribution to the matrix entry which is added to the global Jacobian matrix.

Method `jacobian_apply_volume`

This method is very similar to the previous method except that it multiplies the local Jacobian contribution immediately with a vector and accumulates the result.

The method has the following interface:

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

`x` are the coefficients of the linearization point and `z` are the entries of the vector to be multiplied with the Jacobian. The result is accumulated to the container `r`.

Here is the implementation:

```
    // evaluate derivative reaction term
    auto u = x(lfsu,0);
    auto qprime = param.qprime(u);
```

```
    // and accumulate
    r.accumulate(lfsv,0,qprime*z(lfsu,0)*eg.geometry().volume());
```

Comparison with `jacobian_volume` shows that the Jacobian entry is multiplied with the entry of `z`.

Method `alpha_skeleton`

This is the major new method needed to implement the flux terms in finite volume and discontinuous Galerkin methods. It has the following interface:

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

The arguments comprise an intersection, local trial function and local test space for both elements adjacent to the intersection and containers for the local residual contributions in both elements. The subscripts `_i` and `_o` correspond to "inside" and "outside". W.r.t. our notation above in section 3 "inside" corresponds to "-" and "outside" corresponds to "+".

Note that `alpha_skeleton` needs to assemble residual contributions for all test functions involved with *both* elements next to the intersection and corresponds to sum number three on the right hand side of equation (4).

It starts by extracting the elements adjacent to the intersection

```
    auto cell_inside = ig.inside();
    auto cell_outside = ig.outside();
```

and then extracts their geometries

```
    auto insidegeo = cell_inside.geometry();
    auto outsidegeo = cell_outside.geometry();
```

and the centers

```
    auto inside_global = insidegeo.center();
    auto outside_global = outsidegeo.center();
```

Now the distance of the centers can be computed

```
    inside_global -= outside_global;
    auto distance = inside_global.two_norm();
```

and the measure of the face is extracted

```
    auto facegeo = ig.geometry();
    auto face_volume = facegeo.volume();
```

which puts us in the position to accumulate the residual contributions

```
    auto dudn = (x_o(lfsu_o,0)-x_i(lfsu_i,0))/distance;
    r_i.accumulate(lfsv_i,0,-dudn*face_volume);
    r_o.accumulate(lfsv_o,0, dudn*face_volume);
```

In fact, the contribution to the inside element, i.e. to `r_i` is the flux from the inside to the outside element. The contribution to the outside element residual is exactly the negative value, i.e. we have local conservation.

Method `jacobian_skeleton`

In the computation of the Jacobian w.r.t. skeleton terms we can exploit the fact that the discrete residual form is actually linear in these terms as the nonlinearity is restricted to the volume term only.

An interior face contributes to four matrix parts of the global matrix as there are test functions on the inside and outside elements (corresponding to rows in the matrix) as well as trial functions on the inside and outside element (corresponding to columns of the matrix). In the case of the cell-centered finite volume method for the nonlinear Poisson equations there is only one degree of freedom and test function per element, so there are four matrix entries which the face contributes. In case of higher order discontinuous Galerkin schemes and/or systems of PDEs there are four blocks of the matrix where the face contributes to. The following figure illustrates the matrix structure and the corresponding submatrices. For ease of drawing it is assumed that all trial and test functions of one element are numbered consecutively but this need not be the case!

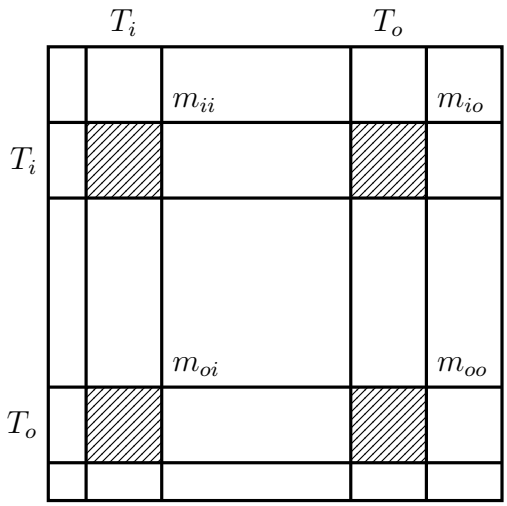

Figure 2: Matrix block contributions computed by `jacobian_skeleton`.

The method has the following interface:

```
template<typename IG, typename LFSU, typename X,
         typename LFSV, typename M>
void jacobian_skeleton (const IG& ig,
       const LFSU& lfsu_i, const X& x_i, const LFSV& lfsv_i,
       const LFSU& lfsu_o, const X& x_o, const LFSV& lfsv_o,
       M& mat_ii, M& mat_io,
       M& mat_oi, M& mat_oo) const
```

It is very similar to `alpha_skeleton` except that four containers are passed where the matrix entries of the four blocks need to be stored.

The computation of distance of cell centers and the face volume are exactly the same as in `alpha_skeleton`. Then the matrix entries are given by:

```
    // contribution to jacobian entries
    mat_ii.accumulate(lfsv_i,0,lfsv_i,0, face_volume/distance);
    mat_io.accumulate(lfsv_i,0,lfsv_o,0,-face_volume/distance);
    mat_oi.accumulate(lfsv_o,0,lfsv_i,0,-face_volume/distance);
    mat_oo.accumulate(lfsv_o,0,lfsv_o,0, face_volume/distance);
```

Method `jacobian_apply_skeleton`

The `jacobian_apply_skeleton` method needs to compute the local Jacobian contributions and multiply them with a given coefficient vector. It has the following interface

```
  template<typename IG, typename LFSU, typename X, typename LFSV,
           typename Y>
  void jacobian_apply_skeleton
  ( const IG& ig,
    const LFSU& lfsu_i, const X& x_i, const X& z_i, const LFSV& lfsv_i,
    const LFSU& lfsu_o, const X& x_o, const X& z_o, const LFSV& lfsv_o,
    Y& y_i, Y& y_o) const
```

`x_i`, `x_o` are the linearization point and `z_i`, `z_o` are the coefficients to multiply with.

As the skeleton terms are linear with respect to degrees of freedom the Jacobian does not depend on the linearization point and we may reuse the `alpha_skeleton` method here:

```
    alpha_skeleton(ig,lfsu_i,z_i,lfsv_i,lfsu_o,z_o,lfsv_o,y_i,y_o);
```

Method `alpha_boundary`

The `alpha_boundary` method is also new. It corresponds to the fourth sum on the right hand side of equation (4) which is again linear in the degrees of freedom. The interface is now:

```
  template<typename IG, typename LFSU, typename X,
           typename LFSV, typename R>
  void alpha_boundary (const IG& ig,
                       const LFSU& lfsu_i, const X& x_i,
                       const LFSV& lfsv_i, R& r_i) const
```

The residual contribution depends only on quantities on the inside element of the intersection.

First we need to check whether the face is on the Dirichlet boundary:

```
    auto facegeo = ig.geometry();
    auto facecenterlocal =
```

```
    referenceElement(facegeo).position(0,0);
  bool isdirichlet = param.b(ig.intersection(),facecenterlocal);
  if (!isdirichlet) return;
```

Then the distance from face center to cell center is computed:

```
  // inside cell center
  auto insidecenterglobal = ig.inside().geometry().center();

  // face center in global coordinates
  auto facecenterglobal = facegeo.center();

  // compute distance of these two points
  insidecenterglobal -= facecenterglobal;
  auto distance = insidecenterglobal.two_norm();
```

and the residual contribution can be accumulated:

```
  // face volume for integration
  auto face_volume = facegeo.volume();

  // contribution to residual
  r_i.accumulate(lfsv_i,0,x_i(lfsu_i,0)/distance*face_volume);
```

Method `jacobian_boundary`

The `jacobian_boundary` method computes the Jacobian contributions resulting from boundary face integrals and has the following interface:

```
template<typename IG, typename LFSU, typename X,
         typename LFSV, typename M>
void jacobian_boundary (const IG& ig,
                        const LFSU& lfsu_i, const X& x_i,
                        const LFSV& lfsv_i, M& mat_ii) const
```

The interface is the same as for `alpha_boundary` except that a matrix container is passed as the last argument.

As the contributions only depend on test and trial functions of the inside element there is only contribution to one matrix entry:

```
  mat_ii.accumulate(lfsv_i,0,lfsv_i,0,face_volume/distance);
```

Method `jacobian_apply_boundary`

Finally, the `jacobian_apply_boundary` does a matrix-free Jacobian times vector multiplication. Due to linearity we can reuse the `alpha_boundary` method:

```
template<typename IG, typename LFSU, typename X,
         typename LFSV, typename Y>
void jacobian_apply_boundary
( const IG& ig,
  const LFSU& lfsu_i, const X& x_i, const X& z_i,
```

```
    const LFSV& lfsv_i, Y& y_i) const
  {
    // reuse alpha_boundary because it is linear
    alpha_boundary(ig,lfsu_i,z_i,lfsv_i,y_i);
  }
```

4.6 Running the Example

Figure 3 shows results for three different values of η on a relatively coarse mesh. Clearly, the piecewise constant approximation can be seen.

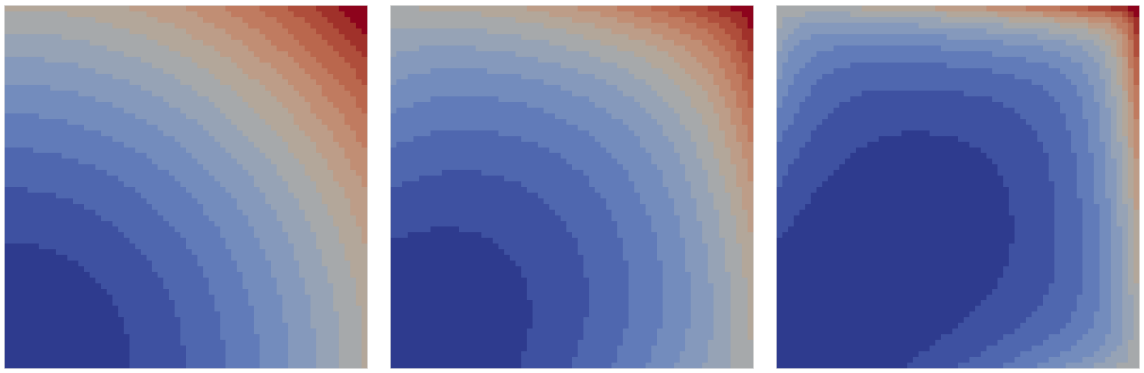

Figure 3: Illustration of the influence of the parameter η in nonlinearity on the solution. $\eta = 0$ (left), $\eta = 10$ (middle), $\eta = 100$ (right).

The table in Figure 4 shows a run-time comparison of the Q_1 conforming finite element method with numerical Jacobian from tutorial 01 with the cell-centered finite volume method implemented here on a 512^2 mesh in $2d$ and $\eta = 100$: One can see that matrix assembly is five times faster, both due to less work per element and analytic Jacobian. The time per iteration in the algebraic multigrid solver is about the same but the number of iterations is less for the FV scheme. In total the FV scheme is more than two times faster.

Figure 4: Comparison of conforming finite element and cell-centered finite volume method

	$Q1$ conforming FEM	cell-centered FV
# DOF	263169	262144
Matrix assembly time	0.87	0.20
Time per iteration	0.10	0.08
# Linear Iterations	15	9
# Newton Iterations	5	5
Total time	16.83	6.95

5 Outlook

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

- Compare the convergence of Newton's method in tutorial 01 and 02. Does the exact Jacobian in combination with a different discretization scheme make any difference?

- Implement a convection term with upwinding. The equation in strong conservative form is

$$\nabla\{\vec{\beta}u - \nabla u\} + q(u) = f$$

  where $\vec{\beta}(x)$ is a given (divergence free) velocity field.

  In the residual form the full upwind discretization is realized by the following term:

$$r_h^{conv}(u,v) = \sum_{F\in\mathcal{F}_h^i} \vec{\beta}(x_F)u_F^{\uparrow}\big[v(x_{T_F^-}) - v(x_{T_F^+})\big]|F|$$

  where the upwind value is

$$u_F^{\uparrow} = \begin{cases} u(x_F^-) & \vec{\beta}(x_F)\cdot\nu_F \geq 0 \\ u(x_F^+) & \text{else} \end{cases} .$$