**Tutorial**


U. Blunck, T. Hahn

# Boundary Value Problem for Incompressible Generalized Porous Media Equation

September 20th, 2012


Version 1.2

http://www.hiflow3.org

# Contents

# Using HiFlow$^3$ for solving the incompressible Generalized Porous Media Equation

## 1 Introduction

HiFlow$^3$ is a multi-purpose finite element software providing powerful tools for efficient and accurate solution of a wide range of problems modeled by partial differential equations (PDEs). Based on object-oriented concepts and the full capabilities of C++ the HiFlow$^3$ project follows a modular and generic approach for building efficient parallel numerical solvers. It provides highly capable modules dealing with the mesh setup, finite element spaces, degrees of freedom, linear algebra routines, numerical solvers, and output data for visualization. Parallelism - as the basis for high performance simulations on modern computing systems - is introduced on two levels: coarse-grained parallelism by means of distributed grids and distributed data structures, and fine-grained parallelism by means of platform-optimized linear algebra back-ends.

As the generalized porous media equation presented in this tutorial is closely related to the Navier-Stokes equation, the basic structure and the program have been inherited from the HiFlow$^3$ tutorial "Boundary Value Problem for Incompressible Navier-Stokes Equation" by M. Baumann, A. Helfrich-Schkarbanenko, E. Ketelaer, S. Ronnås and M. Wlotzka [BHSK$^+$].

### 1.1 How to use the Tutorial

You find the example code (`porous_media.cc`, `porous_media.h`)and a parameter file (`porous_media.xml`) in the folder `/hiflow/examples/porous_media`. The geometry data (`*.inp`, `*.vtu`) is stored in the folder `/hiflow/examples/data`.

#### 1.1.1 Using HiFlow$^3$ as a Developer

First build and compile HiFlow$^3$ . Go to the directory `/build/example/porous_media`, where the binary generalized porous media equation tutorial is stored. Type
**./porous_media** to execute the program in sequential mode. You need to make sure that the default parameter file `porous_media.xml` is stored in the same directory as the binary, and that the geometry data specified in the parameter file is stored in `/hiflow/examples/data`.

# 2 Mathematical Setup

## 2.1 Problem

We consider the simulation of stationary porous flow in a two-dimensional channel $\Omega$ with narrow in- and outflow. A rectangular obstacle is introduced at the inlet to facilitate spreading of the fluid.

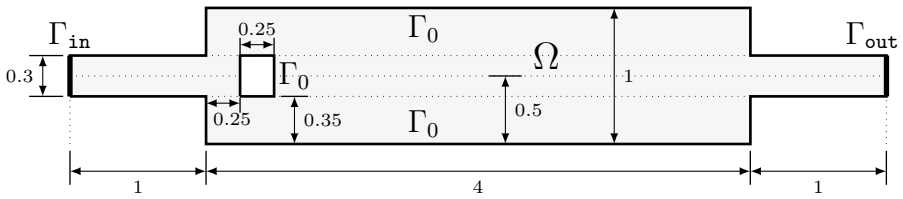



Figure 1: Non-dimensional ratio of the measurements used for the geometry.

The liquid is assumed to be an incompressible Newtonian fluid. The porous medium is assumed to be uniformly distributed with well connected vacancies for each respective subdomain. The size of the single components of the porous medium, as well as the size of the vacancies, is assumed to be significantly smaller than the modeled domain. Thus, the fluid flow within the column can be modeled using the Generalized Porous Media Equation [Blu12], see (1a).
There are three different boundary conditions to be distinguished:

The inflow condition at the inflow $\Gamma_{\text{in}}$, see (1c).
The outflow condition at the exit $\Gamma_{\text{out}}$, see (1d).
"No-Slip" condition at the remaining boundaries $\Gamma_0 (= \partial\Omega \backslash \overline{(\Gamma_{\text{in}} \cup \Gamma_{\text{out}})})$, (1e).

At the outflow, the natural Neumann- (so called "Do-Nothing"-) condition, adapted to porous flow, is applied. On both of the remaining two boundaries, Dirichlet boundary conditions are used.
In the formulation of the GPME used in this tutorial, the external body forces are neglected.
These premises lead to the following boundary value problem for the unknown velocity field $\mathbf{u} = (u_1, u_2)^\top$ and the pressure field $p$ (with each $u_1$, $u_2$ and $p$ being functions of $(x_1, x_2)^\top$):

$$\frac{1}{\varepsilon}(\mathbf{u}\cdot\nabla)\frac{\mathbf{u}}{\varepsilon} + \frac{1}{\varepsilon\varrho_f}\nabla p\,\varepsilon + \frac{\nu_e}{\varepsilon}\Delta\mathbf{u} + \frac{\nu_f}{\kappa}\mathbf{u} + \frac{1.75}{\sqrt{150}}\frac{1}{\sqrt{\kappa}}\frac{\|\mathbf{u}\|}{\varepsilon^{3/2}}\mathbf{u} = \mathbf{0} \quad \text{in } \Omega \tag{1a}$$

$$\nabla\cdot\mathbf{u} = 0 \quad \text{in } \Omega \tag{1b}$$

$$\mathbf{u} = \mathbf{g} \quad \text{on } \Gamma_{\text{in}} \tag{1c}$$

$$(-\boldsymbol{\mathcal{I}}p + \frac{\nu_e}{\varepsilon}\nabla\mathbf{u})\cdot\mathbf{n} = 0 \quad \text{on } \Gamma_{\text{out}} \tag{1d}$$

$$\mathbf{u} = \mathbf{0} \quad \text{on } \Gamma_0 \tag{1e}$$

The porosity $\varepsilon$ is defined as the ratio $\frac{V_{free}}{V_{ref}}$ of the interstitial volume $V_{free}$ to the absolute one $V_{ref}$ of a reference volume and is assumed to behave locally constant. Same holds for the permeability $\kappa$, which is specific to the utilized porous media. The density of the fluid $\varrho_f$ is constant in the domain, as only incompressible fluids are considered. The effective kinematic viscosity of the fluid within the porous medium $\nu_e$ is assumed to be equal to the kinematic viscosity of the fluid itself $\nu_f$ [GA94, p. 356], which is a fluid-specific constant. The vector $\mathbf{n} = (n_1, n_2)^\top$ (with each component dependent on $(x_1, x_2)^\top$) denotes the outer normal vector on $\partial\Omega$ and $\boldsymbol{\mathcal{I}} \in \mathbb{R}^{2\times 2}$ the unit matrix. The flow profile described by the given function $\mathbf{g} : \Gamma_{\mathtt{in}} \to \mathbb{R}^2$ is enforced at the inflow.

## 2.2 Solving the non-linear problem with Newton's method

Due to the terms accounting for the convective acceleration $\left(\frac{1}{\varepsilon}(\mathbf{u}\cdot\nabla)\frac{\mathbf{u}}{\varepsilon}\right)$ and the non-linear drag contribution $\left(\frac{1.75}{\sqrt{150}}\frac{1}{\sqrt{\kappa}}\frac{\|\mathbf{u}\|}{\varepsilon^{3/2}}\mathbf{u}\right)$, the Generalized Porous Media Equation (1a) is a non-linear equation. One possibility to solve a non-linear problem is using iterative Newtons method, which in general is a method to find the zero of a function $F$, i.e.

$$F(\boldsymbol{\xi}) = 0,$$

where $\boldsymbol{\xi} = (\mathbf{u}, p)^\top$ represents the solution. The corresponding Newton-Iteration has the form:

$$\boldsymbol{\xi}^{k+1} := \boldsymbol{\xi}^k - \underbrace{(\nabla F(\boldsymbol{\xi}^k))^{-1} F(\boldsymbol{\xi}^k)}_{\mathbf{c}^k}, \qquad k = 0, 1, 2, \ldots$$

Or, avoiding the necessity of deriving of the inverse of $\nabla F(\boldsymbol{\xi}^k)$, the iteration can be written as:

$$\begin{aligned} \nabla F(\boldsymbol{\xi}^k)\,\mathbf{c}^k &= F(\boldsymbol{\xi}^k) \\ \boldsymbol{\xi}^{k+1} &= \boldsymbol{\xi}^k - \mathbf{c}^k \end{aligned} \qquad k = 0, 1, 2, \ldots \tag{2}$$

The vector $\mathbf{c}^k = (\mathbf{c}_u^k, c_p^k)^\top$ denotes the correction term of the $k$ th iteration step. The start solution $\boldsymbol{\xi}^0$ is given.The term $\nabla F(\boldsymbol{\xi}^k)\,\mathbf{c}^k$ denotes the linearization, which is the derivative of $F$ at the linearization point $\boldsymbol{\xi}^k$ in the direction $\mathbf{c}^k$. For the GPME the function $F$ is defined via (1a),(1b) as:

$$\begin{aligned} F(\mathbf{u}^k, p^k) &:= \begin{pmatrix} F_1(\mathbf{u}^k, p^k) \\ F_2(\mathbf{u}^k, p^k) \end{pmatrix} \\ &= \begin{pmatrix} \frac{1}{\varepsilon}\left(\mathbf{u}^k\cdot\nabla\right)\frac{\mathbf{u}^k}{\varepsilon} + \frac{1}{\varepsilon\varrho_f}\nabla p^k\varepsilon - \frac{\nu_e}{\varepsilon}\Delta\mathbf{u}^k + \frac{\nu_f}{\kappa}\mathbf{u}^k + \frac{1.75}{\sqrt{150}}\frac{1}{\sqrt{\kappa}}\frac{\|\mathbf{u}^k\|}{\varepsilon^{3/2}}\mathbf{u}^k \\ \nabla\cdot\mathbf{u}^k \end{pmatrix}. \end{aligned} \tag{3}$$

The derivation of the linearization of the linear terms of (3) is trivial. Therefore it is only shown here for the non-linear terms $\mathcal{N}_1(\mathbf{u}) := \frac{1}{\varepsilon}(\mathbf{u}\cdot\nabla)\frac{\mathbf{u}}{\varepsilon}$ and $\mathcal{N}_2(\mathbf{u}) := \frac{1.75}{\sqrt{150}}\frac{1}{\sqrt{\kappa}}\frac{\|\mathbf{u}\|}{\varepsilon^{3/2}}\mathbf{u}$ (index $k$ is omitted for simplicity):

$$\begin{aligned}
\nabla_{\mathbf{u}}\mathcal{N}_1(\mathbf{u})\cdot\mathbf{c}_u &= \lim_{\lambda\to 0}\tfrac{1}{\lambda}\left(\mathcal{N}_1(\mathbf{u}+\lambda\mathbf{c}_u)-\mathcal{N}_1(\mathbf{u})\right)\\
&= \tfrac{1}{\varepsilon}\lim_{\lambda\to 0}\tfrac{1}{\lambda}\left[\left((\mathbf{u}+\lambda\mathbf{c}_u)\cdot\nabla\right)\tfrac{(\mathbf{u}+\lambda\mathbf{c}_u)}{\varepsilon}-(\mathbf{u}\cdot\nabla)\tfrac{\mathbf{u}}{\varepsilon}\right]\\
&= \tfrac{1}{\varepsilon}\lim_{\lambda\to 0}\tfrac{1}{\lambda}\left[(\mathbf{u}\cdot\nabla)\tfrac{(\mathbf{u}+\lambda\mathbf{c}_u)}{\varepsilon}+(\lambda\mathbf{c}_u\cdot\nabla)\tfrac{(\mathbf{u}+\lambda\mathbf{c}_u)}{\varepsilon}-(\mathbf{u}\cdot\nabla)\tfrac{\mathbf{u}}{\varepsilon}\right]\\
&= \tfrac{1}{\varepsilon}\lim_{\lambda\to 0}\tfrac{1}{\lambda}\left[(\mathbf{u}\cdot\nabla)\tfrac{\mathbf{u}}{\varepsilon}+(\mathbf{u}\cdot\nabla)\tfrac{\lambda\mathbf{c}_u}{\varepsilon}+(\lambda\mathbf{c}_u\cdot\nabla)\tfrac{\mathbf{u}}{\varepsilon}\right.\\
&\qquad\left.+(\lambda\mathbf{c}_u\cdot\nabla)\tfrac{\lambda\mathbf{c}_u}{\varepsilon}-(\mathbf{u}\cdot\nabla)\tfrac{\mathbf{u}}{\varepsilon}\right]\\
&= \tfrac{1}{\varepsilon}\lim_{\lambda\to 0}\left[(\mathbf{u}\cdot\nabla)\tfrac{\mathbf{c}_u}{\varepsilon}+(\mathbf{c}_u\cdot\nabla)\tfrac{\mathbf{u}}{\varepsilon}+(\mathbf{c}_u\cdot\nabla)\tfrac{\lambda\mathbf{c}_u}{\varepsilon}\right]\\
&= \tfrac{1}{\varepsilon}(\mathbf{u}\cdot\nabla)\tfrac{\mathbf{c}_u}{\varepsilon}+\tfrac{1}{\varepsilon}(\mathbf{c}_u\cdot\nabla)\tfrac{\mathbf{u}}{\varepsilon}
\end{aligned}$$

The linearization of term $\mathcal{N}_2$, the non-linear drag term, requires special attention, as the absolute value of $\mathbf{u}$ leads to non-differential behavior at $\mathbf{u}=\mathbf{0}$. For sake of brevity, the coefficient of the term is summarized in the constant $C$.

$$\begin{aligned}
\nabla_{\mathbf{u}}\mathcal{N}_2(\mathbf{u})\cdot\mathbf{c}_u &= \lim_{\lambda\to 0}\frac{1}{\lambda}\left(\mathcal{N}_2(\mathbf{u}+\lambda\mathbf{c}_u)-\mathcal{N}_2(\mathbf{u})\right)\\
&= C\lim_{\lambda\to 0}\frac{1}{\lambda}\left[\|\mathbf{u}+\lambda\mathbf{c}_u\|(\mathbf{u}+\lambda\mathbf{c}_u)-\|\mathbf{u}\|\mathbf{u}\right]\\
&= C\lim_{\lambda\to 0}\frac{1}{\lambda}\left[\|\mathbf{u}+\lambda\mathbf{c}_u\|\mathbf{u}+\lambda\|\mathbf{u}+\lambda\mathbf{c}_u\|\mathbf{c}_u-\|\mathbf{u}\|\mathbf{u}\right]
\end{aligned}$$

Note that the change of each component of the velocity field will influence the value of the non-linear drag term in each other component as well. This poses a great problem, as exact accounting for this correlation is not possible within a linear system and even approximation would largely increase the population of the system matrix.

However, as the influence of the non-linear drag term is only expected to be prominent for very specific parameter combinations, the value of $\|\mathbf{u}+\lambda\mathbf{c}_u\|$ will be approximated using only the values of the velocity field $\mathbf{u}$ of the previous iteration step.

$$\begin{aligned}
\nabla_{\mathbf{u}}\mathcal{N}_2(\mathbf{u})\cdot\mathbf{c}_u &\approx C\lim_{\lambda\to 0}\frac{1}{\lambda}\left[\|\mathbf{u}\|\mathbf{u}+\lambda\|\mathbf{u}\|\mathbf{c}_u-\|\mathbf{u}\|\mathbf{u}\right]\\
&= C\lim_{\lambda\to 0}\left[\|\mathbf{u}\|(\mathbf{c}_u)\right]\\
&= \frac{1.75}{\sqrt{150}}\frac{1}{\sqrt{\kappa}}\frac{\|\mathbf{u}\|}{\varepsilon^{3/2}}\mathbf{c}_u
\end{aligned}$$

The linearization of $F$ is then given by:

$$\nabla F(\mathbf{u}^k, p^k) \cdot \begin{pmatrix} \mathbf{c}_u \\ c_p \end{pmatrix} = \begin{pmatrix} \nabla F_1(\boldsymbol{\xi}^k) \\ \nabla F_2(\boldsymbol{\xi}^k) \end{pmatrix} \cdot \begin{pmatrix} \mathbf{c}_u \\ c_p \end{pmatrix} \tag{4}$$

with

$$\nabla F_1(\boldsymbol{\xi}^k) \cdot \begin{pmatrix} \mathbf{c}_u \\ c_p \end{pmatrix} = \frac{1}{\varepsilon}(\mathbf{u}^k \cdot \nabla)\frac{\mathbf{c}_u}{\varepsilon} + \frac{1}{\varepsilon}(\mathbf{c}_u \cdot \nabla)\frac{\mathbf{u}^k}{\varepsilon} + \frac{1}{\varrho_f \epsilon}\nabla c_p \epsilon - \frac{\nu_e}{\varepsilon}\Delta \mathbf{c}_u + \frac{\nu}{\kappa}\mathbf{c}_u + \frac{1.75}{\sqrt{150}}\frac{1}{\sqrt{\kappa}}\frac{\|\mathbf{u}^k\|}{\varepsilon^{3/2}}\mathbf{c}_u$$

and

$$\nabla F_2(\boldsymbol{\xi}^k) \cdot \begin{pmatrix} \mathbf{c}_u \\ c_p \end{pmatrix} = \nabla \cdot \mathbf{c}_u.$$

To retain the boundary conditions postulated by equations (1c) to (1e), each correction term $(\mathbf{c}_u, c_p)^\top$ has to satisfy

$$\begin{aligned} \mathbf{c}_u &= \mathbf{0} && \text{on } \Gamma_{\mathtt{in}} \\ (-\boldsymbol{\mathcal{I}} c_p + \frac{\nu_e}{\varepsilon}\nabla \mathbf{c}_u) \cdot \mathbf{n} &= 0 && \text{on } \Gamma_{\mathtt{out}} \\ \mathbf{c}_u &= \mathbf{0} && \text{on } \Gamma_0 \ . \end{aligned}$$

## 2.3 Weak Formulation

To solve a problem using finite element methods, a variational formulation of the problem must be given. It can be derived by multiplying the equation with some test functions, integrating over the domain, applying integration by parts and the Gauss theorem. Therefore the domain $\Omega$ has to be a Lipschitz domain [McL00].
The solution space for the velocity $\mathbf{u}$ is defined as

$$\mathcal{U}(\Omega) := \{\mathbf{u} : \mathbf{u} \in H^1(\Omega)^2,\ \mathbf{u}|_{\Gamma_{\mathtt{in}}} = \mathbf{g},\ \mathbf{u}|_{\Gamma_0} = \mathbf{0}\} \tag{5}$$

And the corresponding spaces of the velocity and pressure component of the correction term are defined as:

$$\mathcal{V}_u(\Omega) := \{\mathbf{u} : \mathbf{u} \in H^1(\Omega)^2,\ \mathbf{u}|_{\Gamma_{\mathtt{in}} \cup \Gamma_0} = \mathbf{0}\} \tag{6}$$

$$\mathcal{V}_p(\Omega) := \{p : p \in L^2(\Omega)\}, \tag{7}$$

Where $L^2$ is the Lebesgue[1] space of equivalence classes of square integrable functions in $\Omega$ and $H^1$ the Sobolev[2] space of equivalence classes of square integrable functions with square integrable (weak) first-derivatives in $\Omega$.

The single utilized functions are elements of these function spaces as follows:

$$p, c_p \in \mathcal{V}_p, \quad \mathbf{c}_u \in \mathcal{V}_u, \quad \mathbf{u} \in \mathcal{U}.$$

The indices $k$ of the single functions have been omitted.

In each Newton step (2) a problem of the following form has to be solved:

**Problem**

Find $\mathbf{c} = (\mathbf{c}_u, c_p)^\top$ with $\mathbf{c}_u \in \mathcal{V}_u$ and $c_p \in \mathcal{V}_p$ such that

$$\begin{aligned} \int_\Omega \nabla F_1(\boldsymbol{\xi}^k)\, \mathbf{c} \circ \mathbf{v}\, \mathrm{d}\mathbf{x} = & \quad \int_\Omega F_1(\boldsymbol{\xi}^k) \circ \mathbf{v}\, \mathrm{d}\mathbf{x} \\ \int_\Omega \nabla F_2(\boldsymbol{\xi}^k)\, \mathbf{c}\, q\, \mathrm{d}\mathbf{x} = & \quad \int_\Omega F_2(\boldsymbol{\xi}^k)\, q\, \mathrm{d}\mathbf{x} \end{aligned} \tag{8}$$

for all test functions $\mathbf{v} \in \mathcal{V}_u(\Omega)$ and $q \in \mathcal{V}_p(\Omega)$, where $\circ$ denotes multiplication by components.

The application of the Gauss theorem mentioned above, leads to a formulation of the left-hand side of the equation:

$$\begin{aligned} &\int_\Omega \nabla F_1(\boldsymbol{\xi}^k)\mathbf{c} \circ \mathbf{v}\, \mathrm{d}\mathbf{x} \\ &\quad = \int_\Omega \frac{1}{\varepsilon^2}(\mathbf{u}^k \cdot \nabla)\, \mathbf{c}_u \circ \mathbf{v} + \frac{1}{\varepsilon^2}(\mathbf{c}_u \cdot \nabla)\mathbf{u}^k \circ \mathbf{v} + \frac{\nu}{\kappa}\mathbf{c}_u \circ \mathbf{v} - \frac{1}{\varrho_f} c_p(\nabla \circ \mathbf{v}) \\ &\qquad + \frac{1.75}{\sqrt{150}} \frac{1}{\sqrt{\kappa}} \frac{1}{\varepsilon^{3/2}} \|\mathbf{u}^k\| \mathbf{c}_u \circ \mathbf{v} + \frac{\nu_e}{\varepsilon} \nabla \mathbf{c}_u : \nabla \mathbf{v}\, \mathrm{d}\mathbf{x} \end{aligned}$$

and

$$\int_\Omega \nabla F_2(\boldsymbol{\xi}^k)\mathbf{c}\, q\, \mathrm{d}\mathbf{x} \;=\; \int_\Omega (\nabla \cdot \mathbf{c}_u) q\, \mathrm{d}\mathbf{x}.$$

For the right-hand side the analogous terms

$$\begin{aligned} \int_\Omega F_1(\boldsymbol{\xi}^k) \circ \mathbf{v}\, \mathrm{d}\mathbf{x} \quad = \quad & \int_\Omega \frac{1}{\varepsilon^2} \left(\mathbf{u}^k \cdot \nabla\right) \mathbf{u}^k \circ \mathbf{v} + \frac{\nu}{\kappa}\mathbf{u}^k \circ \mathbf{v} - \frac{1}{\varrho_f} p(\nabla \circ \mathbf{v}) \\ & + \frac{1.75}{\sqrt{150}} \frac{1}{\sqrt{\kappa}} \frac{1}{\varepsilon^{3/2}} \|\mathbf{u}^k\| \mathbf{u}^k \circ \mathbf{v} + \frac{\nu_e}{\varepsilon} \nabla \mathbf{u}^k : \nabla \mathbf{v}\, \mathrm{d}\mathbf{x} \end{aligned}$$

[1] Henri Len Lebesgue, (∗ 1875, † 1941)

[2] Sergei Lvovich Sobolev, (∗ 1908, † 1989)

and

$$\int_\Omega F_2(\boldsymbol{\xi}^k)\, q \,\mathrm{d}\mathbf{x} \quad = \quad \int_\Omega (\nabla \cdot \mathbf{u}^k) q \,\mathrm{d}\mathbf{x}$$

are obtained. The operator : denotes the scalar product of two matrices ($\nabla\mathbf{a} : \nabla\mathbf{b} = (\sum_i \frac{\partial a_i}{\partial x_j}\frac{\partial b_i}{\partial x_j})_j$).

Using the finite element method for the discretization, the linear variational formulation (8) results in a stiffness matrix, which corresponds to $\nabla F(\boldsymbol{\xi}^k)$ in the step (2) of Newtons method. During the discretization process the stiffness matrix and residual vector have to be assembled.

# 3 The Commented Program

## 3.1 Preliminaries

The GPME tutorial needs the following two input files:

- `porous_media.xml` A xml-file containing all information needed to execute the program. It is read in by the program at runtime and thus does not require the recompilation of the program when parameters are changed. Parameters for example defining the termination condition of the non-linear and linear solver are listed as well as parameters of porosity and permeability.
- Geometry data: The file containing the geometry is specified in the parameter file. The file corresponding to the geometry shown in figure 1 is called `porous2d_barrier.inp`.

For the simulation of the model considering 3 spatial dimensions, the defined variable `DIMENSION` within the `porous_media.h` file needs to be adapted. The xml-file `porous_media3d.xml` and the geometry data contained in the file `porous3d_barrier.inp` are utilized in this case.

HiFlow$^3$ does not generate meshes for the domain $\Omega$. Meshes in `*.inp` and `*.vtu` format can be read in. It is possible to extend the reader for other formats. Some geometry data is provided in the test/data folder. Furthermore it is possible to generate other geometries by using external programs (Mesh generators) or by hand. Both formats provide the possibility to mark cell or facets by material numbers.
To distinguish different boundary conditions on the boundary different material numbers are set. The parameter file defines the meaning of the material number: In the parameter file (`porous_media.xml`) you find the the boundary parameters InflowMaterial and OutflowMaterial. In this case the variable `InflowMaterial` is set to 10 and the variable `OutflowMaterial` to 12. In the function `prepare_bc()`, see section 3.3.4, these parameters are read in, so that the program can distinguish the different parts of the boundary and set the correct boundary condition.

To execute the program in sequential mode type **./porous_media** .

## 3.2 Main Function

The main function starts the simulation of the porous flow problem (`porous_media.cc`: line 185 ff.).

```
.
int main ( int argc, char** argv )
{
    // initialize MPI
    MPI_Init ( &argc, &argv );

    // choose parameterfile depending on dimension
```

```
    if ( DIMENSION == 2 )
        PARAM_FILENAME = "porous_media.xml";
    else
        PARAM_FILENAME = "porous_media3d.xml";
    try
    {
        // run application
        GPME app;
        app.run ( );
    }
    catch ( std::exception& e )
    {
        std::cerr << "\nProgram ended with uncaught exception.\n";
        std::cerr << e.what ( ) << "\n";
        return -1;
    }

    // finalize MPI
    MPI_Finalize ( );

    return 0;
}
```

## 3.3 Member Functions

Following member functions are components of the Generalized Porous Media Equation tutorial:

- `run()`
- `prepare_mesh()`
- `prepare()`
- `prepare_bc()`
- `solve()`
- `visualize()`
- `compute_residual()`
- `compute_jacobian()`

### 3.3.1 run()

The member function `run()` calls the functions `solve()` and `visualize()` to solve the stationary flow problem and to generate the data for the visualization. The function is defined in the class `GPME` (`porous_media.cc`: line 39 ff.).

*Remark:* At the current state, there are two main applications of the GPME implemented, which require different parameters and non-dimensional scales. Setting the variable `Type` of the `FlowModel` within the configuration file to `"Column"` will execute a computation utilizing the geometry shown in figure 1. If `Type` is set to `"Channel"`, computations are carried out for a simple, uniform channel (See e.g. figure 8).

```
    virtual void run ( )
    {
        // string simul_name_ used as prefix to all output data
        simul_name_ = params_["Output"]["SolutionFilename"].get<std::↩
            string>( );

        // set the name of the log files
        std::ofstream info_log ( ( simul_name_ + "_info_log" ).c_str (↩
             ) );
        LogKeeper::get_log ( "info" ).set_target ( &info_log );

        // output parameters for logging
        LOG_INFO ( "parameters", params_ );

        // The type of mesh is given in the xml-files
        geometry_ = params_["FlowModel"]["Type"].get<std::string>( );

        // Test wether a suitable type of mesh was stated.
        if ( geometry_ != std::string ( "Channel" ) )
        {
            if ( geometry_ != std::string ( "Column" ) )
            {
                throw UnexpectedParameterValue ( "Geometry.Type", ↩
                    geometry_ );
            }
        }

        // store rank of processor
        MPI_Comm_rank ( comm_, &rank_ );
        // store number of processes/partitions
        MPI_Comm_size ( comm_, &num_partitions_ );

        // the time that is needed to prepare the mesh is measured
        setbuf ( stdout, NULL );
        start_timer ( "Reading Mesh ...", start_t, ptm );

        // Read, refine and partition mesh.
        prepare_mesh ( );
        end_timer ( start_t, end_t, diff_t, ptm );
        LOG_INFO ( "Reading Mesh ...      ", diff_t << "s" );

        // Set up datastructures and read in some parameters.
        prepare ( );
        LOG_INFO ( "simulation", "Solving stationary problem" );
        std::cout << "Solving problem" << std::endl;
        info_log.flush ( );

        // Solve nonlinear problem
        solve ( );

        // Write visualization data of the solution in a file.
        visualize ( );
    }
```

### 3.3.2 prepare_mesh()

The member function `prepare_mesh()` reads in the mesh (`porous_media.cc`: line 213 ff.)

```
void GPME::prepare_mesh ( )
{
    // The following needs to be done only by one process, therefore ↩
        it is only
```

```
// done by the process whose rank = MASTER_RANK
if ( rank ( ) == MASTER_RANK )
{
    // The name of the mesh is given in the xml-file. The program reads it in
    // and uses this mesh
    const std::string mesh_name = params_["Mesh"][geometry_].get<std::string>( );
    std::string mesh_filename = std::string ( DATADIR ) + mesh_name;
    master_mesh_ = read_mesh_from_file ( mesh_filename, DIMENSION, DIMENSION, 0 );

    // The initial mesh can be refined if it is too coarse in the beginning. If
    // this is desired, the variable RefLevel can be set to the desired level
    // in the xml-file
    refinement_level_ = 0;
    const int initial_ref_lvl = params_["Mesh"]["InitialRefLevel"].get<int>( );

    // refine mesh globally until the desired refinement level RefLevel is
    // reached
    for ( int r = 0; r < initial_ref_lvl; ++r )
    {
        master_mesh_ = master_mesh_->refine ( );
        ++refinement_level_;
    }
    LOG_INFO ( "mesh", "Refinement level = " << refinement_level_ );
}

// If the computation is executed parallel, each process computes the solution
// on one part of the mesh. The distribution is done via the following
// function call
MeshPtr local_mesh = partition_and_distribute ( master_mesh_, MASTER_RANK,
                                                comm_ );
assert ( local_mesh != 0 );
SharedVertexTable shared_verts;

// Ghost cells occur on the boundaries between two cell partitions. Since in
// the finite element method cells depend on their neighbor-cells, some need
// to be computed by two processes, but belong only to one. The other process
// computes on a so called ghost cell.
mesh_ = compute_ghost_cells ( *local_mesh, comm_, shared_verts );

// output on console
std::ostringstream rank_str;
rank_str << rank ( );

// write out mesh data
PVtkWriter writer ( comm_ );
std::string output_file = std::string ( simul_name_ + "_mesh_local.pvtu" );
writer.add_all_attributes ( *mesh_, true );
writer.write ( output_file.c_str ( ), *mesh_ );

// create boundary mesh and write out boundary mesh data
MeshPtr bdy_mesh = MeshPtr ( mesh_->extract_boundary_mesh ( ) );
VtkWriter bdy_writer;
```

```
    bdy_writer.add_attribute ( "_mesh_facet_index_", DIMENSION - 1 );
    std::stringstream bdy_sstr;
    bdy_sstr << simul_name_ << "_bdy_sd_" << rank_ << ".vtu";
    bdy_writer.write ( bdy_sstr.str ( ).c_str ( ), *bdy_mesh );
}
```

### 3.3.3 prepare()

The member function `prepare()` reads in the needed parameters, initializes the linear algebra objects and calls the member function `prepare_bc()` (`porous_media.cc`: line 280 ff.).

*Remark:* The computations of the program presented in this tutorial are carried out using non-dimensional scales, which are dependent on average fluid velocity, reference height and the resulting reference pressure. In particular the computation of the average velocity of the fluid depends on the selected `FlowModel` (see section 3.3.1) and the number of spatial dimensions. For the problems presented in this tutorial, the corresponding values are computed based on the maximal inflow velocity and the given reference height (cf. sec. 5.1.2).

The necessity and importance of the parameter `ContiWeight` is discussed in section 6.

```
void GPME::prepare ( )
{
    // prepare model parameters
    rho_ = params_["FlowModel"]["Density"].get<double>( );
    mu_ = params_["FlowModel"]["Viscosity"].get<double>( );
    conti_weight = params_["FlowModel"]["ContiWeight"].get<double>( );

    // The parameter u_max is read in and the avarage flow speed um_ ↩
        is computed
    // depending on the setting. R_in is the radius of the inflow tube↩
        and R_out
    // is the radius of the outflow tube.
    // 2D-Channel: um_=umax * 2/3
    // 2D-Column:  um_=umax * 2/3 * R_in/R_out = 2/3*0.3= 1/5
    // 3D-Channel: um_=umax * 1/2
    // 3D-Column:  um_=umax * 1/2 * (R_in/R_out)^2 = 1/2*0.3*0.3 = ↩
        9./200
    if ( DIMENSION == 2 )
    {
        if ( geometry_ == std::string ( "Channel" ) )
        {
            Um_ = 2 * params_["FlowModel"]["InflowSpeed"].get<double>(↩
                ) / 3;
            Href_ = params_["FlowModel"]["HeightRef"].get<double>( );
            D_ = params_["FlowModel"]["InflowDiameter"].get<double>( )↩
                ;
        }
        else
        { // Geometry Column
            Um_ = params_["FlowModel"]["InflowSpeed"].get<double>( ) /↩
                5;
            Href_ = params_["FlowModel"]["HeightRef"].get<double>( );
            // If this geometry is used, the inflow diameter is only
            // 0.3 * the diameter of the tube so this has to be ↩
                corrected
```

```
            D_ = .3 * params_["FlowModel"]["InflowDiameter"].get<↩
                double>( );
        }
    }
    else
    {
        if ( geometry_ == std::string ( "Channel" ) )
        {
            Um_ = params_["FlowModel"]["InflowSpeed"].get<double>( ) /↩
                2;
            Href_ = params_["FlowModel"]["HeightRef"].get<double>( );
            D_ = params_["FlowModel"]["InflowDiameter"].get<double>( )↩
                ;
        }
        else
        { // Geometry Column
            Um_ = .3 * .3 * params_["FlowModel"]["InflowSpeed"].get<↩
                double>( ) / 2;
            Href_ = params_["FlowModel"]["HeightRef"].get<double>( );
            // The inflow diameter is only 0.3 * the diameter of the ↩
                tube if
            // this geometry is used so this has to be corrected
            D_ = .3 * params_["FlowModel"]["InflowDiameter"].get<↩
                double>( );
        }
    }

    start_timer ( "Preparing Space ...", start_t, ptm );

    // prepare space, therefore read in polynomial degree of finite ↩
        elements of
    // different variables
    std::vector< int > degrees ( DIMENSION + 1 );
    const int u_deg = params_["FiniteElements"]["VelocityDegree"].get<↩
        int>( );
    const int p_deg = params_["FiniteElements"]["PressureDegree"].get<↩
        int>( );
    for ( int c = 0; c < DIMENSION; ++c )
    {
        degrees.at ( c ) = u_deg;
    }
    degrees.at ( DIMENSION ) = p_deg;

    // initialize finite element space
    space_.Init ( degrees, *mesh_ );

    // compute matrix graph
    SparsityStructure sparsity;
    global_asm_.compute_sparsity_structure ( space_, sparsity );

    // prepare linear algebra structures
    couplings_.Clear ( );
    couplings_.Init ( communicator ( ), space_.dof ( ) );

    couplings_.InitializeCouplings ( sparsity.off_diagonal_rows,
                                     sparsity.off_diagonal_cols );

    // initialization of needed global matrix and vectors and setting ↩
        them to 0

    CoupledMatrixFactory<Scalar> CoupMaFact;
    matrix_ = CoupMaFact.Get (
                              params_["LinearAlgebra"]["NameMatrix"].↩
                                  get<std::string>( ) )->
            params ( params_["LinearAlgebra"] );
    matrix_->Init ( comm_, couplings_ );
    matrix_->InitStructure ( vec2ptr ( sparsity.diagonal_rows ),
                             vec2ptr ( sparsity.diagonal_cols ),
```

```
                                sparsity.diagonal_rows.size ( ),
                                vec2ptr ( sparsity.off_diagonal_rows ),
                                vec2ptr ( sparsity.off_diagonal_cols ),
                                sparsity.off_diagonal_rows.size ( ) );
    matrix_->Zeros ( );

    CoupledVectorFactory<Scalar> CoupVecFact;

    sol_ = CoupVecFact.Get (
                                params_["LinearAlgebra"]["NameVector"].↩
                                    get<std::string>( ) )->
            params ( params_["LinearAlgebra"] );
    sol_->Init ( comm_, couplings_ );
    sol_->InitStructure ( );
    sol_->Zeros ( );

    prev_sol_ = CoupVecFact.Get (
                                 params_["LinearAlgebra"]["NameVector↩
                                     "].get<std::string>( ) )->
            params ( params_["LinearAlgebra"] );
    prev_sol_->Init ( comm_, couplings_ );
    prev_sol_->InitStructure ( );
    prev_sol_->Zeros ( );

    cor_ = CoupVecFact.Get (
                                params_["LinearAlgebra"]["NameVector"].↩
                                    get<std::string>( ) )->
            params ( params_["LinearAlgebra"] );
    cor_->Init ( comm_, couplings_ );
    cor_->InitStructure ( );
    cor_->Zeros ( );

    res_ = CoupVecFact.Get (
                                params_["LinearAlgebra"]["NameVector"].↩
                                    get<std::string>( ) )->
            params ( params_["LinearAlgebra"] );
    res_->Init ( comm_, couplings_ );
    res_->InitStructure ( );
    res_->Zeros ( );

    end_timer ( start_t, end_t, diff_t, ptm );
    LOG_INFO ( "Preparing Space ...      ", diff_t << "s" );

    start_timer ( "Preparing BC ...", start_t, ptm );
    // prepare dirichlet BC
    prepare_bc ( );
    end_timer ( start_t, end_t, diff_t, ptm );
    LOG_INFO ( "Preparing BC ...         ", diff_t << "s" );

    // Assign Values for Epsilon and Kappa
    // eps_free is the porosity where no porous media is found
    // eps_por is the porosity of the porous media
    // kap_free is the permeability where no porous media is found
    // kap_por is the permeability of the porous media
    eps_free = params_["FlowModel"]["PorosityFree"].get<double>( );
    eps_por = params_["FlowModel"]["PorosityPor"].get<double>( );
    kap_por = params_["FlowModel"]["PermeabilityPor"].get<double>( );
    kap_free = params_["FlowModel"]["PermeabilityFree"].get<double>( )↩
        ;
}
```

### 3.3.4 prepare_bc()

The member function `prepare_bc()` sets up the Dirichlet boundary values (`porous_media.cc`: line 410 ff.).

```
void GPME::prepare_bc ( )
{
    dirichlet_dofs_.clear ( );
    dirichlet_values_.clear ( );

    // compute Dirichlet values, distinguish between different geometries and
    // dimensions. The material numbers of the facets which lie on the boundary
    // of the mesh determine wether this facets is an inflow boundary, and outflow
    // boundary or a dirichlet boundary (i.e. a wall).
    if ( geometry_ == std::string ( "Channel" ) )
    {
        // read in material numbers of inflow and outflow boundary
        const int inflow_bdy = params_["Boundary"]["InflowMaterial"].get<int>( );
        const int outflow_bdy = params_["Boundary"]["OutflowMaterial"].get<int>( );
        U_max_ = params_["FlowModel"]["InflowSpeed"].get<double>( );

        if ( DIMENSION == 2 )
        {
            ChannelFlowBC bc[2] = {
                                    ChannelFlowBC ( 0, D_, U_max_, inflow_bdy, outflow_bdy ),
                                    ChannelFlowBC ( 1, D_, U_max_, inflow_bdy, outflow_bdy )
            };

            for ( int var = 0; var < DIMENSION; ++var )
            {
                compute_dirichlet_dofs_and_values ( bc[var], space_,
                                                    var,
                                                    dirichlet_dofs_,
                                                    dirichlet_values_ );
            }
        }
        else
        {
            assert ( DIMENSION == 3 );
            ChannelFlowBC3d bc[3] = {
                                      ChannelFlowBC3d ( 0, D_, U_max_, inflow_bdy, outflow_bdy ),
                                      ChannelFlowBC3d ( 1, D_, U_max_, inflow_bdy, outflow_bdy ),
                                      ChannelFlowBC3d ( 2, D_, U_max_, inflow_bdy, outflow_bdy )
            };
            for ( int var = 0; var < DIMENSION; ++var )
            {
                compute_dirichlet_dofs_and_values ( bc[var], space_,
                                                    var,
                                                    dirichlet_dofs_,
                                                    dirichlet_values_ );
            }
        }
    }
    else if ( geometry_ == std::string ( "Column" ) )
    {
        // read in material numbers of inflow and outflow boundary
        const int inflow_bdy = params_["Boundary"]["InflowMaterial"].get<int>( );
        const int outflow_bdy = params_["Boundary"]["OutflowMaterial"].get<int>( );
```

```
        U_max_ = params_["FlowModel"]["InflowSpeed"].get<double>( );

        if ( DIMENSION == 2 )
        {
            ChannelFlowBC bc[2] = {
                                    ChannelFlowBC ( 0, D_, U_max_, ↩
                                        inflow_bdy, outflow_bdy ),
                                    ChannelFlowBC ( 1, D_, U_max_, ↩
                                        inflow_bdy, outflow_bdy )
            };

            for ( int var = 0; var < DIMENSION; ++var )
            {
                compute_dirichlet_dofs_and_values ( bc[var], space_,
                                                    var,
                                                    dirichlet_dofs_,
                                                    dirichlet_values_ ↩
                                                        );
            }
        }
        else
        {
            assert ( DIMENSION == 3 );
            ChannelFlowBC3d bc[3] = {
                                      ChannelFlowBC3d ( 0, D_, U_max_, ↩
                                          inflow_bdy, outflow_bdy ),
                                      ChannelFlowBC3d ( 1, D_, U_max_, ↩
                                          inflow_bdy, outflow_bdy ),
                                      ChannelFlowBC3d ( 2, D_, U_max_, ↩
                                          inflow_bdy, outflow_bdy )
            };
            for ( int var = 0; var < DIMENSION; ++var )
            {
                compute_dirichlet_dofs_and_values ( bc[var], space_,
                                                    var,
                                                    dirichlet_dofs_,
                                                    dirichlet_values_ ↩
                                                        );
            }
        }
    }
    else
    {
        assert ( false );
    }

    // apply boundary conditions to initial solution
    if ( !dirichlet_dofs_.empty ( ) )
    {
        // correct solution with dirichlet boundary conditions
        sol_->SetValues ( vec2ptr ( dirichlet_dofs_ ), dirichlet_dofs_↩
            .size ( ),
                          vec2ptr ( dirichlet_values_ ) );
    }
}
```

The boundary conditions are defined in the class `ChannelFlowBC` (`porous_media.h`: line 61 ff.).

```
struct ChannelFlowBC
{
    /// \brief constructor
    ///
    /// \param [in] var  index of variable
```

```
/// \param [in] D  channel diameter
/// \param [in] Um  maximum inflow
/// \param [in] inflow_bdy  material number of inflow boundary
/// \param [in] outflow_bdy  material number of outflow boundary

ChannelFlowBC ( int var, double D, double Um, int inflow_bdy,
                int outflow_bdy )
: var_ ( var ), R_ ( D / 2 ), Um_ ( Um ), inflow_bdy_ ( inflow_bdy ),
outflow_bdy_ ( outflow_bdy )
{
    assert ( var_ == 0 || var_ == 1 );
}

/// \brief evaluates values of all Dirichlet degrees of freedom which
///         lie on face.
///
/// \param [in] face face on which values of dirichlet dofs will be
///                 evaluated
/// \param [in] coords_on_face coordinates of dofs on face

std::vector<double> evaluate ( const Entity& face,
                               const std::vector<Coord>&
                                   coords_on_face ) const
{
    // stores values of Dirichlet dofs
    std::vector<double> values;

    // material number of face needed for evaluation
    const int material_num = face.get_material_number ( );

    // variables to distinguish type of boundary
    const bool outflow = ( material_num == outflow_bdy_ );
    const bool inflow = ( material_num == inflow_bdy_ );

    // Following Dirichlet boundary conditions are set: if the face lies on an
    // inflow boundary. On outflow, no Dirichlet boundaries are applied. In case
    // of no outflow we distinguish and if so set
    // u_x = 4*Um * y * (1-y) / D^2 and u_y = 0. Otherwise, set u_x = u_y = 0 .

    // check if face lies on the outflow boundary or not
    if ( !outflow )
    {
        // depending on number of dofs on face, set size of values
        values.resize ( coords_on_face.size ( ) );

        // loop over points on the face
        for ( int i = 0; i < static_cast < int > ( coords_on_face.size ( ) ); ++i )
        {
            // evaluate dirichlet function at each point and store coordinates of
            // each dof
            const Coord& pt = coords_on_face[i];

            if ( inflow )
            {
                if ( var_ == 0 )
                { // x-component
                    // values[i] = 4. * Um_ * (pt[1] - 0.35) * (0.65 - pt[1])/(D_ * D_);
                    values[i] = -Um_ * ( ( pt[1] / R_ - 1 ) * ( pt[1] / R_ - 1 ) - 1 );
```

```
                    }
                    else if ( var_ == 1 )
                    {
                        // y-component
                        values[i] = 0.;
                    }
                    else
                    {
                        assert ( false );
                    }
                }
                else
                {
                    // not inflow: u = 0
                    values[i] = 0.;
                }
            }
        }
        return values;
    }
    // index of variable
    const int var_;
    // radius of channel
    const double R_;
    // maximum inflow velocity
    const double Um_;
    // material number of inflow and outflow boundary
    const int inflow_bdy_, outflow_bdy_;
};
```

### 3.3.5 solve()

The member function `solve()` reads in some parameters and solves the nonlinear flow problem using the Newton method (`porous_ media.cc`: line 484 ff.).

```
void GPME::solve ( ) {

    // read in nonlinear solver parameters
    const int nls_max_iter =
            params_["NonlinearSolver"]["MaximumIterations"].get<int>( ↩
                );
    const double nls_abs_tol =
            params_["NonlinearSolver"]["AbsoluteTolerance"].get<double↩
                >( );
    const double nls_rel_tol =
            params_["NonlinearSolver"]["RelativeTolerance"].get<double↩
                >( );
    /*const double nls_div_tol =
                params_["NonlinearSolver"]["DivergenceLimit"].get<↩
                    double>();*/


  [...]

  start_timer ( "Setting up GMRES ...", start_t, ptm );

    // read in linear solver parameters
    GMRES<LAD> gmres;
    const int lin_max_iter =
            params_["LinearSolver"]["MaximumIterations"].get<int>( );
    const double lin_abs_tol =
```

```
        params_["LinearSolver"]["AbsoluteTolerance"].get<double>( ↩
            );
const double lin_rel_tol =
        params_["LinearSolver"]["RelativeTolerance"].get<double>( ↩
            );
const double lin_div_tol =
        params_["LinearSolver"]["DivergenceLimit"].get<double>( );
const int basis_size = params_["LinearSolver"]["BasisSize"].get<↩
    int>( );

// set up linear solver
gmres.InitControl ( lin_max_iter, lin_abs_tol, lin_rel_tol, ↩
    lin_div_tol );
gmres.InitParameter ( basis_size, "NoPreconditioning" );
gmres.SetupOperator ( *matrix_ );

end_timer ( start_t, end_t, diff_t, ptm );
LOG_INFO ( "Setting up GMRES ...      ", diff_t << "s" );

std::ofstream iteration ( ( "Data/" + simul_name_ + "_iteration" )↩
    .c_str ( ) );

// Write visualization data of the solution in a file.
visualize ( );

start_timer ( "Computing Residual ...", start_t, ptm );

// Newtons method is used to solve the nonlinear problem.  The
// functions compute_residual() and compute_jacobian()
// update the variables matrix_ and res_, respectively.
// The vector cor_ is set up to be used for the correction, and
// the solution state is stored in sol_ .
iter_ = 0;
compute_residual ( );

end_timer ( start_t, end_t, diff_t, ptm );
LOG_INFO ( "Computing Residual ...     ", diff_t << "s" );

// counter for Newton iterations
int iter = 0;

// variables for storing the average time it takes to compute the ↩
    residual,
// flow and Jacobian and to visualize and solve
double avgres ( 0. ), avgflow ( 0. ), avgvisu ( 0. ), avgsolv ( 0.↩
    ), avgjaco ( 0. );

// compute start residual
const double initial_res_norm = res_->Norm2 ( );
LOG_INFO ( "nonlinear", "Nonlinear solver starts with residual ↩
    norm "
           << initial_res_norm );
std::cout << "Nonlinear solver starts with residual norm "
        << initial_res_norm << std::endl;
double res_norm = initial_res_norm;
double prev_res_norm = initial_res_norm;
double norm_ratio ( 1. );
double step_;

// Newton method
while ( iter < nls_max_iter
        && res_norm > nls_abs_tol
        && res_norm > nls_rel_tol * initial_res_norm )
{
    // Solve DF * cor = res
    iter_ = iter;
    start_timer ( "Computing Jacobian ...", start_t, ptm );
```

```
        // updates matrix_ with the jacobian matrix
        compute_jacobian ( );

        end_timer ( start_t, end_t, diff_t, ptm );

        // the time it took to compute the Jacobian in this step (↩
            diff_t) is added
        // to the variable avgjaco to compute the average time
        avgjaco += diff_t;
        LOG_INFO ( "Computing Jacobian ...      ", diff_t << "s    ("
                  << avgjaco / ( iter + 1 ) << ")" );

        // Compute correction cor
        cor_->Zeros ( );
        std::cout << "[ Iteration " << iter
                << " ]:  Solving linear problem with residual: "
                << res_norm << " ." << std::endl;
        iteration << iter << ",  " << res_norm << ",  " << norm_ratio ↩
            << std::endl;
        start_timer ( "Solving ...", start_t, ptm );
        gmres.Solve ( *res_, cor_ );
        end_timer ( start_t, end_t, diff_t, ptm );
        avgsolv += diff_t;
        LOG_INFO ( "Solving ...                 ", diff_t << "s    ("
                  << avgsolv / ( iter + 1 ) << ")" );

        cor_->UpdateCouplings ( );

        // update sol = sol - cor
        step_ = 1.;
        std::cout << "Ratio of = " << norm_ratio << ", resulting ↩
            Steplength = "
                << step_ << std::endl;
        sol_->Axpy ( *cor_, -step_ ); /// damped Newton method
        sol_->UpdateCouplings ( );

        start_timer ( "Computing Flow Profile ...", start_t, ptm );

        // For computing the flow profile at x = 4 use ↩
            compute_flowprofile()
        // For computing the flow profile at x = 3 use ↩
            compute_flowprofile2()
        compute_flowprofile ( );
        // compute_flowprofile2();
        end_timer ( start_t, end_t, diff_t, ptm );

        // the time it took to compute the flowprofile in this step (↩
            diff_t) is
        // added to the variable avgflow to compute the average time
        avgflow += diff_t;
        LOG_INFO ( "Computing Flow Profile ...  ", diff_t << "s    ("
                  << avgflow / ( iter + 1 ) << ")" );

        start_timer ( "Visualizing ...", start_t, ptm );

        // visualize solution of each Newton step
        visualize ( );

        end_timer ( start_t, end_t, diff_t, ptm );

        // the time it took to viualize the solution in this step (↩
            diff_t) is added
        // to the variable avgvisu to compute the average time
        avgvisu += diff_t;
        LOG_INFO ( "Visualizing ...             ", diff_t << "s    (" ↩
            << avgvisu / ( iter + 1 ) << ")" );

        // Compute new residual
```

```
        start_timer ( "Computing new residual ...", start_t, ptm );

        compute_residual ( );

        prev_res_norm = res_norm;
        res_norm = res_->Norm2 ( );
        norm_ratio = res_norm / prev_res_norm;

        end_timer ( start_t, end_t, diff_t, ptm );
        avgres += diff_t;
        LOG_INFO ( "Computing new Residual ...  ", diff_t << "s    ("
                   << avgres / ( iter + 1 ) << ")" );
        ++iter;
    }

    std::cout << "Nonlinear Solver Residual " << res_norm << std::endl;
    std::cout << "Nonlinear Solver Steps " << iter << std::endl;
    LOG_INFO ( "Nonlinear solver residual", res_norm );
    LOG_INFO ( "Nonlinear solver steps", iter );
```

### 3.3.6 visualize()

The member function `visualize()` writes data for visualization of the current solution (`porous_media.cc`: line 661 ff.).

```
void GPME::visualize ( )
{
    // initialize visualization
    int num_intervals = 2;
    ParallelCellVisualization<double> visu ( space_, num_intervals,
        comm_, MASTER_RANK );

    // setup output file name
    std::stringstream input;
    input << simul_name_ << "_solution_tutorial";
    input << "_stationary";

    std::vector<double> material_number ( mesh_->num_entities ( mesh_
        ->tdim ( ) ), 0 );
    std::vector<double> remote_index ( mesh_->num_entities ( mesh_->
        tdim ( ) ), 0 );
    std::vector<double> sub_domain ( mesh_->num_entities ( mesh_->tdim
         ( ) ), 0 );

    for ( mesh::EntityIterator it = mesh_->begin ( mesh_->tdim ( ) );
          it != mesh_->end ( mesh_->tdim ( ) );
          ++it )
    {
        int temp1, temp2;
        mesh_->get_attribute_value ( "_remote_index_", mesh_->tdim ( )
            ,
                                     it->index ( ),
                                     &temp1 );
        mesh_->get_attribute_value ( "_sub_domain_", mesh_->tdim ( ),
                                     it->index ( ),
                                     &temp2 );
        remote_index.at ( it->index ( ) ) = temp1;
        sub_domain.at ( it->index ( ) ) = temp2;
        material_number.at ( it->index ( ) ) = mesh_->
            get_material_number ( mesh_->tdim ( ), it->index ( ) );
    }
```

```
    sol_->UpdateCouplings ( );

    visu.visualize ( EvalFeFunction<LAD>( space_, *( sol_ ), 0 ), "u" );
#if(DIMENSION >= 2)
    visu.visualize ( EvalFeFunction<LAD>( space_, *( sol_ ), 1 ), "v" );
#endif
#if(DIMENSION == 3)
    visu.visualize ( EvalFeFunction<LAD>( space_, *( sol_ ), 2 ), "w" );
#endif
    visu.visualize ( EvalFeFunction<LAD>( space_, *( sol_ ), DIMENSION ), "p" );

    visu.visualize_cell_data ( material_number, "Material Id" );
    visu.visualize_cell_data ( remote_index, "_remote_index_" );
    visu.visualize_cell_data ( sub_domain, "_sub_domain_" );

    visu.write ( input.str ( ) );
}
```

### 3.3.7 compute_residual()

The member function `compute_residual()` computes the residual for Newton's method (`porous_media.cc`: line 710 ff.).

```
void GPME::compute_residual ( )
{
    PorousMediaAssembler local_asm ( *sol_, mu_, rho_, Um_, Href_,
        conti_weight,
                                     eps_free, eps_por, kap_free,
                                         kap_por,
                                     geometry_ );

    global_asm_.assemble_vector ( space_, local_asm, *res_ );

    // correct boundary conditions -- set Dirichlet dofs to 0
    if ( !dirichlet_dofs_.empty ( ) )
    {
        std::vector<LAD::DataType> zeros ( dirichlet_dofs_.size ( ),
            0. );
        res_->SetValues ( vec2ptr ( dirichlet_dofs_ ), dirichlet_dofs_
            .size ( ),
                          vec2ptr ( zeros ) );
    }

    res_->UpdateCouplings ( );
}
```

The operator for the assembling of the residual is implemented in the class `PorousMediaAssembler` (`porous_media.h`: line 440 ff.).

```
void operator() ( const Element<double>& element,
            const Quadrature<double>& quadrature,
            LocalVector& lv )
    {
        AssemblyAssistant<DIMENSION, double>::initialize_for_element (
            element, quadrature );
```

```
        // interpolate previous values of solution and the gradient of↩
             the solution
        // to quadrature points
        for ( int v = 0; v < DIMENSION; ++v )
        {
            prev_vel_[v].clear ( );
            grad_prev_vel_[v].clear ( );
            evaluate_fe_function ( solution_, v, prev_vel_[v] );
            evaluate_fe_function_gradients ( solution_, v, ↩
                grad_prev_vel_[v] );
        }
        pressure_k_.clear ( );
        evaluate_fe_function ( solution_, DIMENSION, pressure_k_ );

        const int num_q = num_quadrature_points ( );

        double eps_local, kap_local;

        // enforce local constant porosity (epsilon) and permeability ↩
            (kappa)
        // on each element
        double eps = 0;
        double kap = 0;
        for ( int q = 0; q < num_q; ++q )
        {
            eps_local = evaluate_epsilon ( x ( q ) );
            kap_local = evaluate_kappa ( x ( q ) );
            if ( eps_local > eps )
            {
                eps = eps_local;
            }
            if ( kap_local > kap )
            {
                kap = kap_local;
            }
        }

        // compute the needed constants for computation of local ↩
            element residual
        double inveps = 1. / eps;
        double poweps = 1. / pow ( eps, ( 3 / 2 ) );
        inv_darcy = ref_l * ref_l / kap;
        inv_darcy_root = sqrt ( inv_darcy );

        // loop over quadrature points
        for ( int q = 0; q < num_q; ++q )
        {
            const double wq = w ( q );
            const double dJ = std::abs ( detJ ( q ) );

            // get previous solution in vector form
            Vec<DIMENSION, double> vel_k;
            for ( int var = 0; var < DIMENSION; ++var )
            {
                vel_k[var] = prev_vel_[var][q];
            }

            // compute norm of previous solution
            norm_ = sqrt ( dot ( vel_k, vel_k ) );
            // if (norm_ < 0.05) norm_=.05;

            // L1(v) = -1/rho*\int(p_k*div(v))
            // loop over variables of test functions
            for ( int v_var = 0; v_var < DIMENSION; ++v_var )
            {
                // loop over test functions
                for ( int i = 0; i < num_dofs ( v_var ); ++i )
```

```
    {
        // compute integral
        lv[dof_index ( i, v_var )] +=
                -wq * ( pressure_k_[q] * grad_phi ( i, q, ↩
                    v_var )[v_var] ) * dJ;
    }
}

// L2(v) = nu / eps * \int( \grad{u_k} : \grad{v} )
// loop over variables of test functions
for ( int v_var = 0; v_var < DIMENSION; ++v_var )
{
    // loop over test functions
    for ( int i = 0; i < num_dofs ( v_var ); ++i )
    {
        // compute integral
        lv[dof_index ( i, v_var )] +=
                wq * ( inv_reynolds * inveps *
                dot ( grad_phi ( i, q, v_var ),
                grad_prev_vel_[v_var][q] ) ) * dJ;
    }
}

// L3(v) = nu/kappa*\int(v)
// loop over variables of test functions
for ( int v_var = 0; v_var < DIMENSION; ++v_var )
{
    // loop over test functions
    for ( int i = 0; i < num_dofs ( v_var ); ++i )
    {
        // compute integral
        lv[dof_index ( i, v_var )] +=
                wq * ( inv_reynolds * inv_darcy * phi ( i,↩
                    q, v_var ) ) * dJ;
    }
}

// L4(q) = \int(q * div(u_k))
// index of pressure variable
const int q_var = DIMENSION;
// variable for divergence
double div_u_k = 0.;
// computing the divergence
for ( int d = 0; d < DIMENSION; ++d )
{
    div_u_k += grad_prev_vel_[d][q][d];
}
// loop over test functions
for ( int i = 0; i < num_dofs ( q_var ); ++i )
{
    // compute integral
    lv[dof_index ( i, q_var )] +=
            wq / ref_u * conti_ * ( div_u_k * phi ( i, q, ↩
                q_var ) ) * dJ;
}

// N1(v) = \int(u_k*\grad{u_k}*v)
// loop over variables of test functions
for ( int v_var = 0; v_var < DIMENSION; ++v_var )
{
    // loop over test functions
    for ( int i = 0; i < num_dofs ( v_var ); ++i )
    {
        // compute integral
        lv[dof_index ( i, v_var )] +=
                wq * ( inveps * inveps * dot ( ↩
                    grad_prev_vel_[v_var][q], vel_k )
                * phi ( i, q, v_var ) ) * dJ;
```

```
            }
        }

        // N2(v) = 1.75/(sqrt(150)*kappa*epsilon^(3/2)) * norm(u_k↩
            ) * v
        // loop over variables of test functions
        for ( int v_var = 0; v_var < DIMENSION; ++v_var )
        {
            // loop over test functions
            for ( int i = 0; i < num_dofs ( v_var ); ++i )
            {
                // compute integral
                lv[dof_index ( i, v_var )] +=
                        wq * ( inv_darcy_root * 1.75 / ( sqrt ( ↩
                            150 ) ) * poweps * norm_
                        * phi ( i, q, v_var ) ) * dJ;
            }
        }
    }
}
```

### 3.3.8 compute_jacobian()

The member function `compute_jacobian()` computes the Jacobian matrix for Newton's method (`porous_media.cc`: line 1406 ff.).

```
void GPME::compute_jacobian ( )
{
    PorousMediaAssembler local_asm ( *sol_, mu_, rho_, Um_, Href_, ↩
        conti_weight,
                                     eps_free, eps_por, kap_free, ↩
                                         kap_por, geometry_ );

    // assemble system matrix
    global_asm_.assemble_matrix ( space_, local_asm, *matrix_ );

    // correct BC -- set Dirichlet rows to identity
    if ( !dirichlet_dofs_.empty ( ) )
    {
        matrix_->diagonalize_rows ( vec2ptr ( dirichlet_dofs_ ), ↩
            dirichlet_dofs_.size ( ), 1. );
    }
}
```

The operator for the assembling of the jacobian is implemented in the class `PorousMediaAssembler` (`porous_media.h`: line 264 ff.).

```
    void operator() ( const Element<double>& element,
            const Quadrature<double>& quadrature,
            LocalMatrix& lm )
    {
        AssemblyAssistant<DIMENSION, double>::initialize_for_element (↩
             element, quadrature );

        // interpolate previous values of solution and the gradient of↩
             the solution
        // to quadrature points
        for ( int v = 0; v < DIMENSION; ++v )
        {
            prev_vel_[v].clear ( );
            grad_prev_vel_[v].clear ( );
```

```
        evaluate_fe_function ( solution_, v, prev_vel_[v] );
        evaluate_fe_function_gradients ( solution_, v, ↩
            grad_prev_vel_[v] );
    }

    const int num_q = num_quadrature_points ( );

    double eps_local, kap_local;

    // enforce local constant porosity (epsilon) and permeability ↩
        (kappa)
    // on each element
    double eps = 0;
    double kap = 0;
    for ( int q = 0; q < num_q; ++q )
    {
        eps_local = evaluate_epsilon ( x ( q ) );
        kap_local = evaluate_kappa ( x ( q ) );
        if ( eps_local > eps )
        {
            eps = eps_local;
        }
        if ( kap_local > kap )
        {
            kap = kap_local;
        }
    }

    // compute the needed constants for computation of local ↩
        element matrix
    double inveps = 1. / eps;
    double poweps = 1. / pow ( eps, ( 3 / 2 ) );
    inv_darcy = ref_l * ref_l / kap;
    inv_darcy_root = sqrt ( inv_darcy );

    // loop over quadrature points
    for ( int q = 0; q < num_q; ++q )
    {
        // compute weight for quadrature points
        const double wq = w ( q );
        // compute determinant of Jacobi matrix for each ↩
            quadrature point
        const double dJ = std::abs ( detJ ( q ) );

        // get previous solution in vector form
        Vec<DIMENSION, double> vel_k;
        for ( int var = 0; var < DIMENSION; ++var )
        {
            vel_k[var] = prev_vel_[var][q];
        }

        // compute norm of previous solution
        norm_ = sqrt ( dot ( vel_k, vel_k ) );

        // assemble L1(p, v) = - \int{p div{v}}
        // index of pressure variable
        const int p_var = DIMENSION;
        // loop over variables of test functions
        for ( int v_var = 0; v_var < DIMENSION; ++v_var )
        {
            // loop over test functions
            for ( int i = 0; i < num_dofs ( v_var ); ++i )
            {
                // loop over ansatz functions
                for ( int j = 0; j < num_dofs ( p_var ); ++j )
                {
                    // compute integral
                    lm ( dof_index ( i, v_var ), dof_index ( j, ↩
```

```
                p_var ) ) +=
                    -wq * ( phi ( j, q, p_var ) *
                    grad_phi ( i, q, v_var )[v_var] ) * dJ;
            }
        }
    }

    // assemble L2(u,v) = nu / eps * \int {\grad(u) : \grad(v)}
    // loop over variables of test functions and ansatz functions
    for ( int u_var = 0; u_var < DIMENSION; ++u_var )
    {
        // loop over test functions
        for ( int i = 0; i < num_dofs ( u_var ); ++i )
        {
            // loop over ansatz functions
            for ( int j = 0; j < num_dofs ( u_var ); ++j )
            {
                // compute integral
                lm ( dof_index ( i, u_var ), dof_index ( j, u_var ) ) +=

                    wq * ( inv_reynolds * inveps
                    * dot ( grad_phi ( j, q, u_var ), grad_phi ( i, q, u_var ) ) )
                    * dJ;
            }
        }
    }

    // assemble L3(u,v) = \int { nu/kappa * v }
    // loop over variables of test functions and ansatz functions
    for ( int u_var = 0; u_var < DIMENSION; ++u_var )
    {
        // loop over test functions
        for ( int i = 0; i < num_dofs ( u_var ); ++i )
        {
            // loop over ansatz functions
            for ( int j = 0; j < num_dofs ( u_var ); ++j )
            {
                // compute integral
                lm ( dof_index ( i, u_var ), dof_index ( j, u_var ) ) +=
                    wq * ( inv_reynolds * inv_darcy
                    * phi ( i, q, u_var ) ) * dJ;
            }
        }
    }

    // assemble L4(u, q) = \int{q div(u)}
    // index of pressure variable
    const int q_var = DIMENSION;
    // loop over variables of ansatz functions
    for ( int u_var = 0; u_var < DIMENSION; ++u_var )
    {
        // loop over test functions
        for ( int i = 0; i < num_dofs ( q_var ); ++i )
        {
            // loop over ansatz functions
            for ( int j = 0; j < num_dofs ( u_var ); ++j )
            {
                // compute integral
                lm ( dof_index ( i, q_var ), dof_index ( j, u_var ) ) +=
                    wq / ref_u * conti_ * ( phi ( i, q,
```

```
                        q_var )
                    * grad_phi ( j, q, u_var )[u_var] ) * ↩
                        dJ;
        }
    }
}

// assemble N1_1(u,v) = \int { (vel_k*\grad{u})*v } / eps
// loop over variables of test functions and ansatz ↩
    functions
for ( int u_var = 0; u_var < DIMENSION; ++u_var )
{
    // loop over test functions
    for ( int i = 0; i < num_dofs ( u_var ); ++i )
    {
        // loop over ansatz functions
        for ( int j = 0; j < num_dofs ( u_var ); ++j )
        {
            // compute integral
            lm ( dof_index ( i, u_var ), dof_index ( j, ↩
                u_var ) ) +=
                    wq * ( inveps * inveps * dot ( vel_k, ↩
                        grad_phi ( j, q, u_var ) )
                    * phi ( i, q, u_var ) ) * dJ;
        }
    }
}

// assemble N1_2(u,v) = \int { (u\grad{u_k}*v } / eps^2
// loop over variables of test functions
for ( int test_var = 0; test_var < DIMENSION; ++test_var )
{
    // loop over variables of ansatz functions
    for ( int trial_var = 0; trial_var < DIMENSION; ++↩
        trial_var )
    {
        // loop over test functions
        for ( int i = 0; i < num_dofs ( test_var ); ++i )
        {
            // loop over ansatz functions
            for ( int j = 0; j < num_dofs ( trial_var ); ↩
                ++j )
            {
                // compute integral
                lm ( dof_index ( i, test_var ), dof_index ↩
                    ( j, trial_var ) ) +=
                        wq * ( inveps * inveps * ↩
                            grad_prev_vel_[test_var][q][↩
                            trial_var]
                        * phi ( j, q, trial_var )
                        * phi ( i, q, test_var ) ) * dJ;
            }
        }
    }
}

// assemble N2(u,v) =
// \int{C * norm_ * v} * 1.75/(sqrt(150) * sqrt(kappa) * ↩
    eps^(3/2))
// loop over variables of test functions and ansatz ↩
    functions
for ( int u_var = 0; u_var < DIMENSION; ++u_var )
{
    // loop over test functions
    for ( int i = 0; i < num_dofs ( u_var ); ++i )
    {
        // loop over ansatz functions
        for ( int j = 0; j < num_dofs ( u_var ); ++j )
```

```
                    {
                        // compute integral
                        lm ( dof_index ( i, u_var ), dof_index ( j, ↩
                            u_var ) ) +=
                                wq * ( inv_darcy_root * 1.75 / ( sqrt ↩
                                    ( 150 ) ) * poweps * norm_
                                * phi ( i, q, u_var ) ) * dJ;
                    }
                }
            }
        }
    }
```

# 4 Program Output

## 4.1 Sequential Mode

Executing the program sequentially by typing **./porous_media** generates following output data:

- Mesh/Geometry data:
  - `mesh_local.pvtu` Global mesh
  - `mesh_local_0.vtu` Global mesh owned by process 0 containing the mesh information

- Solution data:

  Since it is only possible to visualize data of polynomial degree 1 using the vtk-format, the information of the degrees of freedom of higher order are lost.
  - `porous_media_solution_stationary.vtu` Solution of the velocity field and the pressure variable (vtk-format).

- Log files:
  - `porous_media_debug_log` is a list of errors helping to simplify the debugging process. This file is empty if the program runs without errors.
  - `porous_media_info_log` is a list of parameters and some helpful informations to control the program, for example information about the residual of the linear and non-linear solver.

- Flow profiles:

  If the command `compute_flowprofile()` (or `compute_flowprofile2()`) is included within the routine `solve()` (`porous_media.cc` line 158 and 160), the flow profiles of the resulting solution are evaluated at 100 equidistant evaluation points, ranging from the border of the geometry to it's center. Note that the utilized functions are *not* yet optimized and may significantly increase the required time, in particular if the computations are carried out on a coarse mesh.
  - `porous_media_flow_profile` is the evaluation of the resulting flow profile at $x = 4$.
  - `porous_media_flow_profile2` is the evaluation of the resulting flow profile at $x = 3$.

  These files can be visualized utilizing the provided Gnumeric [$G^+$] file `visualize.gnumeric`.

## 4.2 Visualization of the Solution

HiFlow$^3$ only generates output data, see section 3.3.6, but does not visualize. The mesh/geometry data as well as the solution data can be visualized by any external program which can handle the `vtk` data format as e.g. the program ParaView[Squ07].

# 5 Examples

## 5.1 Example: Porous Flow in a Chromatographic Column

The first example to be studied is the simulation of porous flow in a channel with barrier.

### 5.1.1 Configuration File

The configuration file used for the problem described in this tutorial (`porous_media.xml`).

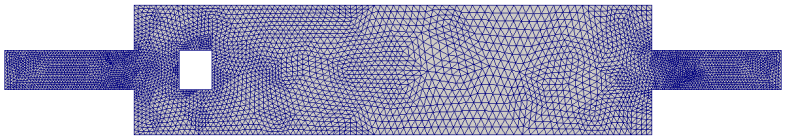
```
<Param>
  <Application>
    <Dimension>2</Dimension>
  </Application>
  <LinearAlgebra>
    <Platform>CPU</Platform>
    <Implementation>OPENMP</Implementation>
    <MatrixFormat>CSR</MatrixFormat>
    <NameMatrix>CoupledMatrix</NameMatrix>
    <NameVector>CoupledVector</NameVector>
  </LinearAlgebra>
  <FlowModel>
    <InflowDiameter>1.</InflowDiameter>
    <Type>Column</Type>
    <Density>1000.0</Density>
    <Viscosity>1.0e-3</Viscosity>
    <InflowSpeed>.2</InflowSpeed>
    <HeightRef>1.</HeightRef>
    <PorosityFree>.99</PorosityFree>
    <PermeabilityFree>1.0e4</PermeabilityFree>
    <PorosityPor>.38</PorosityPor>
    <PermeabilityPor>5.0e-3</PermeabilityPor>
    <ContiWeight>1.</ContiWeight>
  </FlowModel>
  <Mesh>
    <Column>porous2d_barrier.inp</Column>
    <Channel>porous2d_channel.inp</Channel>
    <InitialRefLevel>1</InitialRefLevel>
  </Mesh>
  <FiniteElements>
    <VelocityDegree>2</VelocityDegree>
    <PressureDegree>1</PressureDegree>
  </FiniteElements>
  <LinearSolver>
    <Name>GMRES</Name>
    <Method>NoPreconditioning</Method>
    <MaximumIterations>3000</MaximumIterations>
    <AbsoluteTolerance>1.e-12</AbsoluteTolerance>
    <RelativeTolerance>1.e-6</RelativeTolerance>
    <DivergenceLimit>1.e6</DivergenceLimit>
    <BasisSize>50</BasisSize>
  </LinearSolver>
  <Instationary>
    <SolveInstationary>0</SolveInstationary>
    <Method>CrankNicolson</Method>
    <Timestep>0.1</Timestep>
    <Endtime>10.0</Endtime>
  </Instationary>
  <Boundary>
    <InflowMaterial>10</InflowMaterial>
    <OutflowMaterial>12</OutflowMaterial>
```

```
  </Boundary>
  <NonlinearSolver>
    <Name>Newton</Name>
    <MaximumIterations>60</MaximumIterations>
    <AbsoluteTolerance>1.e-10</AbsoluteTolerance>
    <RelativeTolerance>1.e-9</RelativeTolerance>
    <DivergenceLimit>1.e6</DivergenceLimit>
  </NonlinearSolver>
  <Output>
    <SolutionFilename>porous_media</SolutionFilename>
  </Output>
  <UsePressureFilter>0</UsePressureFilter>
</Param>
```

*Remark:* The utilized flow conditions `PorosityFree` $= 0.99$ and `Permeability Free` $= 1.0 \cdot 10^4$ are assumed to approximate free flow conditions.

### 5.1.2 Parameter Distributions and Boundary Conditions

The parameter distribution of porosity ($0 < \varepsilon < 1$) and permeability ($0 < \kappa$) can be selected arbitrary on the domain, as piecewise constant behavior of both parameters is enforced during the assembly of the algebraic objects (see section 3.3.7 and 3.3.8). The implementation results in applied free flow condition at both the tubing at the in- and outflow. These areas are shown in red in figure 2b. The "porous" subdomain is visualized by the color blue.



(a) Refined 2-dimensional mesh used for the example.

(b) Distribution of areas with free- (red) and porous-flow conditions (blue) applied.

Figure 2: General setup of the mesh and parameter-domain values used for the example.

As free flow conditions are assumed at the inflow $\Gamma_{\texttt{in}}$, a parabolic Poiseuille[3]

[3] Jean Louis Lonard Marie Poiseuille, ($*$ 1797, † 1869)

profile

$$\begin{pmatrix} u_1 \\ u_2 \end{pmatrix} = \begin{pmatrix} -u_{max} \frac{(y-y_m)^2 - R_{in}^2}{R_{in}^2} \\ 0 \end{pmatrix} \tag{11}$$

is applied, where $y_m = 0.5\,m$ is the center of the inflow and $R_{in} = 0.15\,m$ it's radius. The maximal inflow velocity is selected as $u_{max} = 0.2\frac{m}{s^2}$.

### 5.1.3 Visualization of the Solution

The computed solution to the presented problem can be seen in figure 4. The two single velocity components $u_1$ and $u_2$ as well as the pressure field $p$ of the solution are visualized separately. As stated above, the computations were carried out using non-dimensional scales (cf. [Blu12, sec. 3.6]): The velocity components are scaled using the average velocity $\bar{u}_{col}$ within the column, which equals $\frac{1}{5}u_{max}$ for the posed geometry. The pressure field is scaled using $\varrho_f \bar{u}_{col}^2$. The height of the main column, which will be used as the reference length is set to be $1\,m$.

Another possible way to visualize the solution is using streamlines. This can e.g. be done in ParaView, using the *StreamTracer* filter. Note that the provided solution file does not yet contain the required vectorial depiction of the solution required for the application of the filter. Such a representation can e.g. be achieved using the provided *Calculator* filter and the command "`u*iHat+v*jHat`".
The corresponding visualization of the solution's velocity field (fig. 4a and 4b) using streamlines can be seen in figure 3.

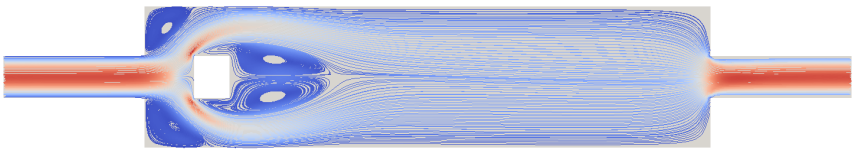


Figure 3: Visualization of the velocity field of the 2-dimensional solution using streamlines.

## 5.2 Example: Wall Effect

One of the main advantages of the GPME is its ability to account complex flow behavior while only minor accommodations are need to be done to the presented program. These changes regard the distribution of porosity and permeability. Simply including the three lines of code below //Wall Effect for both porosity


(a) Velocity component in x-direction: $u_1$.


(b) Velocity component in y-direction: $u_2$.


(c) Pressure $p$.

Figure 4: Solution of the GPME for the provided example.

(`porous_media.h`: line 82) and permeability (`porous_media.h`: line 135), results in the solution displayed in figure 6 (the solution computed without accounting for the wall effect has been added as direct comparison). Figure 5 shows the resulting distributions, which is set to be scaled identical for both parameters. The impact of the wall effect on the resulting solution of the GPME is even more apparent for the geometry of a simple channel. The solution computed for constant porosity and permeability on the entire domain is displayed in figure 7. The corresponding solution accounting for the wall effect can be seen in figure 8.

### 5.3 3-dimensional Problem

As stated at the beginning of section 3, computations can also be carried out for problems including 3 spatial dimensions by assigning the value 3 instead

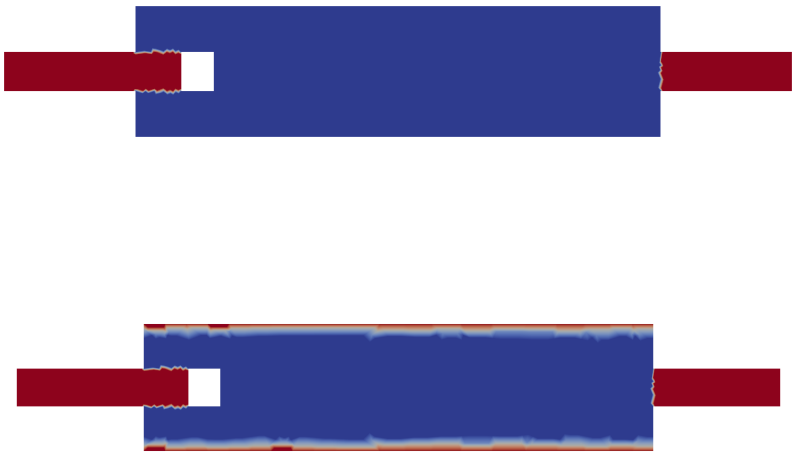


Figure 5: Parameter distribution applied for evaluating the influence of the wall effect on the 2-dimensional column model.

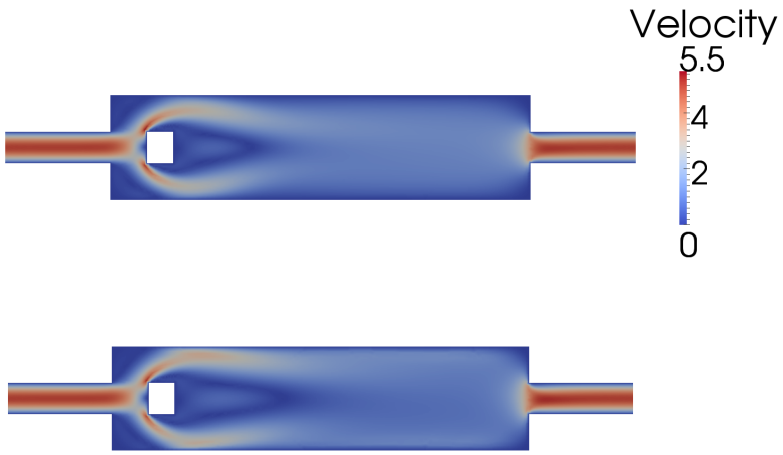



Figure 6: Solutions of the GPME evaluated for the 2-dimensional column model with (bottom) and without (top) wall effect. Parameters selected as seen in the previous example .

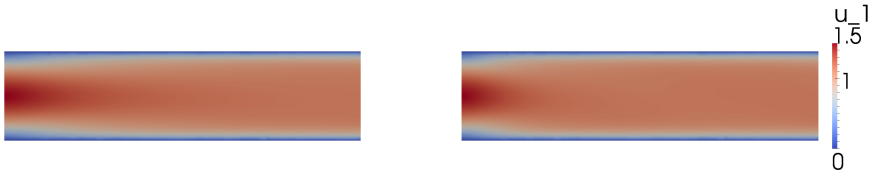



Figure 7: $u_1$ component of the solution of the GPME computed for a channel with constant porosity and permeability.


(a) $u_1$ component of the solution.

(b) Distribution of porosity and permeability. Areas of free flow are displayed in "red", areas of porous flow in "blue"

Figure 8: Solution of the GPME for channel flow accounting for the wall effect. The same parameters used for the computations leading to the solution in figure 7 have been utilized.

of 2 for the variable `DIMENSION` within the `porous_media.h` file and compiling the program. Note that a separate configuration file `porous_media3d.xml` is provided for this case.
The default geometry used for 3 dimensional computations is the body of rotation of the geometry shown in figure 1. The corresponding mesh, as visualized in figure 9, is provided in the file `porous3d_barrier.inp`.

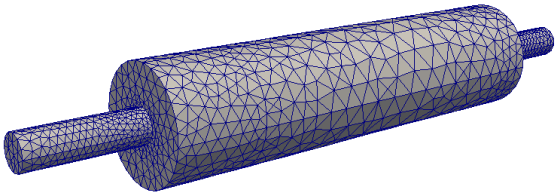


Figure 9: Default unrefined mesh for the 3-dimensional simulation (4534 nodes)

A solution of the 3-dimensional problem evaluated for the default parameter selection provided in the configuration file can be seen in figure 10. Again, ParaView's *StreamTracer* filter has been used to vividly visualize the solution. (As 3 velocity components need to be regarded, the command used within the *Calculator* filter (see 5.1.3) needs so be adapted accordingly: `u*iHat+v*jHat+w*kHat`).

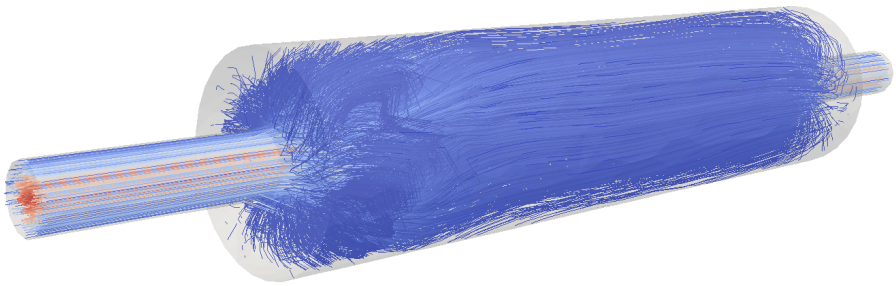


Figure 10: Visualization of the solution of the 3-dimensional problem using streamlines.

# 6 Optimization regarding Convergence

Most often, the posed program will produce valid solutions to the GPME. There are however certain situations in which the solutions behavior and/or the convergence of the algorithm is impaired:

One of the major problems faced during the evaluation of the GPME, in particular for scaled problems, is the weighting of the equation of continuity. The norm of the error regarding the conservation of mass (1b) and the conservation of momentum (1a) is minimized simultaneously, which can result in the emerging of local minima and slow down or prevent global convergence. Hence, the weighting of the single equations needs to be adapted, depending on the scales of the single contributions. This can be done using the parameter `Conti Weight` within the parameter file. Unfortunately, the order of magnitude of the single terms depends not only on the permeability or the porosity, but multiple parameters. It is thus advisable, in particular prior to evaluating the GPME for complex 3-dimensional problems, to test convergence behavior on a simpler domain for the given parameter selection.
One major improvement of the program regarding operability and scalability would be the automatic adaption of said weight depending on the selected parameters.

A second problem is the emerging of unexpected solution-behavior near the outflow boundary and at boundary transition, where the fluid tends to channel next to a boundary instead of transitioning uniformly along the cross-section.
As this kind of behavior usually evolves within the first few steps of the solving algorithm, the effect might quite possibly be connected to the inaccurate derivation and evaluation of the non-dimensional drag term. An exact derivation however would forfeit the sparsity structure of the system matrix and thus greatly increase the required computational expense.

# References


[BHSK$^+$] M. Baumann, A. Helfrich-Schkarbanenko, E. Katelaer, S. Ronnås, and M. Wlotzka, *Boundary value problem for incompressible navier-stokes equation*, HiFlow$^3$ Tutorial.

[Blu12] U. Blunck, *Numerical simulation of high-performance liquid chromatography using the generalized porous media equation*, Diploma Thesis, Karlsruhe Institute of Technology, 2012.

[G$^+$] A. J. Guelzow et al., *The gnumeric manual, version 1.10*, http://projects.gnome.org/gnumeric/doc/gnumeric-info.shtml.

[GA94] R. C. Givler and S. A. Altobelli, *A determination of the effective viscosity for the brinkman–forchheimer flow model*, Journal of Fluid Mechanics (1994), Volume 258, no. 1, 355–370.

[McL00] W. McLean, *Strongly elliptic systems and boundary integral equations*, Cambridge University Press, 2000.

[Squ07] A. H. Squillacote, *The paraview guide: a parallel visualization application*, Kitware [Clifton Park, NY], 2007.