Technical university of Liberec

Faculty of mechatronics, informatics
and interdisciplinary studies

FLOW123D - draft, scheme of new inputs

version 1.7.0

**Documentation of file formats
and brief user manual.**

Liberec, 2011

Acknowledgement. This work was realized under the state subsidy of the Czech Republic within the research and development project "Advanced Remediation Technologies and Processes Center" 1M0554 – Program of Research Centers PP2-D01 supported by Ministry of Education.

Contents

0.1

Chapter 1

Quick start

Flow123D is a software for simulation of water flow and reactionary solute transport in a heterogeneous porous and fractured medium. In particular it is suited for simulation of underground processes in a granite rock massive. The program is able to describe explicitly processes in 3D medium, 2D fractures, and 1D chanels and exchange between domains of different dimension. The computational mesh is therefore collection of 3D tetrahedrons, 2D triangles and 1D line segments.

The water flow model assumes a saturated medium described by Darcy law. For discretization, we use lumped mixed-hybrid finite element method. We support both steady and unsteady water flow.

The solute transport model can deal with several dissolved substances. It contains non-equilibrium dual porosity model, i.e. exchange between mobile and immobile pores. There is also model for several types of adsorption in both the mobile and immobile zone. The implemented adsorption models are linear adsorption, Freundlich isotherm and Langmuir isotherm. The solute transport model uses finite volume discretization with up-winding in space and explicit Euler discretization in time. The dual porosity and the adsorption are introduced into transport by operator splitting. The dual porosity model use analytic solution and the non-linear adsorption is solved numerically by the Newton method.

Reaction between transported substances can be modeled either by a SEMCHEM module, which is slow, but can describe all sorts of reactions. On the other hand, for reactions of the first order, i.e. linear reactions or decays, we provide our own solver which is much faster. Reactions are coupled with transport by the operator splitting method,

The program provides output of the pressure, the velocity and the concentration fields in two file formats. You can use file format of GMSH mesh generator and post-processor or you can use output into widely supported VTK format. In particular we recommend Paraview software for visualization and post-processing of VTK data.

The program is implemented in C/C++ using essentially PETSC library for linear algebra. The water flow as well as the transport simulation and reactions can be computed in parallel using MPI environment.

The program is distributed under GNU GPL v. 3 license and is available on the project web page: http://dev.nti.tul.cz/trac/flow123d

1.1 Basic usage

1.1.1 How to run the simulation.

On the Linux system the program can be started either directly or through a script `flow123d.sh`. When started directly, e.g. by the command

```
> flow123d -s example.ini
```

the program requires one argument after switch `-s` which is the name of the principal input file. Full list of possible command line arguments is as follows.

-s *file*
: Set principal INI input file. All relative paths in the INI file are relative against current directory.

-S *file*
: Set principal INI input file. All relative paths in the INI file are relative against directory of the INI file. This is equivalent to change directory to the directory of the INI file at the start of the program.

-i *path*
: When there is string `${INPUT}` in the any path in the INI file, it will be replaced by given *path*.

-o *path*
: Every relative path for any output file will be relative to this *path*.

-l *[file_name]*
: Set base name of log files or turn logging off if no file name is given.

All other parameters will be passed to the PETSC library. An advanced user can influence lot of parameters of linear solver. In order to get list of supported options use parameter `-help`.

Alternatively, you can use script `flow123d.sh` to start parallel jobs or limit resources used by the program. This script accepts the same parameters as the program itself and further following additional parameters:

-h
: Usage overview.

-t *timeout*
: Upper estimate for real running time of the calculation. Kill calculation after *timeout* seconds. Can also be used by PBS to choose appropriate job queue.

-np *number of processes*
: Specify number of parallel processes for calculation.

-m *memory limit*
: Limits total available memory to *memory limit* bytes.

-n *priority*
Change (lower) priority for the calculation. See `nice` command.

-r *out file*
Stdout and stderr will be redirected to *out file*.

On the Windows system we use Cygwin libraries in other to emulate Linux API. Therefore you have to keep the Cygwin libraries within the same direcotry as the program executable. The Windows package that can be downloaded from project web page contains both the Cygwin libraries and the mpiexec command for starting parallel jobs on the Windows workstations.

Then you can start the sequential run by the command:

```
> flow123d.exe -s example.ini
```

or the parallel run by the command:

```
> mpiexec.exe -np 2 flow123d.exe -s example.ini
```

The program accepts the same parameters as the Linux version, but there is no script similar to `flow123d.sh` for the Windows system.

1.1.2 Structure of input

The principal input file of the program is an INI file which contains names of other necessary input files. Those are the file with calculation mesh (`*.msh`), the file with specification of adjacency between dimensions (`*.ngh`), the file with material description (`*.mtr`) and the file with boundary conditions for the water flow problem (`*.bcd`). For unsteady water flow you have to specify file with initial condition for the pressure (key `Input/Initial`) and optionally one can introduce file with water sources (key `Input/Sources`).

In the case of transport simulation one have to specify also the file with transport boundary conditions (`*.tbc`) and the file with transport initial condition for individual substances (`*.tic`).

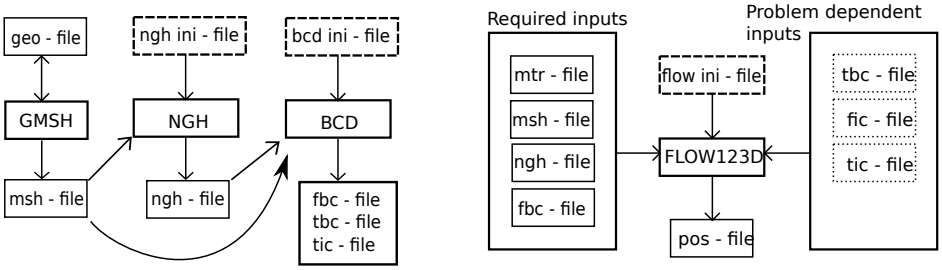

Figure 1.1: Preparation of input files.

For the preparation of input files we use several utilities (see Figure 1.1). We usually begin with a `*.geo` file as a description of the domain geometry. This come as an input

for the GMSH mesh generator, which produce the mesh file. Then we run program `ngh` to produce adjacency file. Finally we can use program `bcd` for the preparation of files with boundary and initial conditions. The file with material properties has to be created manually, preferably by modifying some of the example problems. The programs `ngh` and `bcd` are distributed together with flow123d with their own limited documentation.

The output files can be either `*.msh` files accepted by the GMSH or one can use VTK format that can be post-processed by Paraview.

In the following chapter, we briefly describe structure of individual input files in particular the main INI file. In the last chapter, we describe mathematical models and numerical methods used in the Flow123d.

Chapter 2

Mathematical models of physical reality

Chapter 3

Numerical methods

3.0.3 Advection-Diffusion equation

Solute transport is governed by advection equation which can be written in the form

$$\frac{\partial c}{\partial t} + \boldsymbol{v}\frac{\partial c}{\partial x} = 0, \tag{3.1}$$

where c is concentration $[M^3 \cdot L^{-3}]$, t is time $[T]$, v is velocity $[L \cdot T^{-1}]$, and x is coordinate in cartesian system $[L]$. Assuming solution which is constant on every element (cell centered finite volume method) and integrating equation (3.1) we get

$$\int_{e_i} \frac{\partial c}{\partial t} dV + \int_{e_i} \boldsymbol{v}\frac{\partial c}{\partial x}\, dV = 0.$$

After some rearrangements we obtain on i-th element (e_i)

$$\frac{\partial c_i}{\partial t} V_i + c \int_{\partial e_i} \boldsymbol{v}\, \mathbf{dS} = 0, \tag{3.2}$$

where c_i is average concentration in e_i and V_i its volume, c will be specified later (there are two main possibilities - c_i or concentration from neighbouring element). Term $\frac{\partial c}{\partial t}$ we approximate by explicit Euler difference

$$\frac{\partial c}{\partial \mathrm{t}} \approx \frac{c_i^{n+1} - c_i^n}{\Delta t}. \tag{3.3}$$

Where Δt is a time step and upper index at c_i means values in the discrete time steps $n+1$ and n. We assume that all elements have piecewise smooth element boudary ∂e with outwards directed normal. Inside the area Ω we introduce internal flows. With respect to e_i, we define internal flow intake U_{ij}^- (from element e_j) and internal flow drain U_{ij}^+ (to element e_j) as follows

$$\begin{aligned} U_{ij}^- &= \min\Big(\int_{\partial e_i \cap \partial e_j, i \neq j} \boldsymbol{v}\, \mathbf{dS}, 0\Big), \\ U_{ij}^+ &= \max\Big(\int_{\partial e_i \cap \partial e_j, i \neq j} \boldsymbol{v}\, \mathbf{dS}, 0\Big). \end{aligned} \tag{3.4}$$

Those flows realizes solute transport in the area Ω. On the $\partial\Omega$ we define external flows which will be important for transport Dirichlet boundary conditions. In the same way as for internal flows we assume (with respect to element e_i) external flow intake U_{ij}^{e-} (from $\partial\Omega$) and external flow drain U_{ij}^{e+} (to $\partial\Omega$).

$$\begin{aligned} U_{ik}^{e-} &= \min(\int\limits_{\partial e_i \cap \partial\Omega} \boldsymbol{v}\,\mathbf{dS}, 0), \\ U_{ik}^{e+} &= \max(\int\limits_{\partial e_i \cap \partial\Omega} \boldsymbol{v}\,\mathbf{dS}, 0). \end{aligned} \tag{3.5}$$

Direction of the velocity $\boldsymbol{v}$, which affects sign of the U-terms is significant for the construction solution. For the solution stability it is suitable to use an upwind scheme, which can by written for finite difference on simple 1D geometry in the form

$$\begin{aligned} v > 0 : \frac{\partial c}{\partial \mathrm{x}} &\approx \frac{c_i^n - c_{i-1}^n}{\Delta x}, \\ v < 0 : \frac{\partial c}{\partial \mathrm{x}} &\approx \frac{c_{i+1}^n - c_i^n}{\Delta x}. \end{aligned} \tag{3.6}$$

This scheme can be interpreted as well as in finite volume method - in convection term one can get c value opposite the flow of the quantity $\boldsymbol{v}$ direction. For every e_i we introduce itemsets $\mathcal{N}_i, \mathcal{B}_i$ which contains indexes of neighbourging elements, local boundary conditions respectivelly. Assuming upwind scheme, using (3.4), (3.5), and (3.3) we can write solution of the equation (3.2) (relation between two consecutive time steps) on e_i in the form

$$c_i^{n+1} = c_i^n - \frac{\Delta t}{V_i}\left[\sum_{j\in\mathcal{N}_i}\left[U_{ij}^{+}c_i + U_{ij}^{-}c_j\right] + \sum_{k\in\mathcal{B}_i}\left[U_{ik}^{e+}c_i + U_{ik}^{e-}c_{B_{ik}}\right]\right]. \tag{3.7}$$

Where $c_{B_{ik}}$ are values of Dirichlet boundary conditions which belong to e_i. Formula (3.7) can be rewritten into the matrix notation

$$\mathbf{c}^{n+1} = (\mathbf{I} + \Delta t\mathbf{A})\cdot\mathbf{c}^n + \Delta t\mathbf{B}\cdot{\mathbf{c_B}}^n \tag{3.8}$$

Where $\mathbf{c}$ is vector of c_i^{n+1}, $\mathbf{A}$ is a square matrix composed from $\frac{U_{ij}^{+}}{V_i}$, $\frac{U_{ij}^{-}}{V_i}$, and $\frac{U_{ij}^{e+}}{V_i}$. $\mathbf{B}$ is in general rectangular matrix composed from $\frac{U_{ij}^{e-}}{V_i}$ and ${\mathbf{c_B}}^n$ is vector of Dirichlet boundary conditions.matrix definition. There is one stability condition for time step which is called Courant-Friedrich-Levy condition. For the problem without sources/sinks it can be written as

$$\Delta t_{max} = \min_i\left(\frac{V_i}{\sum_j U_{ij}^{+} + \sum_k U_{ik}^{e+}}\right) = \min_i\left(\frac{V_i}{\sum_j |U_{ij}^{-}| + \sum_k |U_{ik}^{e-}|}\right). \tag{3.9}$$

This condition has a physical interpretation, which can be understood as conservation law - volume that intakes/drains to/from element e_i can not be higher then element volume V_i. From algebraical point of view this condition can be seen as a condition which bounds norm of the evolution operator as follows

$$\|\mathbf{I} + \Delta t\mathbf{A} \quad \Delta t\mathbf{B}\| \leq 1. \tag{3.10}$$

3.0.4 Generalization

This approach can be used as well as for more general element connections – for compatible/non-compatible element interconnection, if we know the flow integral values (U_{ij}^{+} or U_{ij}^{-}). The most general case of connection is relation among n elements like

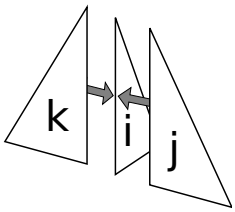

Figure 3.1: Edge with 3 elements

in figure (3.1). For this case we define edge element indexset $\mathcal{G}_l$ that contains all the indexes of elements which sides make l-th edge (g_l), so that $\mathcal{G}_l = \{i, j, k\}$. For $\mathcal{G}_l$ we introduce its subsets $\mathcal{G}_{ij}$, $\mathcal{G}_{ji}$, $\mathcal{G}_{ik}$, $\mathcal{G}_{ki}$, $\mathcal{G}_{kj}$, and $\mathcal{G}_{jk}$, where $\mathcal{G}_{ij} = \mathcal{G}_{ik} = \mathcal{G}_l \backslash i = \{j, k\}$, $\mathcal{G}_{ji} = \mathcal{G}_{jk} = \mathcal{G}_l \backslash j = \{i, k\}$, and $\mathcal{G}_{ki} = \mathcal{G}_{kj} = \mathcal{G}_l \backslash k = \{i, j\}$. It can be written in the same way for any edge g with more than 3 elements, it is hold $|\mathcal{G}_g| - 1 = |\mathcal{G}_{ab}|; \forall a, b \in \mathcal{G}_g$. For l-th edge (g_l) we can define total edge flow U_{g_l} eg. as

$$\begin{aligned} U_{g_l} &= \sum_{m \in \mathcal{G}_{ji}} \left[U_{mj}^{+} + \frac{U_{jm}^{+}}{|\mathcal{G}_{ji}|} \right] = \sum_{m \in \mathcal{G}_{jk}} \left[U_{mj}^{+} + \frac{U_{jm}^{+}}{|\mathcal{G}_{jk}|} \right] \\ &= \sum_{m \in \mathcal{G}_{ij}} \left[U_{mi}^{+} + \frac{U_{im}^{+}}{|\mathcal{G}_{ij}|} \right] = \sum_{m \in \mathcal{G}_{ik}} \left[U_{mi}^{+} + \frac{U_{im}^{+}}{|\mathcal{G}_{ik}|} \right] \\ &= \sum_{m \in \mathcal{G}_{ki}} \left[U_{mk}^{+} + \frac{U_{km}^{+}}{|\mathcal{G}_{ki}|} \right] = \sum_{m \in \mathcal{G}_{kj}} \left[U_{mk}^{+} + \frac{U_{km}^{+}}{|\mathcal{G}_{kj}|} \right], \end{aligned} \tag{3.11}$$

U_{g_l} with respect to any e_m; $m \in \mathcal{G}_l$ has to have the same value because continuity equation, for assumed incompresible flow, has to be fulfilled in every edge. Edges with more than two elements and two and more nonzero intakes to edge realize an ideal mixing (to an average concentration) with weights which will be specified later. This fact modifies equation (3.7) on the general mesh into the form

$$c_i^{n+1} = c_i^n - \frac{\Delta t}{V_i} \left[\sum_{j \in \mathcal{N}_i} \left[U_{ij}^{+} c_i + \frac{U_{ij}^{-}}{\sum\limits_{k \in \mathcal{G}_{ij}} \left[U_{ki}^{+} + \frac{U_{ik}^{+}}{|\mathcal{G}_{ij}|} \right]} \sum_{k \in \mathcal{G}_{ij}} U_{ki}^{+} c_k \right] + \sum_{k \in \mathcal{B}_i} \left[U_{ik}^{e+} c_i + U_{ik}^{e-} c_{B_{ik}} \right] \right]. \tag{3.12}$$

The edges with total edge flow $U_{g_l} = 0$ can occur breakdown in the equation (3.12) via term $\sum\limits_{k \in \mathcal{G}_{ij}} \left[U_{ki}^{+} + \frac{U_{ik}^{+}}{|\mathcal{G}_{ij}|} \right] = 0$. This fact implies as well as numerator $U_{ij}^{-} = 0$. In order to avoid dividing by zero we have to assume computation only for nonzero flows. Concentrations c_k, $k \in \mathcal{G}_{ij}$ that may intakes into element e_i are weighted with weights

$$\alpha_k = \frac{U_{ki}^{+}}{\sum\limits_{k \in \mathcal{G}_{ij}} \left[U_{ki}^{+} + \frac{U_{ik}^{+}}{|\mathcal{G}_{ij}|} \right]}, \tag{3.13}$$

so that the ideal mixing in this edge leads to the average concentration

$$c_{av} = \frac{\sum\limits_{k\in\mathcal{G}_{ij}} U^{+}_{ki} c_k}{\sum\limits_{k\in\mathcal{G}_{ij}} \left[U^{+}_{ki} + \frac{U^{+}_{ik}}{|\mathcal{G}_{ij}|}\right]}. \tag{3.14}$$

Matrix notation is the same as in (3.8). Finally ...

3.1 Radioactive Decay and First Order Reactions

Lets consider to have a narrow decay chain without branches. This kind of decay chain can be described by following equation

$$A \xrightarrow{t_{1/2,A}} B \xrightarrow{t_{1/2,B}} C \xrightarrow{t_{1/2,C}} D \xrightarrow{t_{1/2,D}} E,$$

where letters $\{A,\ldots,E\}$ denotes isotopes contained in considered decay chain and $t_{1/2}, i,\ i \in \{A,\ldots,E\}$ is a symbol for a half-life of i-th isotope. For a simulation of radioactive decay and first order reactions matrix multiplication based approach has been developed. It has been based on an arrangement of all the data to matrices. The matrix $\mathbf{C}^k$ contains the information about concentrations of all species (s) in all observed elements (e). The upper index k denotes k-th time step. The matrix $\mathbf{C}^k$ has the dimension $e \times s$ ($rows \times columns$). The transport simulation is realized by matrix multiplication

$$\mathbf{T}\cdot\mathbf{C}^k = \mathbf{C}^{k+1},$$

where $\mathbf{T}$ is a square, block-diagonal matrix, representing a set of algebraic equations constructed from a set of partial differential equations. When the simulation of the radioactive decay or the first order reaction is switched on, one step of simulation changes to

$$\mathbf{T}\cdot\mathbf{C}^k\cdot\mathbf{R} = \mathbf{C}^{k+1},$$

where $\mathbf{R}$ is a square matrix with the dimension $(s \times s)$. It is much easier to construct and to use $\mathbf{R}$, than to include chemical influence to the transport matrix $\mathbf{T}$, because the matrix $\mathbf{R}$ has usually a simple structure and s is much smaller than e. In the most simple case, when the order of identification numbers of isotopes in considered decay chain is the same as the order of identifiers of species transported by groundwater, then just two diagonals are engaged and the matrix R looks as follows:

$$\mathbf{R} = \begin{pmatrix}
\left(\frac{1}{2}\right)^{\frac{\Delta t}{t_{1/2,1}}} & 1-\left(\frac{1}{2}\right)^{\frac{\Delta t}{t_{1/2,i}}} & 0 & \cdots & \cdots & 0 \\
0 & \left(\frac{1}{2}\right)^{\frac{\Delta t}{t_{1/2,2}}} & 1-\left(\frac{1}{2}\right)^{\frac{\Delta t}{t_{1/2,2}}} & 0 & \ddots & 0 \\
\vdots & \ddots & \ddots & \ddots & \ddots & \vdots \\
0 & \ddots & 0 & \left(\frac{1}{2}\right)^{\frac{\Delta t}{t_{1/2,n-2}}} & 1-\left(\frac{1}{2}\right)^{\frac{\Delta t}{t_{1/2,n-2}}} & 0 \\
0 & \cdots & \cdots & 0 & \left(\frac{1}{2}\right)^{\frac{\Delta t}{t_{1/2,n-1}}} & 1-\left(\frac{1}{2}\right)^{\frac{\Delta t}{t_{1/2,n-1}}} \\
0 & \cdots & \cdots & 0 & 0 & 1
\end{pmatrix}$$

Every single j-th column, except the first one, includes the contribution $1-\left(\frac{1}{2}\right)^{\frac{\Delta t}{t_{1/2,j}}}$, $j \in \{A,\ldots,E\}$ from $(j-1)$-th isotope with its half-life $t_{1/2,j-1}$. The term $\left(\frac{1}{2}\right)^{\frac{\Delta t}{t_{1/2,j}}}$ describes concentration decrease caused by the radioactive decay of j-th isotope itself. In general

cases the matrix **R** can have much more complicated structure, especially when the considered decay chain has more branches. The implementation of the radioactive decay in Flow123D does not firmly include standard natural decay chain. Instead of that it is possible for a user to define his/her decay chain.

It is also possible to simulate decay chains with branches as picture 3.2 shows.

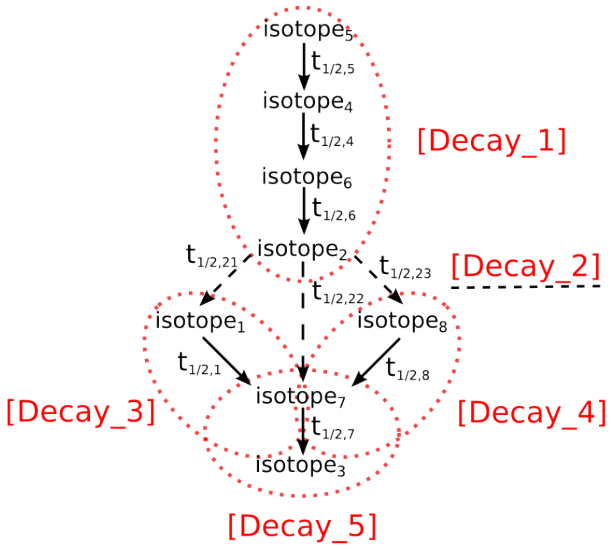

Figure 3.2: Decay chain with branches.

When it comes to a simulation of first order reactions, the kinetic constant is given as an input. The description of a kinetic chemical reaction has obviously two folowing forms

$$A \xrightarrow{k} B,$$
$$\frac{dc^A}{dt} = -k \cdot c^A.$$

The first one description is a standard chemical one. The second equation describes temporal decrease in amount of concentrations of the specie c^A. The constant k is so called kinetic constant and for the case of a first order reactions it is equal to so called reaction rate. The order of reaction with just one reactant is equal to the power of c^A in partial diferential reaction.

For an inclusion of first order reaction into a reaction matrix a half-life needs to be computed from the corresponding kinetic constant k. The derivation follows

$$\begin{aligned}
&A \xrightarrow{k} B \\
&\frac{dc^A}{d\tau} = -k \cdot c_A \\
&\frac{dc^A}{c^A} = -k \cdot d\tau \\
&\int_{c_0^A/2}^{c_0^A} \frac{dc^A}{c^A} = -k \cdot \int_{t_{1/2,A}}^{0} d\tau \\
&\left[ln c^A\right]_{c_0^A/2}^{c_0^A} = -\left[k\tau\right]_{t_{1/2,A}}^{0} \\
&ln c_0^A - \ln \frac{c_0^A}{2} = k \cdot t_{1/2,A} \\
&\ln 2 = k \cdot t_{1/2,A} \\
&t_{1/2,A} = \frac{ln2}{k}
\end{aligned}$$

The matrix **R** is constructed in the same way as for the radioactive decay.

3.2 General Chemical Reactions

For a simulation of general chemical reactions as a part of reactive transport simulation, an application Semchem has been merged together with Flow123D. It enables to simulate following types of reactions:

- Chemical equilibrium (solved using iterative Newtons method)

  mathematical description $K^{(r)} = \prod_i c_i^{\alpha_i^{(r)}}$,

- Slow evolving chemical kinetics (solved using Runge-Kutta method)

  mathematical description $\frac{dc_i}{dt} = -k^{(r)} \prod_j c_j^{\beta_j^{(r)}}$,

- Fast evolving chemical kinetics (discretized using implicit Eulerova method and solved using Newtons method)

  mathematical description $\frac{c_i^{(T+1)} - c_i^T}{\Delta t} = -k^{(r)} \prod_j c_j^{\beta_j^{(r),(T+1)}}$,

- Radioactive decay can be simulated as a special case of first order reaction.

Mathematical description of general chemical reactions contains often nonlinear terms. That brings the need to solve set of nonlinear algebraic equation in every single time step. It can be very time demanding and that is the reason to look for ways how to reduce number of variables and nonlinear equations somehow. Semchem uses transformation of variables into so called 'reaction rate' (ζ^r). 'Reaction rate' is an molar concentration (m_i $[mol/m^3]$) of an exhausted reactant devided by appropriate stechiometric coeficient a_i [1].

$$[m_i] = \frac{[n_i]}{[V]} = \frac{[m]}{[M_m][V]} = \frac{[\rho][V]}{[M_m][V]} = \frac{[\rho]}{[M_m]} \cdot [c_i],$$

where m without subscription denotes weight.

Thanks to this approach it is possible to decrease the number of variables to the number of considered chemical reactions.

Further is described how to decrease number of variables in descriptions of kinetic reactions (3.15) and chemical equilibrium (3.21).

$$\frac{dm_j}{dt} = \sum_r k^{(r)} a_j^{(r)} \prod_i m_i^{a_i^{(r)}} \tag{3.15}$$

Using implicit time discretisation of (3.15) we get

$$m_j^{T+1} = m_j^T + \Delta t \sum_r a_j^{(r)} k^{(r)} \prod_i (m_i^{T+1})^{a_i^{(r)}}. \tag{3.16}$$

Through replacement in (3.16) we obtain an equation (3.17).

$$m_j^{T+1} = m_j^T + \sum_r a_j^{(r)} \zeta^{(r)} \tag{3.17}$$

By a substitution of (3.17) into (3.16), an equation (3.18) is created.

$$m_j^T + \sum_s a_j^{(s)} \zeta^{(s)} = m_j^T + \Delta t \sum_r a_j^{(r)} k^{(r)} \prod_i \left(m_i^T + \sum_R a_i^{(R)} \zeta^{(R)} \right)^{a_i^{(r)}} \tag{3.18}$$

Substraction of m_j^T from both sides of previous equation leads to a set of nonlinear equations (3.19) with just as many variables as considered chemical reactions identified by indeces (r).

$$\zeta^{(s)} = \Delta t \cdot k^{(r)} \prod_i \left(m_i^T + \sum_R a_i^{(R)} \zeta^{(R)} \right)^{a_i^{(r)}} \tag{3.19}$$

(3.19) can be modificated to (3.20)

$$\zeta^{(s)} \prod_i \left(m_i^T + \sum_r a_i^{(r)} \zeta^{(r)} \right)^{-a_i^{(r)}} = \Delta t \cdot k^{(r)} \tag{3.20}$$

which needs to be solved.

Consider to have N species taking part in chemical reactions. For chemical equilibriums the equation (3.21) is obviously used for description. Letter X_i denotes an activity of i-th chemical specie.

$$K^{(r)} = \prod_{i=1}^{N} X_i^{(r)} \tag{3.21}$$

Chapter 4

File formats

4.1 Input files

In this section we shall describe whole structure of the program input, namely the structure of the root input file. File formats of other files used for input of the mesh or large field data (e.g. the GMSH file format) are described in separate sections. The program input consists of the root input file given as the parameter on the command line and possibly several other files with large input data. The root input file is in so called CON file format that is a slight extension of the JSON file format.

In this section we shall describe format of the input files. At first, we specify syntax of an extension to the JSON file format. Then we set rules for input of more specific data constructs. We continue by description of general scheme for input of boundary conditions and material time-space variable data. And finally, we describe setting of particular equations and their solvers.

The aim of this draft is twofold. First, we want to outline the way how to translate current input file format (in version 1.6.5) into the new one without extending the existing functionality. Second, we want to propose a new way how to input general boundary and material data. Desired features are:

- input simple data in simple way
- possibility to express very complex input data
- possibility to generate data automatically, and input very large input sets
- input interface that provides uniform access to the data in program independent of the input format

[... something else?]

As this is a draft version there are lot of remarks, suggestions and questions in square brackets. Some keys are marked OBSOLETE, which means that we want to replace them by something else.

4.1.1 CON file format

The root input file is in the Humanized JSON file format. That is the JSON file format with few syntax extensions and several semantic rules particular to Flow123d. The syntax extensions are

1. You can use one line comments using hash `#`.
2. The quoting of the keys is optional if they do not contain spaces (holds for all Flow keys).
3. You can use equality sign `=` instead of colon `:` for separation of keys and values in JSON objects.
4. You can use any whitespace to separate tokens in JSON object or JSON array.

The aim of these extensions is to simplify writing input files manually. However these extensions can be easily filtered out and converted to the generic JSON format. (This way it can be also implemented in Flow123d.)

For those who are not familiar with the JSON file format, we give the brief description right here. The full description can be found at http://www.json.org/. However, we use term *record* in the place of the *JSON object* in order to distinguish *JSON object*, which is merely a data structure written in the text, and the *C++ object*, i.e. instance of some class.

4.1.2 Humanized JSON

The JSON format consists of four kind of basic entities: *null* token, *true* and *false* tokens, *number*, *string*. *Number* is either integer or float point number possibly in the exponential form and *string* is any sequence of characters quoted in `""` (backslash `\` is used as escape character and Unicode is supported, see full specification for details).

In the following, we mean by white space characters: space, tab, and new line. In particular the newline character (outside of comment or quoting) is just the white space character without any special meaning.

The basic entities can be combined in composed entities, in a *record* or in an *array*. The *record* is set of assignments enclosed in the curly brackets

```
{
        #basic syntax
        "some_number":124,
        "some_string":"Hallo",
        "some_subrecord":{},
        "some_array":[],

        #extended syntax
        non_quoted_key_extension=123,
        separation_by_whitespace="a" sbw_1="b"
        sbw_2="c"
}
```

One assignment is a pair of the *key* and the *value*. *Key* is *string* or token matching regexp
`[a-zA-Z_][a-zA-Z_0-9]*`. *Value* is basic or composed entity. The key and the value are separated by the colon (generic syntax) or equality sign (extended syntax). Pairs are separated by a comma (generic syntax) or sequence of white space characters (extended syntax). The values stored in the record are accessed through the keys like in an associative array. Records are usually used for initialization of corresponding classes.

The second composed entity is the *array* which is sequence of (basic or composed) entities separated by comma (generic syntax) or whitespace sequence (extended syntax) and enclosed in the square brackets. The values stored in the array are accessed through the order. The Flow reader offers either initialization of a container from JSON array or a sequential access. The latter one is the only possible access for the included arrays, which we discuss later.

On any place out of the quoted string you can use hash mark `#` to start a one line comment. Everything up to the new line will be ignored and replaced by single white space.

[What about multiple line strings? (Should be allowed)]

4.1.3 CONSpecial keys

Apart from small extensions of JSON syntax, we impose further general rules on the interpretation of the input files by Flow123d reader. First, the capital only keywords have a special meaning for Flow JSON reader. On the other hand, we use only small caps for keys interpreted through the reader. The special keywords are:

TYPE :

```
TYPE= <enum>
```

The `<enum>` is particular semantic construct described later on. When appears in the record, it specifies which particular class to instantiate. This only has meaning if the record initializes an abstract class. In consistency with the source code, we shall call such records *polymorphic*.

In fact we consider that every record is of some *type* at least implicitly. The *type* of the record is specification of the keys that are interpreted by the program Flow123d. At some places the program assumes a record of specific *type* so you need not to specify `TYPE` key in those records.

INCLUDE_RECORD :

This is a simple inclusion of another file as a content of a record:

```
{
        INCLUDE_RECORD = "<file name>"
}
```

INCLUDE_ARRAY :

```
array=
{
        INCLUDE_ARRAY = "<file name>"
        FORMAT = "<format string>"
}
```

The reader will substitute the include record by a sequentially accessible array. The file has fixed number of space separated data fields on every line. Every line becomes one element in the array of type record. Every line forms a record with key names given by the `<format string>` and corresponding data taken form the line.

The key difference compared to regular JSON arrays is that included arrays can be accessed only sequentially within the program and thus we minimize reader memory overhead for large input data. The idea is to translate raw data into structured format and use uniform access to the data.

Basic syntax for format string could be an array of strings — formats of individual columns. Every format string is an address of key that is given the column. Onother possibility is to give an arbitrary JSON file, where all values are numbers of columns where to take the value.

[. . . better specify format string]

[Possible extensions: - have sections in the file for setting time dependent data - have number of lines at the beginning - have variable format - allow vectors in the 'line records']

REFERENCE :

```
time_governor={
  REF=<address>
}
```

This will set key `time_governor` to the same value as the entity specified by the address. The address is an array of strings for keys and integers for indices. The address can be absolute or relative identification of an entity. The relative address is relative to the entity in which the reference record is contained. One can use string `".."` to move to parent entity and string `"//"` to move to the root record of current file. Indices in address starts from 0.

For example assume the file

```
mesh={
        file_name="xyz"
}
array=[
        {x=1, y=0}
        {x=2, y=0}
        {x=3, y=0}
]
```

```
outer_record={
        output_file="x_out"
        inner_record={
                output_file={REF=["..","output_file"]}  # value "x_out"
        }
        x={REF="/array/2/x"}                                    # value "3"
        f_name={REF=["//","mesh","file_name"]}                  # value "xyz"
}
```

Concept of addres should be better explained and used consistently in reader interface.

4.1.4 Semantic rules

Implicit creation of composed entities

Consider that there is a *type* of record in which all keys have default values (possibly except one). Then the specification of the record *type* can contain a *default key*. Then user can use the value of the *default key* instead of the whole record. All other keys apart from the *default key* will be initialized by default values. This allows to express simple tasks by simple inputs but still make complex inputs possible. In order to make this working, developers should provide default values everywhere it is possible.

Similar functionality holds for arrays. If the user sets a non-array value where an array is expected the reader provides an array with a unique element holding the given value. See examples in the next section for application of these two rules.

Enum construct

Enum values can be integers or strings from particular set. Strings should be preferred for manual creating of input files, while the integer constants are suitable for automatic data preparation.

The input reader provides a way how to define names of members of an enum class and then initialize this enum class from input file. [Need better description]

String types

For purpose of this documentation we distinguish several string types with particular purpose and treatment. Those are:

input filename This has to be valid absolute or relative path to an existing file. The string can contain variable `${INPUT}` which will be replaced by path given at command line through parameter `-i`.

[In order to allow input of time dependent data in individual files, we should have also variable `${TIME\_LEVEL}` From user point of view this is not property of general input filename string, however in implementation this should be done in the same way as `${INPUT}`.]

[? Shall we allow both Windows and UNIX slashes?]

[Developers should provide default names to all files.]

output filename This has to be relative path. The path will be prefixed by the path given at command line through the parameter `-o`. In some cases the path will be also postfixed by extension of particular file format.

formula Expression that will be parsed and evaluated runtime. Documentation of particular key should provide variables which can appear in the expression, however in general it can be function of the space coordinates x, y, z and possibly also function of time t. For full specification of expression syntax see documentation of FParser library: http://warp.povusers.org/FunctionParser/fparser.html#literals

text string Just text without particular meaning.

Record types

A record type like particular definition of a class (e.g. in C++). One record type serves usually for initialization of one particular class. From this point of view one record type is set of keys that corresponding class can read.

For purpose of this manual the record type is given by specification of record's keys, their types, default values and meanings. In the next two sections, we describe all record types that forms input capabilities of Flow123d. Description of a record type has form of table. Table heading consists of the name of the record type. Then for every key we present name, type of the value, default value and text description of key meaning. Type of the value can be record type, array of record types, double, integer, enum or string type. Default value specification can be:

none No default value given, but input is mandatory. You get an error if you don't set this key.

null `null` value. No particular default value, but you need not to set the key. Usually means feature turned off.

explicit value For keys of type: string, double, integer, or enum, the default value is explicit value of this type.

type defaults For keys of some record type we let that record to set its default values.

[? polymorphic record types]

4.2 Record types for input of data fields

In this section we describe record types used to describe general time-space scalar, vector, or tensor fields and records for prescription of boundary conditions. Since one possibility how to prescribe input data fields is by discrete function spaces on computational mesh, we begin with mesh setting.

4.2.1 Mesh type

The mesh record and should provide a mesh consisting of points, lines, triangles and tetrahedrons in 3D space and further definition of boundary segments and element connectivity.

record type: **Mesh**

`file` = <*input filename*> DEFAULT: mesh.msh

The file with computational mesh in the ASCII GMSH format.
http://geuz.org/gmsh/doc/texinfo/gmsh.html#MSH-ASCII-file-format

`boundary_segments` = <*array of boundary segments*> DEFAULT: null optional

The set of 0,1, or 2 dimensional boundary faces of the mesh should be partitioned into boundary segments in order to prescribe unique boundary condition on every boundary face. The segments numbers are assigned to boundary faces by iterating through the array. Initially every boundary face has segment number 0. Every record in the array use "auxiliary" physical domains, elements or direct face specification to specify some set of boundary faces. The new segment number is assigned to each face in the set, possibly overwriting previous value.

Physical domains or its parts that appears in the boundary segment definitions are removed form the computational mesh, however, the element numbers of removed elements are stored in the corresponding boundary face and can be used to define face-wise approximations of functions with support on the boundary.

`neighbouring` = <*input file name*> DEFAULT: neigbours.flw

This should be removed as soon as we integrate ngh functionality into flow.

---

record type: **Boundary segment**

`index` = <*integer*> DEFAULT: index in outer array

The index of boundary segment can be used later to prescribe particular type of boundary condition on it. Indices must be greater or equal to 1 and should form more or less continuous sequence. The zero boundary segment is reserved for remaining part of the boundary. By default we assign indices to boundary segments according to the order in their array in mesh, i.e. index (counted form 0) plus 1.

`physical_domains` = <*array of integers*> DEFAULT: null optional

Numbers of physical domains which form the boundary segment. All elements of these physical domains will be removed from actual computational mesh.

`elements` = <*array of integers*> DEFAULT: null optional

The array contains element numbers which should be removed from computational mesh and added to boundary segment.

`sides` = <*array of integer pairs*> DEFAULT: null optional

The array contains numbers of elements which outer faces will be added to the boundary segment or pairs $[element, side\_on\ _{e}lement]$ identifying individual faces

that will be added to the boundary segment.

4.2.2 Time-space field type

A general time and space dependent, scalar, vector, or tensor valued function is given by array of *steady field data*, i.e. time slices. The time slice contains array of *space functions* for individual materials. Then, the *space function* can by either analytical (given by formula) or numerical, given by type of discrete space and array of *elemental functions*. *Elemental function* is just array of values for every degree of freedom on one element.

The function described by this type is tensor values in general and dimensions of this tensor should be specified outside of the function data. For example in description of Transport record type you should specify that function for initial condition is vector valued with vector size equal to number of substances. It is like template for Time-space field type parametrized by shape of the value tensor given by number of lines N and columnes N.

record type: **Steady field data**

`time` = <*double*> DEFAULT: -Inf

Start time for the spatial filed data.

`time_interpolation` = <*enum*> DEFAULT: constant

time interpolation enum cases:

`constant=0` Keeps constant data until next time cut.

`linear=1` Linear interpolation between current time cut and the next one.

`materials` = <*array of Steady spatial functions*> DEFAULT: null optional

record type: **Space function**

`material` = <*integer*> DEFAULT: 0

Material filter. The function has nonzero value only on elements with given material number. Function with filter 0 takes place for all materials where no function is set.

`analytic` = <*multi-array of function formulas*> DEFAULT: null optional

The shape of the multi-array is given by rank of the value of the function. Since formula parser can deal only with scalar functions, we have to specify individual members of resulting tensor. Formulas can contain variables x, y, z, and t. The formula is used until the next time slice and is evaluated for every solved time step (can depend on equation).

Instead of constant formulas one can use double values. Usage of formulas or doubles need not to be uniform over the tensor.

`numeric_type` = <*enum*> DEFAULT: None

FE space enum cases:

`None=0` Use analytic function.

`P0=1` Zero order polynomial on element.

Currently we support only P0 base functions for data.

`numeric` = <*array of element functions*> DEFAULT: empty array

Usually, this element-wise array should by included from an external file through `INCLUDE_ARRAY` construct.

---

record type: **Element function**

`element_id` = <*integer*> DEFAULT: null optional

Element ID in the mesh. By default the element is identified by the index in the array of element function.

`values` = <*multi-array of doubles*> DEFAULT: null mandatory

Values for degrees of freedom of the base functions on the element. Currently we support only P0 functions which are given by value in the barycenter of the element. In general the value can be tensor, i.e. array of arrays of doubles. However, in accordance to simplification rules, you can use only array of doubles for vector functions or mere double for scalar functions.

[tensor/vector valued element function should be given also as simple array of DOF values. But we have to provide ordering of tensor products of FE spaces]

[As follows from next examples, there is no way how to simply set simply tensors. We can introduce automatic conversion form scalar to vector (constant vector) and vector to tensor (diagonal tensor).]

[we should also allow 'in place' array includes to simplify material tables etc.]

[How to allow parallel input of field data?]

[Should be there explicit mesh reference in the field specification?]

Examples:

```
constant_scalar_function = 1.0
# is same as
constant_scalar_function = {
  time = -Inf,
  time_interpolation= constant,
  materials = [
    {
      rank=0
      numeric_type=None
      analytic=1.0       # the only key withou default value
    }
  ]
}

conductivity_tensor =
  [{ material = 1,
```

```
    rank = 2,
    analytic = [[1.0, 0.0, 0.0], [0.0, 1.0, 0.0], [0.0, 0.0, 1.0]]
  },
  { material = 2,
    rank = 2,
    analytic = [[10.0, 0.0, 0.0], [0.0, 10.0, 0.0], [0.0, 0.0, 10.0]]
  }
]
```

4.2.3 Boundary conditions

Input of boundary conditions is similar to the Time-space fields. For description of one time slice we have type *steady boundary data*. This contains array of *boundary conditions* for individual boundary segments. *Boundary condition* is given by type and parameters that are analytic or numeric functions. However, numeric functions are considered only on boundary elements.

record type: **steady boundary data**

`time` = <*double*> DEFAULT: 0.0

Time when the BC should be applied.

`bc` = <*array of boundary conditions*> DEFAULT: empty

record type: **boundary condition**

`boundary_segment` = <*integer*> DEFAULT: 0

Boundary segment where the boundary condition will be applied.

`bc_type` = <*enum*> DEFAULT: dirichlet

Currently there are just three types of boundary condition common to all equations, but some equations can implement some specific boundary conditions. Common boundary condition types are **BC types** enum cases:

`dirichlet=0`

`neumann=1`

`newton=2` (also known as Robin boundary condition)

`value` = <*space function*> DEFAULT: type defaults

Prescribed value for Dirichlet boundary condition and Newton boundary condition, the normal flux for the Neumann boundary condition. Key `material` of *space function* is irrelevant.

`mean_value` = <*scalar or vector constant*> DEFAULT: 0

Prescribes flux for Neumann boundary condition by total flux over the boundary segment. If both *value* and *mean_value* keys are set, we use only *value* key. [How this interact with fracture opening?]

`newton_coef` = <*scalar space function*> DEFAULT: type defaults

Coefficient that appears in the Newton boundary condition. Key `material` of *space function* is irrelevant.

E.g. denoting u the unknown scalar or vector field and $\partial_n u$ density of the normal flux of this field, the meaning of keys `value` and `newton_coef` is following: $\boldsymbol{v}\ \nabla v$

$$u := value \qquad \text{for Dirichlet boundary}$$
$$A\nabla u \cdot \boldsymbol{n} := value \qquad \text{for Neumann boundary}$$
$$A\nabla u \cdot \boldsymbol{n} := newton_coef(u - value) \qquad \text{for Newton boundary}$$

Specific interpretation of the boundary conditions should be described in particular equations.

[How to allow both analytic and numerical functions here?]

[In order to allow changing BC type in time, the structure has to be: time array of BC segments array of BC type with data alternatively we can have just one array of BC patches, where one patch has: time, BC segment, BC type, BC data patches with non monotone time will be scratched]

[need vector valued Dirichlet (and Neuman, and Newton) for Transport boundary conditions]

[More examples...]

4.3 Other record types recognized by Flow123d

4.3.1 Record types not related to equations

record type: **Root record**

`system` = <*system type*> DEFAULT: type defaults

Record with application setting.

`problem` = <*problem type*> DEFAULT: null mandatory

Record with numerical problem to solve.

`material` = <*input filename*> DEFAULT: material.flw

Old material file still used for initialization of data fields in equations. This should be replaced by material database type. Then certain input data fields in equations can be constructed from material informations. Main obstacle are various adsorption algorithms depending on material number.

Only these keys are recognized directly at main level, however you can put here your own keys and then reference to them. For example `mesh` is part of problem type record, but you can put it on the main level and use reference inside `problem`.

[Should we put problem record to the main level?]

[Should we provide "material database"? Possibility to specify properties of individual materials and use them to construct field data for equations.]

record type: **System**

Description. lkajs eflakd flakds fropi pwnfvpweiurv evm qoiefmv lksdm vqoierrv lakfdsvkjweroivlksdmvaoirrf vm a v ve lkvqekfv jf ais i qkd ffo qk dfjhaopdif u wqwddff if j khr if

`pause_after_run` = <*bool*> DEFAULT: no

Wait for press of Enter after run. Good for Windows users, but dangerous for batch computations. Should be rather an command-line option.

`verbosity` = <*bool*> DEFAULT: no

Turns on/off more verbose mode.

`output_streams` = <*output stream*> DEFAULT: null optional

One or more output data streams. There are two predefined output streams:

vtk ascii stream:

```
{
  name="dafault_vtk_ascii"
  file="flow_output"
  type="vtk_ascii"
}
```

GMSH ascii stream:

```
{
  name="dafault_gmsh_ascii"
  file="flow_output"
  type="gmsh_ascii"
}
```

Possibly here could be variables for check-pointing, debugging, timers etc.

record type: **Output stream**

`name` = <*string*> DEFAULT: null mandatory

Name of the output stream. This is used to set output stream for individual output data.

`file` = <*output filename string*> DEFAULT: stream name

File name of the output file for the stream. The file name should be without extension, the correct extension will be appended according to the format type.

`format` = <*enum*> DEFAULT: vtk_ascii

output format enum cases:

`vtk_ascii=0`

`gmsh_ascii=1`

`precision` = <*integer*> DEFAULT: 8

Number of valid decadic digits to output floating point data into the ascii file formats.

`copy_file` = <*output filename string*> DEFAULT: null optional

Optionally one can set copy file to output data into to different file formats.

`copy_format` = <*enum*> DEFAULT: null optional

`copy_precision` = <*integer*> DEFAULT: 8

4.3.2 Equation related record types

Up to now there is only one problem type: `TYPE=sequential\_coupling`, but in near future we should introduce full coupling e.g. for density driven flow.

The *s*equential coupling problem has following keys:

record type: **Sequential coupling** implements *Problem type*

`description` = <*string*> DEFAULT: null optional

Short text description of solved problem. Now it is only reported on the screen, but could be written into output files or used somewhere else.

`mesh` = <*mesh type*> DEFAULT: type defaults

The computational mesh common for both coupled equations.

`time_governor` = <*time governor type*> DEFAULT: type defaults

Common time governor setting. [Future: allow different setting for each equation]

`primary_equation` = <*darcy flow type*> DEFAULT: type defaults

Independent equation.

`secondary_equation` = <*transport type*> DEFAULT: null optional

Equation with some data dependent on the primary equation.

---

record type: **Time governor**

`init_time` = <*double*> DEFAULT: 0.0

Time when an equation starts its simulation.

`time_step` = <*double*> DEFAULT: 1.0

Initial time step.

`end_time` = <*double*> DEFAULT: 1.0

End time for an equation.

This record type should initialize `TimeGovernor` class, but there are still questions about steady TimeGovernor and if we allow user setting for other parameters.

There are three subtypes *steady saturated MH*, *unsteady saturated MH*, *unsteady saturated LMH* The common keys are:

record type: **Darcy flow**(abstract type)

`TYPE` = <*enum*> DEFAULT:

There are three implementations of Darcy flow. Most keys are common but unsteady solvers accept some extra keys. **darcy flow type** enum cases:

`steady_MH=0`

`unsteady_LMH=1`

`unsteady_MH=2`

`sources` = <*time-space field*> DEFAULT: 0

Density of water sources. Scalar valued field (1x1 tensor).

`sources_file` = <*input file name*> DEFAULT: null optional

File with sources in old format. (OBSOLETE)

`source_formula` = <*space function*> DEFAULT: 0

Space function that prescribes water source.

`coef_tensor` = <*tensor steady field*> DEFAULT: 1.0

Conductivity 3x3 tensor. [Should be always 3x3 and then restricted on 2d and 1d fractures.]

`boundary_condition` = <*array of steady boundary data*> DEFAULT: null mandatory

New scheme for setting boundary conditions.

`boundary_file` = <*input file name*> DEFAULT: null mandatory

File with boundary conditions in old format. (OBSOLETE)

`solver` = <*solver type*> DEFAULT: type defaults

`n_schurs` = <*integer*> DEFAULT: 2

Number of Schur complements to make. Valid values are 0,1,2.

`output` = <*darcy flow output type*> DEFAULT: typ defaults

This is just sub record to separate output setting.

`initial` = <*steady data type*> DEFAULT: null mandatory

Initial condition. Scalar valued field (1x1 tensor). (for unsteady only)

`initial_file` = <*input file name*> DEFAULT: null mandatory

File with initial condition in old format. (OBSOLETE)

`storativity` = <*steady data type*> DEFAULT: null mandatory

Storativity coefficient. Scalar valued field (1x1 tensor). (for unsteady only)

record type: **Darcy flow output**

`save_step` = <*double*> DEFAULT: null optional

Time step between outputs.

`output_times` = <*array of doubles*> DEFAULT: null optional

Force output in prescribed times. Can be combined with regular otuptu given by `save_step`.

`velocity_p0` = <*output stream name*> DEFAULT: default_vtk_ascii

`pressure_p0` = <*output stream name*> DEFAULT: default_vtk_ascii

`pressure_p1` = <*output stream name*> DEFAULT: default_vtk_ascii

4.3.3 Solver type

record type: **Solver type** (abstract)

`TYPE` = <*enum*> DEFAULT: petsc

solver types enum cases:

`petsc=0` Use any PETSc solver for MPIAIJ matrices.

`bddc=1` Use BDDC solver (need not to work with every equation).

`accuracy` = <*double*> DEFAULT: solvers defaults

Absolute residual tolerance.

`max_it` = <*integer*> DEFAULT: 1000

Maximum number of outer iterations.

`parameters` = <*string*> DEFAULT: null optional

String with options for PETSc solvers.

`export_to_matlab` = <*bool*> DEFAULT: no

Save every solved system in matlab format. Useful for debugging and numerical experiments.

4.3.4 Transport type

record type: **Transport type**

`TYPE` = <*enum*> DEFAULT:

Two types are sofar possible. The second type is for advection-diffusion equation which needs implicit solver. **transport type** enum cases:

`TransportOperatorSplitting=0`

`AdvectionDiffusion_DG=1`

`sorption` = <*bool*> DEFAULT: no

`dual_porosity` = <*bool*> DEFAULT: no

`transport_reactions` = <*bool*> DEFAULT: no

What kind of reactions is this? Age of water?

`sigma` = <*double*> DEFAULT: 0

Coefficient of diffusive transfer through fractures.

`alpha_l` = <*double*> DEFAULT: 0

Longitudal dispersivity.

`alpha_t` = <*double*> DEFAULT: 0

Transversal dispersivity.

`d_m` = <*double*> DEFAULT: 1e-6

Molecular diffusivity.

`dg_penalty` = <*double*> DEFAULT: 0

Penalty parameter influencing the discontinuity of the solution.

`reactions` = <*reaction type*> DEFAULT: null optional

Currently only Semchem is supported. [Interface to Phreaq ...]

`decays` = <*array of decays*> DEFAULT: null optional

`substances` = <*array of strings*> DEFAULT: null mandatory

Names for transported substances. Number of substances is given implicitly by size of the array.

`initial` = <*steady data type*> DEFAULT: null mandatory

Vector valued initial condition for mobile phase of all species.

`initial_others` = <*steady data type*> DEFAULT: null optional

Tensor valued initial condition for immobile, mobile-sorbed, immobile-sorbed phases and all species. (3 x n_substances). [alternatively have separate key for each phase]

`initial_file` = <*input file name*> DEFAULT: null mandatory

File with initial condition in old format. (OBSOLETE)

`boundary_condition` = <*array of steady boundary data*> DEFAULT: null mandatory

New scheme for setting boundary conditions.

`boundary_file` = <*input file name*> DEFAULT: null mandatory

File with boundary condition in old format. (OBSOLETE) For time dependent boundary conditions, the filename is postfixed with number of time level.

`bc_times` = <*array of doubles*> DEFAULT: null optional

Times for changing boundary conditions. If you set this variable, you have to prepare a separate file with boundary conditions for every time in the list. Filenames for individual time level are formed from BC filename by appending underscore and three digits of time level number, e.g. `transport_bcd_000, transport_bcd_001, etc.` (OBSOLETE)

`output` = <*transport output*> DEFAULT: type defaults

---

record type: **Transport output**

`save_step` = <*double*> DEFAULT: null optional

Time step between outputs.

`output_times` = <*array of doubles*> DEFAULT: null optional

Force output in prescribed times. Can be combined with regular otuptu given by `save_step`.

`mobile_p0` = <*output stream name*> DEFAULT: default_vtk_ascii

`immobile_p0` = <*output stream name*> DEFAULT: default_vtk_ascii

`mobile_sorbed_p0` = <*output stream name*> DEFAULT: default_vtk_ascii

`immobile_sorbed_p0` = <*output stream name*> DEFAULT: default_vtk_ascii

`mobile_p1` = <*output stream name*> DEFAULT: default_vtk_ascii

`immobile_p1` = <*output stream name*> DEFAULT: default_vtk_ascii

`mobile_sorbed_p1` = <*output stream name*> DEFAULT: default_vtk_ascii

`immobile_sorbed_p1` = <*output stream name*> DEFAULT: default_vtk_ascii

record type: **Decay chain**

`substance_ids` = <*array of integers*> DEFAULT: empty

Sequence of N ids of transported substances defining isotopes contained in the decay chain under consideration.

`nr_of_children` = <*array of integers*> DEFAULT: either empty or filled up with ones

Number of children of each vertex (belonging to one of isotopes) in a simple graph describing considered decay chain. (NOT USED, YET.)

`indices_of_children` = <*array of integers*> DEFAULT: either emmpty or filled up with substanc

Containes identifiers of children of vertex. Children are assigned to vertices throuh the use of numbers listed in nr_of_children. (NOT USED, YET.)

`half_lives` = <*array of doubles*> DEFAULT: empty

This array containes $N-1$ half-lives belonging to isotopes contained in the array substance_ids. If there are no bifurcation key specified, the decay chain is linear $1 \to 2 \to 3$. If there is the bifurcation key, the decay chain is branched $1 \to 2, 1 \to 3$. (Temporary solution.)

`bifurcation` = <*array of double*> DEFAULT: null optional

It should have as many items as the array indices_of_children $(N-1)$. It defines relative part of parental contribution in the case of division of decay chain into more branches. (NOT USED, YET.)
Contains $N-1$ probabilities for individual branches of the bifurcation decay. (Temporary solution.) They should sum to one.

---

record type: **First order reactions**

`substance_ids` = <*array_of_integers*> DEFAULT: empty

Sequence of K ids describing a set of consequtive kinetic reactions of the first order. It is meant to enable simulation of $A \to B \to C \to ... \to Z$ set of reactions easily. (NOT USED, YET.)

`kinetic_constants` = <*array or doubles*> DEFAULT: filled up with 0.0s

Sequence of $K-1$ doubles defining kinetic constants of partial reactions from the set defined in substance_ids. (NOT USED, YET.)

`nr_of_children` = <*array of integer*> DEFAULT: filled up with 1.0s

Number of children of each vertex (belonging to one of species) in a simple graph describing considered first order reactions. (NOT USED, YET.)

`indices_of_children` = <*array of integers*> DEFAULT: either emmpty or filled up with substanc

Containes identifiers of children of vertex. Children are assigned to vertices throuh the use of numbers listed in nr_of_children. (NOT USED, YET.)

`bifurcation` = <*array of double*> DEFAULT: null optional

It should have as many items as the array indices_of_children $(K-1)$. It defines relative part of parental contribution in the case of division of first order reactions describing graph into more branches. (NOT USED, YET.)

`kinetic_konstant` = <*double*> DEFAULT: 0.0

Defines kinetic konstant which describes a simple reaction of the type $A \to B$.

`substance_ids` = <*array of integers*> DEFAULT: empty

Contains 2 identifiers of subsatnces (species) which are taking part in considered reaction.

---

record type: **Simple reactions**

`decay_chains` = <*array of Decay chains*> DEFAULT: empty

Sequence of records describing decay chains under consideration. (NOT USED, YET.)

`first_order_reactions` = <*array of first order reactions*> DEFAULT: empty

Sequence of records describing first order reactions under consideration. (NOT USED, YET.)

`pade_nominator_degree` = <*integer*> DEFAULT: 2

This number defines the degree of matrix polynomial appearing in nominator of Pade approximant of a matrix exponential.

`pade_denominator_degree` = <*integer*> DEFAULT: 2

This number defines the degree of matrix polynomial appearing in denominator of Pade approximant of a matrix exponential.

4.4 Other input files

4.4.1 Mesh file format version 2.0

The only supported format for the computational mesh is MSH ASCII format produced by the GMSH software. You can find its documentation on:

http://geuz.org/gmsh/doc/texinfo/gmsh.html#MSH-ASCII-file-format

Comments concerning Flow123d:

- Every inconsistency of the file stops the calculation. These are:
  - Existence of nodes with the same *node-number*.
  - Existence of elements with the same *elm-number*.
  - Reference to non-existing node.
  - Reference to non-existing material (see below).
  - Difference between *number-of-nodes* and actual number of lines in nodes' section.
  - Difference between *number-of-elements* and actual number of lines in elements' section.
- By default Flow123d assumes meshes with *number-of-tags* = 3.

tag1 is number of material (reference to `.MTR` file) in the element.

tag2 is number of geometry region in which the element lies.

tag3 is partition number (CURRENTLY NOT USED).

In accordance with specification of GMSH mesh format.

- Currently, line (*type* = 1), triangle (*type* = 2) and tetrahedron (*type* = 4) are the only supported types of elements. Existence of an element of different type stops the calculation.
- Wherever possible, we use the file extension `.MSH`. It is not required, but highly recomended.
- This file format can be used also for storing simple dicrete scalar or vector fields. We support output into this format (see Section 4.5)

4.4.2 Neighbouring file format, version 1.0

The file is divided in two sections, header and data. The extension `.NGH` is highly recomended for files of this type.

`$NeighbourFormat`
`1.0` *file-type* *data-size*
`$EndNeighbourFormat`
`$Neighbours`
number-of-neighbours
neighbour-number *type* <*type-specific-data*>
`...`
`$EndNeighbours`

where

file-type `int` — is equal 0 for the ASCII file format.

data-size `int` — the size of the floating point numbers used in the file. Usually *data-size* = sizeof(double).

number-of-neighbours `int` — Number of neighbouring defined in the file.

neighbour-number `int` — is the number (index) of the n-th neighbouring. These numbers do not have to be given in a consecutive (or even an ordered) way. Each number has to be given only onece, multiple definition are treated as inconsistency of the file and cause stopping the calculation.

type `int` — is type of the neighbouring.

<*type-specific-data*> — format of this list depends on the *type*.

Types of neighbouring and their specific data

type = `10` — "Edge with common nodes", i.e. sides of elements with common nodes. (Possible many elements)

type = `11` — "Edge with specified sides", i.e. sides of the edge are explicitly defined. (Possible many elements)

type = `20` — "Compatible", i.e. volume of an element with a side of another element. (Only two elements)

type = `30` — "Non-compatible" i.e. volume od an element with volume of another element. (Only two elements)

type	*type-specific-data*	Description
10	*n_elm eid1 eid2 ...*	number of elements and their ids
11	*n_sid eid1 sid1 eid2 sid2 ...*	number of sides, their elements and local ids
20	*eid1 eid2 sid2 coef*	Elm 1 has to have lower dimension
30	*eid1 eid2 coef*	Elm 1 has to have lower dimension

coef is of the `double` type, other variables are `int`s.

Comments concerning `Flow123d`:

- Every inconsistency or error in the `.NGH` file causes stopping the calculation. These are especially:
  - Multiple usage of the same *neighbour-number*.
  - Difference between *number-of-neighbours* and actual number of data lines.
  - Reference to nonexisting element.
  - Nonsence number of side.
- The variables *sid?* must be nonegative and lower than the number of sides of the particular element.

4.4.3 Material properties file format, version 1.0

The file is divided in two sections, header and data. The extension `.MTR` is highly recomended for files of this type.

```
$MaterialFormat
1.0 file-type  data-size
$EndMaterialFormat
$Materials
number-of-materials
material-number  material-type <material-type-specific-data>  [text]
...
$EndMaterials
```

`$Storativity`
material-number <storativity-coefficient> [text]
...
`$EndStorativity`
`$Geometry`
material-number geometry-type <geometry-type-specific-coefficient> [text]
...
`$EndGeometry`
`$Sorption`
material-number substance-id sorption-type <sorption-type-specific-data> [text]
...
`$EndSorption`
`$SorptionFraction`
material-number <sorption-fraction-coefficient> [text]
...
`$EndSorptionFraction`
`$DualPorosity`
material-number <mobile-porosity-coefficient> <immobile-porosity-coefficient>
<nonequillibrium-coefficient-substance(0)>
...<nonequilibrium-coefficient-substance(n-1)> [text]
...
`$EndDualPorosity`
`$Reactions`
reaction-type <reaction-type-specific-coefficient> [text]
...
`$EndReactions`

where:

file-type `int` — is equal 0 for the ASCII file format.

data-size `int` — the size of the floating point numbers used in the file. Usually *data-size* = sizeof(double).

number-of-materials `int` — Number of materials defined in the file.

material-number `int` — is the number (index) of the n-th material. These numbers do not have to be given in a consecutive (or even an ordered) way. Each number has to be given only onece, multiple definition are treated as inconsistency of the file and cause stopping the calculation (exception $Sorption section).

material-type `int` — is type of the material, see table.

<material-type-specific-data > — format of this list depends on the *material - type*.

<storativity-coefficient> `double` — coefficient of storativity

geometry-type `int` — type of complement dimension parameter (only for 1D and 2D material), for 1D element is supported type 1 - cross-section area, for 2D element is supported type 2 - thickness.

<*geometry-type-specific-coefficient*> `double` — cross-section for 1D element or thickness for 2D element.

substance-id `int` — refers to number of transported substance, numbering starts on *0*.

sorption-type `int` — type 1 - linear sorption isotherm, type 2 - Freundlich sorption isotherm, type 3 - Langmuir sorption isotherm.

<*sorption-type-specific-data* > — format of this list depends on the *sorption - type*, see table.

Note: Section $Sorption is needed for calculation only if *Sorption* is turned on in the *ini* file.

<*sorption-fraction-coefficient*> `double` — ratio of the "mobile" solid surface in the contact with "mobile" water to the total solid surface (this parameter (section) is needed for calculation only if *Dual_porosity* and *Sorption* is together turned on in the ini file).

<*mobile-porosity-coefficient*> `double` — ratio of the mobile pore volume to the total volume (this parameter is needed only if *Transport_on* is turned on in the ini file).

<*immobile-porosity-coefficient*> `double` — ratio of the immobile pore volu-me to the total pore volume (this parameter is needed only if *Dual_porosity* is turned on in the ini file).

<*nonequilibrium-coefficient-substance(i)*> `double` — nonequilibrium coefficient for substance i, $\forall i \in \langle 0, n-1 \rangle$ where n is number of transported substances (this parameter is needed only if *Dual_porosity* is turned on in the ini file).

reaction-type `int` — type 0 - zero order reaction

<*reaction-type-specific-data* > — format of this list depends on the *reaction - type*, see table.

material-type	*material-type-specific-data*	Description
11	k	$\mathbf{K} = (k)$
-11	a	$\mathbf{A} = \mathbf{K}^{-1} = (a)$
21	k	$\mathbf{K} = \begin{pmatrix} k & 0 \\ 0 & k \end{pmatrix}$
22	$k_x \quad k_y$	$\mathbf{K} = \begin{pmatrix} k_x & 0 \\ 0 & k_y \end{pmatrix}$
23	$k_x \quad k_y \quad k_{xy}$	$\mathbf{K} = \begin{pmatrix} k_x & k_{xy} \\ k_{xy} & k_y \end{pmatrix}$
-21	a	$\mathbf{A} = \mathbf{K}^{-1} = \begin{pmatrix} a & 0 \\ 0 & a \end{pmatrix}$
-22	$a_x \quad a_y$	$\mathbf{A} = \mathbf{K}^{-1} = \begin{pmatrix} a_x & 0 \\ 0 & a_y \end{pmatrix}$
-23	$a_x \quad a_y \quad a_{xy}$	$\mathbf{A} = \mathbf{K}^{-1} = \begin{pmatrix} a_x & a_{xy} \\ a_{xy} & a_y \end{pmatrix}$
31	k	$\mathbf{K} = \begin{pmatrix} k & 0 & 0 \\ 0 & k & 0 \\ 0 & 0 & k \end{pmatrix}$
33	$k_x \quad k_y \quad k_z$	$\mathbf{K} = \begin{pmatrix} k_x & 0 & 0 \\ 0 & k_y & 0 \\ 0 & 0 & k_z \end{pmatrix}$
36	$k_x \quad k_y \quad k_z \quad k_{xy} \quad k_{xz} \quad k_{yz}$	$\mathbf{K} = \begin{pmatrix} k_x & k_{xy} & k_{xz} \\ k_{xy} & k_y & k_{yz} \\ k_{xz} & k_{yz} & k_z \end{pmatrix}$
-31	a	$\mathbf{A} = \mathbf{K}^{-1} = \begin{pmatrix} a & 0 & 0 \\ 0 & a & 0 \\ 0 & 0 & a \end{pmatrix}$
-33	$a_x \quad a_y \quad a_z$	$\mathbf{A} = \mathbf{K}^{-1} = \begin{pmatrix} a_x & 0 & 0 \\ 0 & a_y & 0 \\ 0 & 0 & a_z \end{pmatrix}$
-36	$a_x \quad a_y \quad a_z \quad a_{xy} \quad a_{xz} \quad a_{yz}$	$\mathbf{A} = \mathbf{K}^{-1} = \begin{pmatrix} a_x & a_{xy} & a_{xz} \\ a_{xy} & a_y & a_{yz} \\ a_{xz} & a_{yz} & a_z \end{pmatrix}$

Note: all variables (k, k_x, k_y, k_z, k_{xy}, k_{xz}, k_{yz}, a, a_x, a_y, a_z, a_{xy}, a_{xz}, a_{yz}) are of the `double` type.

sorption-type	*sorption-type-specific-data*	Description
1	$k_D[1]$	$s = k_D c$
2	$k_F[(L^{-3} \cdot M^1)^{(1-\alpha)}] \quad \alpha[1]$	$s = k_F c^\alpha$
3	$K_L[L^3 \cdot M^{-1}] \quad s^{max}[L^{-3} \cdot M^1]$	$s = \frac{K_L s^{max}\ c}{1+K_L c}$

Note: all variables (k_D, k_F, α, K_L, s^{max}) are of the `double` type.

reaction-type	*reaction-type-specific-data*	Description
0	*substance-id*[1] $\quad k[M \cdot L^{-3} \cdot T^{-1}]$	$\frac{\partial c_m^{[substance\text{-}id]}}{\partial t} = k$

Where $c_m^{[substance\text{-}id]}$ is mobile concentration of substance with id *substance-id* and Δt is the internal transport time step defined by CFL condition.

text `char[]` — is a text description of the material, up to 256 chars. This parameter is

optional.

Comments concerning `Flow123d`:

- If *number-of-materials* differs from actual number of material lines in the file, it stops the calculation.

4.4.4 Boundary conditions file format, version 1.0

The file is divided in two sections, header and data.

`$BoundaryFormat`
`1.0` *file-type data-size*
`$EndBoundaryFormat`
`$BoundaryConditions`
number-of-conditions
condition-number type <*type-specific-data*> *where* <*where-data*> *number-of-tags* <*tags*> *[text]*
`...`
`$EndBoundaryConditions`

where

file-type `int` — is equal 0 for the ASCII file format.

data-size `int` — the size of the floating point numbers used in the file. Usually *data-size* = sizeof(double).

number-of-conditions `int` — Number of boundary conditions defined in the file.

condition-number `int` — is the number (index) of the n-th boundary condition. These numbers do not have to be given in a consecutive (or even an ordered) way. Each number has to be given only onece, multiple definition are treated as inconsistency of the file and cause stopping the calculation.

type `int` — is type of the boundary condition. See below for definitions of the types.

<*type-specific-data*> — format of this list depends on the *type*. See below for specification of the *type-specific-data* for particular types of the boundary conditions.

where `int` — defines the way, how the place for the contidion is prescribed. See below for details.

<*where-data*> — format of this list depends on *where* and actually defines the place for the condition. See below for details.

number-of-tags `int` — number of integer tags of the boundary condition. It can be zero.

< *tags* > *number-of-tags*`*int` — list of tags of the boundary condition. Values are separated by spaces or tabs. By default we set *number-of-tags*=1, where *tag1* defines group of boundary conditions, "type of water" in our jargon. This can be used to calculate total fluxes through the boundary group.

[text] `char[]` — arbitrary text, description of the fracture, notes, etc., up to 256 chars. This is an optional parameter.

Types of boundary conditions and their data

type = `1` — Boundary condition of the Dirichlet's type

type = `2` — Boundary condition of the Neumann's type

type = `3` — Boundary condition of the Newton's type

type	*type-specific-data*	Description
1	*scalar*	Prescribed value of pressure (in meters [m])
2	*flux*	Prescribed value of flux through the boundary
3	*scalar sigma*	Scalar value and the σ coefficient

scalar, *flux* and *sigma* are of the `double` type.

Ways of defining the place for the boundary condition

where = `1` — Condition on a node

where = `2` — Condition on a (generalized) side

where = `3` — Condition on side for element with only one external side.

where	<*where-data*>	Description
1	*node-id*	Node id number, according to `.MSH` file
2	*elm-id sid-id*	Elm. id number, local number of side
3	*elm-id*	Elm. id number

The variables *node-id*, *elm-id*, *sid-id* are of the `int` type.

Comments concerning `Flow123d`:

- We assume homegemous Neumman's condition as the default one. Therefore we do not need to prescribe conditions on the whole boundary.
- If the condition is given on the inner edge, it is treated as an error and stops calculation.
- Any inconsistence in the file stops calculation. (Bad number of conditions, multiple definition of condition, reference to non-existing node, etc.)

- At least one of the conditions has to be of the Dirichlet's or Newton's type. This is well-known fact from the theory of the PDE's.
- Local numbers of sides for *where* = 2 must be lower than the number of sides of the particular element and greater then or equal to zero.
- The element specified for *where* = 3 must have only one external side, otherwise the program stops.

4.4.5 Transport boundary conditions file format, version 1.0

The file is divided in two sections, header and data.

`$Transport_BCDFormat`
`1.0` *file-type data-size*
`$EndTransport_BCDFormat`
`$Transport_BCD`
number-of-conditions
transport-condition-number boundary-condition-number value1 value2 ...
`$EndTransport_BCD`

where

file-type `int` - is equal 0 for the ASCII file format.

data-size `int` - the size of the floating point numbers used in the file. Usually data-size = sizeof(double)

number-of-conditions `int` - Number of conditions defined in the file.

transport-condition-number `int` - is the number (index) of the n-th transport condition. These numbers do not have to be given in a consecutive (or even an ordered) way. Each number has to be given only once, multiple definition are treated as inconsistency of the file and cause stopping the calculation.

boundary-condition-number `int` - id number of the boundary-condition where transport boundary condition is prescribed.

valueN `double` - prescribed boundary concentration of substance N (should be from interval $[0, 1]$).

Comments concerning FLOW123d: Number of transport boundary conditions has to be same as number of boundary conditions. Program stops computation in the other case.

4.4.6 Element data file format, version 1.0

Several input data fields are given as constant scalars on every element. In particular this is used for water sources, initial condition of pressure, initial condition for concentrations and substance sources in transport. Common file format of these files is:

```
$FieldName
number-of-lines
eid value1 value2 ...
...
$EndFieldName
```

where

`$FieldName` — Unique name of the input field. Since all field data are enclosed by `$FieldName` and `$EndFieldName` one can even have different fields in one common file.

number-of-sources `int` — Number of data lines that has to match number of elements in the mesh.

eid `int` — is id-number of the element (in the input mesh file).

valueN `double` — list of field values. Number of values is specific for each particular type of input.

Description of individual input fields.

water sources FieldName=`Sources`, there is only one value per line — the density of water source on the element.

pressure initial condition FieldName=`PressureHead`, there is only one value per line — the initial pressure value on the element.

substance sources FieldName=`TransportSources`, number of values is 3 times number of substances. The density of one substance source is given by formula:

$$f = d + \sigma(c - c_N)$$

where f is total source, the first term is fixed Neuman-like source density d. The second term is Newton-like source density, where σ is transmisitvity, c is actual concentration, and c_N is prescribed concentration. For every substance there is triplet of three parameters: d, σ, c_N. The order of substances is same as in the main INI file.

concentration initial conditions FieldName=`Concentrations`, number of values equal to number of transported substances, the order of substances is same as in the main INI file.

Comments concerning `Flow123d`:

- Every inconsistency or error in the `.SRC` file causes stopping the calculation. These are especially:
  - Difference between *number-of-lines* and actual number of data lines.
  - Reference to nonexisting element.

4.5 Output files

Flow123d support output of scalar and vaector data fields into two formats. First one can use native fromat of program GMSH (usualy with extension `msh`) which contains computational mesh and then various datafields for squence of time levels. For second we support output into XML version of VTK files. These files can be viewed and postprocessed by several vizualization softwares. However, our primal goal is to support data transfer into Paraview vizualization software. See key Pos_format.

4.5.1 Output data fields of water flow module

Water flow module provides output of four data fields.

pressure on elements Pressure head in length units $[L]$ piecewise constant on every element. This field is directly produced by the MH method and thus contains no postprocessing error.

pressure in nodes Same pressure head field, but interpolated into $P1$ continuous scalar field. Namely you lost dicontinuities on fractures.

velocity on elements Vector field of water flux volume unit per time unit $[L^3/T]$. For every element we evaluate discrete flux field in barycenter.

piezometric head on elements Piezometric head in length units $[L]$ piecewise constant on every element. This is just pressure on element plus z-coordinate of the barycenter. This field is produced only on demand (see key output_piezo_head).

4.5.2 Output data fields of transport

Transport module provides output only for concentrations (in mobile phase) as a field piecewise constant over elements. There is one field for every substance and names of those fields contain names of substances given by key Substances. The physical unit is mass unit over volume unit $[M/L^3]$.

4.5.3 Auxiliary output files

Profiling information

On every run we collect some bacis profilling informations. After all computations these data are written into the file `profiler%y%m%d_%H.%M.%S.out` where `%y`, `%m`, `%d`, `%H`, `%M`, `%S` are two digit numbers representing year, month, day, hour, minute, and second of the program start time.

Water flux information

File contains water flow balance, total inflow and outflow over boundary segments. Further there is total water income due to sources (if they are present).

Raw water flow data file

You can force Flow123d to write raw data about results of MH method. The file format is:

```
$FlowField
T=<time>
<number fo elements>
<eid> <pressure> <flux x> <flux y> <flux z> <number of sides> <pressures on sides> <flu
...
$EndFlowField
```

where

`<time>` — is simulation time of the raw output.

`<number of elements>` — is number of elements in mesh, which is same as number of subsequent lines.

`<eid>` — element id same as in the input mesh.

`<flux x,y,z>` — components of water flux interpolated to barycenter of the element

`<number of sides>` — number of sides of the element, infulence number of remaining values

`<pressures on sides>` — for every side average of the pressure over the side

`<fluxes on sides>` — for ever side total flux through the side

Chapter 5

Test and tutorial problems

Chapter 6

Units

Quantity	**Unit**
lenght	L
time	T
conductivity	
concentration	
diffusivity	

Table 6.1: The table of units used in the document.

Chapter 7

Tests

7.1 Test 01 – Steady flow

This test considers steady Darcy flow in a cube domain which is cutted by 2D fractures which are further separated by a 1D channel in their cross section. The multidimensional connections between 1D, 2D and 3D elements are involved in the computation. Dirichlet boundary condition is prescribed for flow.

- *problem type* – sequential coupling,
- *primary equation* – steady mixed hybrid

Geometry and boundary conditions

A cube with its side 1.0 L is cutted by two diagonal 2D fractures which are further separated by a 1D channel in their cross section.

Geometry parameters are defined for different physical domains. Thickness of the 2D fractures is set to 1.0 L and the area of the cross section is set to 1.0 L^2. These parameters are unrealistic (the side of the cube is 1.0 L long) but it is compensated in the equation in the fraction with conductivity.

There are only simple Dirichlet boundary conditions. Pressure gradient in direction from one corner of the cube to the oposite corner is applied on all boundary faces of all dimensions.

Parameters

- **Cross section area:** 1D channel is set to 1.0 L^2.
- **Thickness:** 2D fractures are set to 1.0 L.
- **Conductivity:** The conductivity of materials:
  - 1D channel is set to $K = 10$.
  - 2D fractures are set to $\mathbf{K} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$.

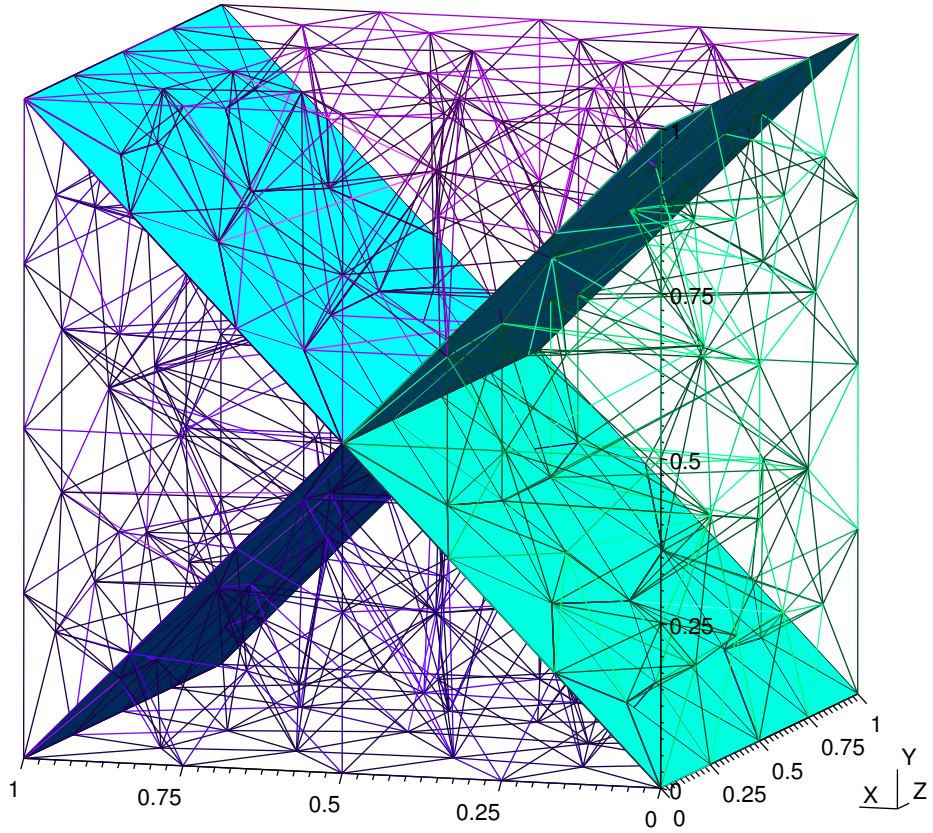

Figure 7.1: Test 01 – mesh

– cube material is set to $\mathbf{K} = \begin{pmatrix} 0.1 & 0 & 0 \\ 0 & 0.1 & 0 \\ 0 & 0 & 0.1 \end{pmatrix}$.

- There is no transport so there are not any other parameters.

Verification

This test verifies solving steady Darcy flow by mixed hybrid method. There are different dimensional connections which are 2D-3D connection between the cube and the flat fractures and 1D-2D connection between the 1D channel and the two flat fractures in their crossection.

7.2 Test 02 – Steady flow in 2D and transport

This test involves steady Darcy flow in 2D, connections of 1D-2D elements, Dirichlet boundary condition for flow and transport, transport of two substances with zero initial condition for concentration. There are acutally two different cases computed in this test. Dual porosity and sorption features in explicit transport. Dispersion is defined in implicit transport.

The coefficient of diffusive transfer through a fracture (means between the fracture and

the surrounding material) is set to zero so the substance cannot be diffused through the fracture's boundary.

- *problem type* – sequential coupling,
- *primary equation* – steady mixed hybrid
- *secondary equation* – transport operator splitting (explicit), discontinuous Galerkin method (implicit)

Geometry and boundary conditions

The domain is two-dimensional slice through a part of a relief which involves several one-dimensional fractures.

Simple Dirichlet flow boundary condition is defined on left and right side where pressure heads are prescribed. There is no flow through the upper and lower boundary of the model. This all causes a flow along the x axis.

Dirichlet boundary condition for transport is prescribed on both sides as it is for flow boundary condition and the value of concentration is 1.0 for both substances. Initial concentration of the substances is zero in the whole area.

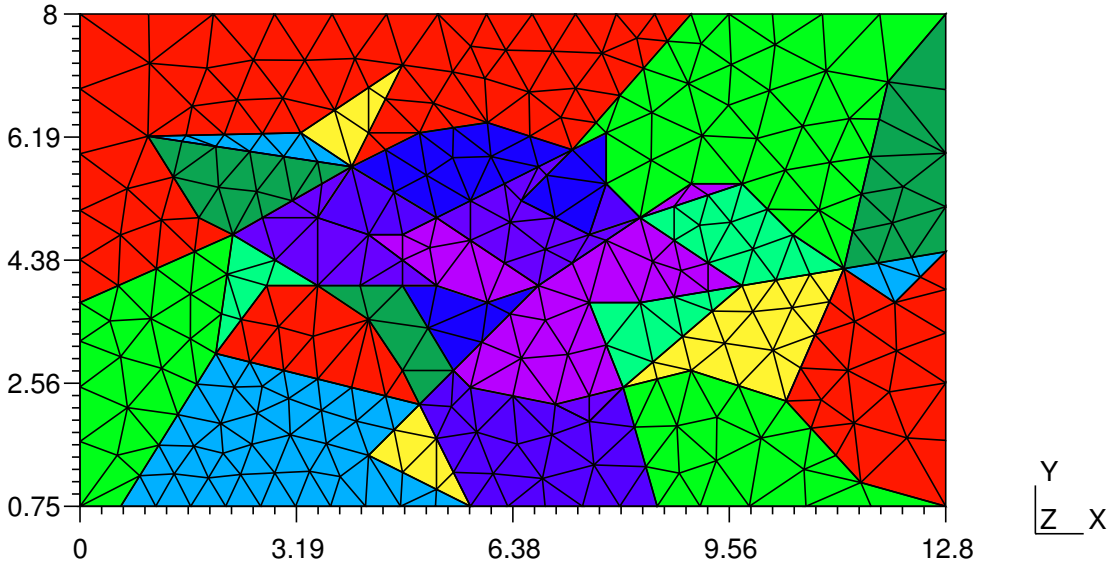

Figure 7.2: Test 02 – mesh

Parameters

The flow is steady and the transport is solved in time interval $(0, 5.0)$. The output is written every 0.5. Time parameters for implicitly computed transport are the same only initial time step is set to 0.5.

- **Cross section area:** 1D fractures are set to 1.0 L^2.
- **Thickness:** domain is set to 1.0 L.
- **Conductivity:** The conductivity of materials:
  - fracture material is set to $K = 10$.
  - plane material is set to $\mathbf{K} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$.
- **Sorption:** The sorption parameters are for both materials equal:
  - linear sorption isotherm parameter of the first substance is set to $k_d = 0.02$.
  - Freundlich sorption isotherm parameters of the second substance are set to $k_f = 0.02$, $\alpha = 0.5$
- **Dual porosity:** The dual porosity parameters are for both materials equal:
  - mobile porosity coefficient is set to 0.25
  - immobile porosity coefficient is set to 0.25
  - nonequilibrium coefficient of both substances 0.01
- **Sorption fraction:** The sorption fraction parameters are for both materials equal and set to $SF = 0.5$.
- **Diffusivity coefficients:** These are not set so default values are applied $\sigma = 0$, $\alpha_l = 0$, $\alpha_t = 0$, $d_m = 1e - 6$.

Verification

This test verifies explicitly computed transport considering only convection with dual porosity and sorption and implicitly computed transport considering both convection and dispersion. Transport through 1D-2D element connections is computed in addition to the first test.

7.3 Test 03 – Steady flow in 2D and transport

This test differs from the previous one only by simpler structure of its geometry. It shows how the substace flows in the main fracture and divides in two other fractures. The substance spreads in the fractures much faster in comparision to transport in the plane.

Geometry and boundary conditions

There is a plane with side 1.0 which is cutted by fractures. The main fracture divides in two other fractures.

Parameters

The flow is steady and the transport is solved in time interval $(0, 1.0)$. The output is written every 0.01. Initial time step for transport computed implicitly is set to 0.1 and the output is written every 0.1.

Other parameters are the same as in test 02.

Verification

This test verifies the same features as the test 02 does but on a simpler geometry.

7.4 Test 05 – Darcy flow boundary conditions

There are three types of boundary conditions – Dirichlet, Neumann and Robin that are tested. All three test have the same geometry and boundary conditions are derived from the same analytical solution.

We will prescribe analytical solution $u = xy$ of Laplace equation $-\Delta u = 0$.

- *problem type* – sequential coupling,
- *primary equation* – steady mixed hybrid

Geometry and boundary conditions

The geometry is simple – square plane in xy coordinates with corner points [0,0] and [1,1]. Each side has its own boundary regions called `.bc_south`, `.bc_east`, `.bc_north`, `.bc_west`.

Dirichlet test. All sides have pressure prescribed. These are south: $u_D = 0$; east: $u_D = y$; north: $u_D = x$; west: $u_D = 0$.

Neumann test. Two sides have pressure prescribed for the Dirichlet boundary condition: east: $u_D = y$; west: $u_D = 0$. Two other sides have flux prescribed: south: $q_N = x$; north $q_N = -x$.

Robin test. Two sides have pressure prescribed for the Dirichlet boundary condition: east: $u_D = y$; west: $u_D = 0$. For Robin boundary condition we get from the equation boundary pressure

$$u_R = \frac{1 + \sigma_R}{\sigma_R} x. \tag{7.1}$$

We choose $\sigma_R = 0.5$ and then we get $u_R = -2x$ on the south side and $u_R = 3x$ on the north side.

Parameters

- **Conductivity:** on region `plane` is 1.0.

- **Thickness:** on region `plane` is by default 1.0 L.
- There are no other regions, no transport so there are not any other parameters.

Verification

This test verifies prescribing different types boundary conditions.

7.5 Test 08 – Steady Darcy flow with source

This test is aimed at verifying steady Darcy flow with source which is prescribed by space function formula. This formula is processed by the function parser.

We will solve Laplace equation $-\Delta u = f$ where source f is prescribed by function: $f = 2(1 - y^2) + 2(1 - x^2)$.

We can easily prove that the analytic solution is $u = (1 - x^2)(1 - y^2)$ by replacing it in the Laplace equation.

- *problem type* – sequential coupling,
- *primary equation* – steady mixed hybrid

Geometry and boundary conditions

The domain is a square with opposite vertices $[-1, -1]$ and $[1, 1]$. Zero dirichlet boundary condition is prescribed on all boundaries – along the circumference of the square.

Parameters

- **Conductivity:** The conductivity of plane material is 1.0.
- There are no other materials, no transport so there are not any other parameters.

Verification

As it was mentioned above, this test mainly verifies that the function parser works properly. The source formula to be parsed is given in the key `source_formula`. The solution (pressure) is a paraboloid with a top in $[0, 0, 1]$.

7.6 Test 10 – Unsteady flow in 2D

Unsteady flow in 2D domain is simulated in this test and is computed by both mixed hybrid and lumped mixed hybrid method. No transport is involved.

- *problem type* – sequential coupling,

- *primary equation* – unsteady mixed hybrid, unsteady lumped mixed hybrid

Geometry and boundary conditions

The domain is a square with oposite vertices $[0, 0]$ and $[1, 1]$. Different Dirichlet boundary condition for flow is prescribed on two opposite sides – 0.0 on the left and 100.0 on the right.

Parameters

The flow is solved in time interval $(0, 0.5)$ with step 0.01. The output is written every 0.1.

- **Conductivity:** The conductivity of plane material is 0.02.
- Initial pressure is set to zero everywhere.
- There are no other materials, no transport so there are not any other parameters.

Verification

This test verifies two different numerical methods – the problem is computed by both mixed hybrid and lumped mixed hybrid method.

7.7 Test 11 – Radioactive decay chain with more branches

8 isotopes are members of considered decay chain with three branches. Transport boundary conditions does not matter because zero presure gradient is considered. Final concentrations of all isotopes except C decrease to zero after 20 time steps, whereas C concentration grows to 0.36.

$$E \to D \to F \to B \quad \begin{array}{lll} 0.2B \to A & A \to G & \\ 0.6B \to H & H \to G & G \to C \\ 0.2B \to G & & \end{array}$$

- *problem type* – sequential coupling,
- *primary equation* – steady mixed hybrid
- *secondary equation* – transport operator splitting
- *reactions* – linear reactions

Geometry

The domain is a prism which base is a right-angled triangle with its ordinates 3.0 units long. There are then only three tetrahedron elements in the mesh.

Parameters

The flow is steady and the transport is solved in time interval $(0, 10.0)$. The output is written every 0.5.

Half-lives are equal to 0.5 for all isotopes. Initial concentrations are summarized in the table below:

isotop	A	B	C	D	E	F	G	H
initial concentration	0.01	0.02	0.03	0.04	0.05	0.06	0.07	0.08

Verification

7.8 Test 12 – Radioactive decay

There are actually two tests of the radioactive decay. The first one considers first order reaction of two isotopes determined by kinetic constant and the other one describes radioactive decay chain of three isotopes.

- *problem type* – sequential coupling,
- *primary equation* – steady mixed hybrid
- *secondary equation* – transport operator splitting
- *reactions* – linear reactions

Geometry and boundary conditions

The domain is a prism which base is a right-angled triangle with its ordinates 3.0 units long. There are then only three tetrahedron elements in the mesh.

There are two Dirichlet boundary conditions for flow prescribed.

- **Conductivity:** The conductivity of the prism material is 0.01.
- There is no other parameter for flow or transport.

7.8.1 First order reaction determined by kinetic constant

The only linear reaction between D and F substances.

$$D \xrightarrow{k} F$$

Parameters

The flow is steady and the transport is solved in time interval $(0, 10.0)$. The output is written every 0.5.

- **Substances:** 6 substances to be transported – A, B, C, D, E, F
- **Kinetic constant:** $k = 0.277258872$

Verification

7.8.2 Radioactive decay chain

The considered radioctive decay chain is:

$$D \xrightarrow{t_{1/2,D}} F \xrightarrow{t_{1/2,F}} B$$

Parameters

Time parameters are the same as they are above.

- **Substances:** 6 substances to be transported – A, B, C, D, E, F
- **Decay half-lives:** $t_{1/2,D} = t_{1/2,F} = 2.5$

Verification

7.9 Test 13 – Solute mixing on the edge

This test realizes mixing of substances on the edges of planes and also does quantitative test on a trivial transport problem. The problem is computed with both explicit and implicit transport.

- *problem type* – sequential coupling,
- *primary equation* – steady mixed hybrid
- *secondary equation* – transport operator splitting (explicit), discontinuous Galerkin method (implicit)

Geometry and boundary conditions

The domain is a fork where the main branch of length 5 with the incoming solute is in the xy plane. Then it is divided into two other branches of length $5\sqrt{2}$, one with positive and the another with negative z coordinate. There are different conductivities in each branch.

Dirichlet boundary conditions for flow and transport are prescribed at the beginning of the main plane ($x = 0$) and at the end of the secondary branches ($x = 10$).

flow: $h_D = -x - z + 10.0$ which gives 10.0 at point [0,0,0] and ± 5 at points [10,0,∓ 5]
transport: concentration is 1.0 at point [0,0,0] and 0.0 at points [10,0,∓ 5]

Initial concentration of the substances is zero in the whole area.

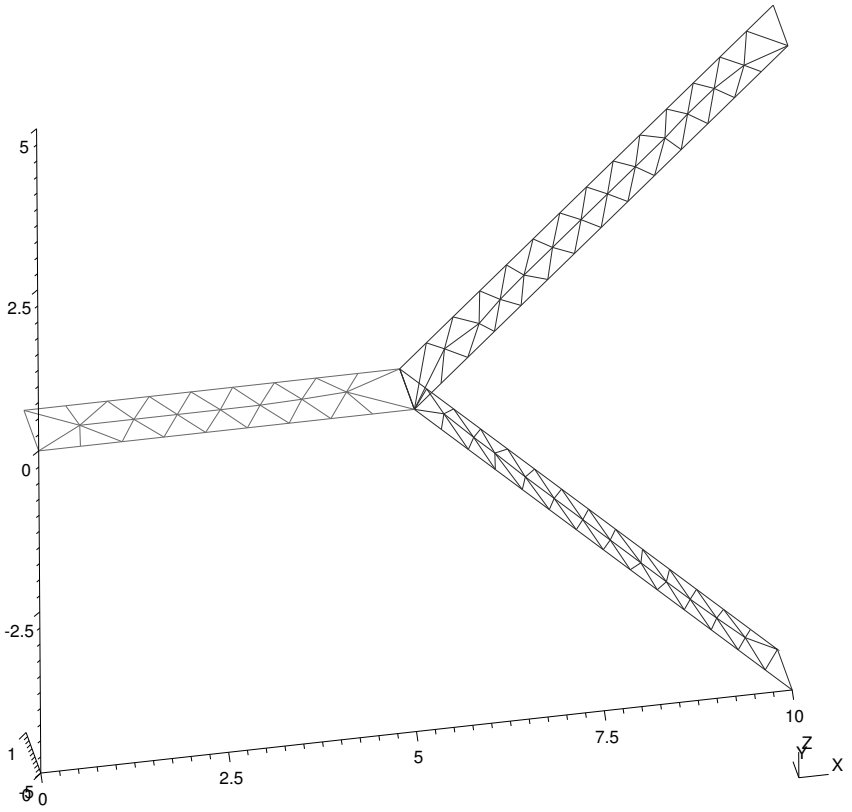

Figure 7.3: Test 13 – mesh

Parameters

The flow is steady and the transport is solved in time interval $(0, 100.0)$. The output is written every 0.5. Time parameters for implicitly computed transport are the same only initial time step and output time is set to 5.0.

- **Thickness:** all planes are set to 1.0 L.
- **Conductivity:** The conductivity of materials (isotropic planes):
  - main branch (material num. 17): $K = 1$.
  - branch (positive z, material num. 18): $K = 0.1$.
  - branch (negative z, material num. 19): $K = 0.1$.
- **Diffusivity coefficients:** are used in implicit transport wth dispersion. Default parameters are set ($d_m = 1e - 6$, others are zero, see manual or parameters in test02 in 7.2).

7.10 Test 14 – Variable transport boundary condition

There is considered a time variable boundary condition for transport in this test. Steady flow with constant velocity is caused by a pressure gradient from one side of a 2D strip to another. Dirichlet boundary condition for transport evolving in time is prescribed on the right side.

- *problem type* – sequential coupling,
- *primary equation* – steady mixed hybrid
- *secondary equation* – transport operator splitting (explicit), discontinuous Galerkin method (implicit)

Geometry and boundary conditions

Flow boundary condition is prescribed all around the plane and is equal x and causes constant flow from right to left (pressure prescribed on the upper and lower sides are along x equal so have no influence). Transport boundary condition has the same prescribtion as flow only the values evolves in time.

Initial concentration is zero on the whole plane. Two pulses of nonzero concentration are applied on the boundary. The changes of the boundary condition at specified times are shown in the following table:

time	0	1	3	6	7
concentration	0	20	0	40	0

- **Thickness:** all planes are set to 1.0 L.
- **Conductivity:** The conductivity of material (isotropic plane): $K = 0.1$.
- **Diffusivity coefficients:** are used in implicit transport wth dispersion. Default parameters are set except $dg_penalty = 1e-4$ (see manual or parameters in test02 in 7.2).

Parameters

The flow is steady and the transport is solved in time interval $(0, 10.0)$. The output is written every 1.0. Time parameters for implicitly computed transport are the same only initial time step is set to 1.0.

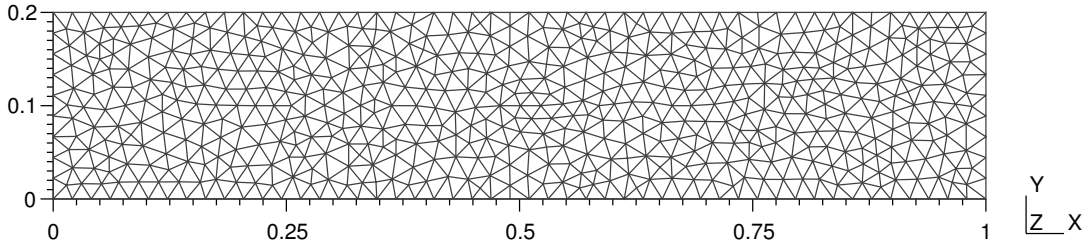

Figure 7.4: Test 14 – mesh

Verification

7.11 Test 15 – Unsteady flow with transport

Transport of a single pulse of concentration moving along a 2D strip is solved. This test involves unsteady flow computed by lumped hybrid method, transport is solved both with explicit and implicit (involves dispersion) scheme.

- *problem type* – sequential coupling,
- *primary equation* – unsteady lumped mixed hybrid
- *secondary equation* – transport operator splitting (explicit), discontinuous Galerkin method (implicit)

Geometry and boundary conditions

The domain is a 2D strip with dimensions 1.0x16.0. Zero Dirichlet boundary for flow is prescribed at $x = 0$, zero Neumann boundary is elsewhere.

Dirichlet transport boundary condition is set on the left side to 10.0 only at the beginning. Then is this boundary condition zero.

Parameters

Initial pressure is zero everywhere. The source is prescribed with function $f = -x$ along the strip.

- **Thickness:** all planes are set to 1.0 L.
- **Conductivity:** The conductivity of material (isotropic plane): $K = 1.0$.
- **Source formula:** $f = -x$
- **Diffusivity coefficients:** are used in implicit transport wth dispersion. Default parameters are set ($d_m = 1e-6$, others are zero, see manual or parameters in test02 in 7.2).

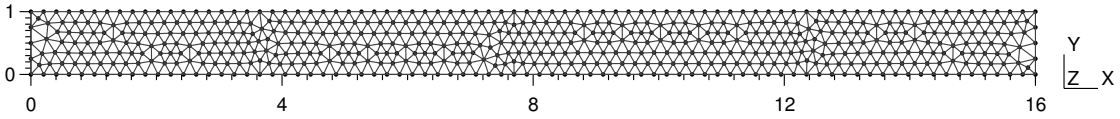

Figure 7.5: Test 15 – mesh

Verification

The test is similar to the test 10 but here in addition the computation of a transport in an unsteady flow field is verified.

7.12 Test 16 – Substance concentration source in transport

This test include a source of concentration of a substance. The domain is a 2D strip in vertical direction. There is a steady flow with constant velocity in the vertical direction. Two sources are situated on two elements at the top of the strip and the substance is transported down along the strip. The concentration values of the sources are defined in the `tso` input file.

- *problem type* – sequential coupling,
- *primary equation* – steady mixed hybrid
- *secondary equation* – transport operator splitting

Geometry

Parameters

Verification

7.13 Test 17 – Radioactive decay – Pade approximation

This test solves radioactive decay chain of five isotopes using Pade approximation. The considered radioctive decay chain is:

$$A \xrightarrow{t_{1/2,A}} B \xrightarrow{t_{1/2,B}} C \xrightarrow{t_{1/2,C}} D \xrightarrow{t_{1/2,D}} E$$

- *problem type* – sequential coupling,
- *primary equation* – steady mixed hybrid

- *secondary equation* – transport operator splitting
- *reactions* – Pade approximation

Geometry

The geometry and material and transport parameters are the same as in test 12.

Parameters

- **Substances:** 5 substances to be transported – A, B, C, D, E
- Polynomial degree of the nominator and the denominator of Pade approximation is 3.
- **Decay half-lives:**

$t_{1/2,A}$	$t_{1/2,B}$	$t_{1/2,C}$	$t_{1/2,D}$
1.3863	2.3105	1.5403	1.1552

Verification

7.14 Test 18 – Diffusion through fractures

This test is aimed at transport caused just by diffusion.

There is a triangular domain with zero pressure everywhere so no flow is present. Triangular element with high concentration of a substance lies in the middle of the domain and its sides neighbour with fractures. The coeffients of molecular diffusion and diffusive transfer through fractures are the parameters of the implicit transport and are set in the configuration file.

Geometry

Parameters

Verification

========== PREPARING ==========

7.15 Test 20 – Dirichlet boundary condition

This test involves steady Darcy flow in 3D determined by Dirichlet boundary condition. The analytic solution is prescribed $u = xyz$. We can see from the formula that there are no sources $-\Delta u = 0$ (zero right hand side) and we can easily define Dirichlet boundary conditions on the sides of the cube just by evaluating the solution there.

Geometry

The domain is a cube with its side 1.0 L long. Dirichlet boundary conditions are summarized in the following table. Physical domains corresponds with the numbers in `geo` file, the row *plane* contains equations of the planes (sides of the cube). The row *Dirichlet* contains solution on the planes. The row *boundary* segment contains numbers of segments defined in `con` file.

boundary segment	1	2	3	4	5	6
physical domain	27	28	29	30	31	32
plane	$z-1=0$	$x-1=0$	$z=0$	$x=0$	$y-1=0$	$y=0$
Dirichlet $[u_D]$	xy	yz	0	0	xz	0

Parameters

- **Conductivity:** cube material is set to $\mathbf{K} = \begin{pmatrix} 1.0 & 0 & 0 \\ 0 & 1.0 & 0 \\ 0 & 0 & 1.0 \end{pmatrix}$.
- There is no transport so there are not any other parameters.

Verification

This test verifies prescribing Dirichlet boundary condition.

7.16 Test 21 – Neumann boundary condition

This test uses the same geometry and parameters as in the test 20 (viz 7.15) but there are prescribed both Dirichlet and Neumann boundary conditions.

The table of the boundary conditions is below. The row *Dirichlet* contains contains solution on the planes and the row *Neumann* contains flow through the planes.

boundary segment	1	2	3	4	5	6
physical domain	27	28	29	30	31	32
plane	$z-1=0$	$x-1=0$	$z=0$	$x=0$	$y-1=0$	$y=0$
Dirichlet $[u_D]$	-	yz	0	0	-	0
Neumann $[-\nabla u \cdot \mathbf{n}]$	$-xy$	-	-	-	$-xz$	-

Verification

This test verifies prescribing Neumann boundary condition.

7.17 Test 22 – Newton boundary condition

This test uses the same geometry and parameters as in the test 20 (viz 7.15) but there is prescribed Newton boundary condition $-\nabla u \cdot \mathbf{n} = \sigma(u - u_T)$.

The table of the boundary conditions where parameters σ and u_T are written is below. The values of parameters were chosen to satisfy condition $-\nabla u \cdot \mathbf{n} = -(yz, xz, xy) \cdot \mathbf{n} = \sigma(u - u_T)$

boundary segment	1	2	3	4	5	6
physical domain	27	28	29	30	31	32
plane	$z - 1 = 0$	$x - 1 = 0$	$z = 0$	$x = 0$	$y - 1 = 0$	$y = 0$
$-\nabla u \cdot \mathbf{n}$	$-xy$	$-yz$	xy	yz	$-xz$	
σ	xy	yz	0	0	xz	0
u_T	$u_T = xy$	$u = yz$	$u = 0$	$u = 0$	$u = xz$	$u = 0$

Verification

This test verifies prescribing Newton boundary condition.

Chapter 8

Main input file reference

abstract type: **Problem**

Descendants:

The root record of description of particular the problem to solve.

SequentialCoupling

record: **SequentialCoupling** implements abstract type: Problem

Record with data for a general sequential coupling.

`TYPE` = *<selection: Problem_TYPE_selection>*

Default: SequentialCoupling []

Sub-record selection.

`description` = *<String (generic)>*

Default: *<optional>* []

Short description of the solved problem. Is displayed in the main log, and possibly in other text output files.

`mesh` = *<record: Mesh>*

Default: *<obligatory>* []

Computational mesh common to all equations.

`time` = *<record: TimeGovernor>*

Default: *<optional>* []

Simulation time frame and time step.

`primary_equation` = *<abstract type: DarcyFlowMH>*

Default: *<obligatory>* []

Primary equation, have all data given.

`secondary_equation` = *<abstract type: Transport>*

Default: <*optional*> []

The equation that depends (the velocity field) on the result of the primary equation.

---

record: **Mesh**

Record with mesh related data.

`mesh_file` = <*input file name*>

Default: <*obligatory*> []

Input file with mesh description.

`regions` = <*Array of record: Region*>

Default: <*optional*> []

List of additional region definitions not contained in the mesh.

`sets` = <*Array of record: RegionSet*>

Default: <*optional*> []

List of region set definitions. There are three region sets implicitly defined: ALL (all regions of the mesh), BOUNDARY (all boundary regions), and BULK (all bulk regions)

---

record: **Region**

Definition of region of elements.

`name` = <*String (generic)*>

Default: <*obligatory*> []

Label (name) of the region. Has to be unique in one mesh.

`id` = <*Integer [0,]*>

Default: <*obligatory*> []

The ID of the region to which you assign label.

`element_list` = <*Array of Integer [0,]*>

Default: <*optional*> []

Specification of the region by the list of elements. This is not recomended

---

record: **RegionSet**

Definition of one region set.

`name` = <*String (generic)*>

Default: <*obligatory*> []

Unique name of the region set.

`region_ids` = <*Array of Integer [0,]*>

Default: <*optional*> []

List of region ID numbers that has to be added to the region set.

`region_labels` = <*Array of String (generic)*>

Default: <*optional*> []

List of labels of the regions that has to be added to the region set.

`union` = <*Array [2, 2] of String (generic)*>

Default: <*optional*> []

Defines region set as a union of given pair of sets. Overrides previous keys.

`intersection` = <*Array [2, 2] of String (generic)*>

Default: <*optional*> []

Defines region set as an intersection of given pair of sets. Overrides previous keys.

`difference` = <*Array [2, 2] of String (generic)*>

Default: <*optional*> []

Defines region set as a difference of given pair of sets. Overrides previous keys.

---

record: **TimeGovernor**

Setting of the simulation time. (can be specific to one eqaution)

`start_time` = <*Double* >

Default: 0.0 []

Start time of the simulation.

`end_time` = <*Double* >

Default: <*obligatory*> []

End time of the simulation.

`init_dt` = <*Double [0,]*>

Default: <*optional*> []

Initial guess for the time step. The time step is fixed if hard time step limits are not set.

`min_dt` = <*Double [0,]*>

Default: "Machine precision or 'init_dt' if specified" []

Hard lower limit for the time step.

`max_dt` = <*Double [0,]*>

Default: "Whole time of the simulation or 'init_dt' if specified" []

Hard upper limit for the time step.

---

abstract type: **DarcyFlowMH**

Descendants:

Mixed-Hybrid solver for saturated Darcy flow.

Steady_MH

Unsteady_MH

Unsteady_LMH

record: **Steady_MH** implements abstract type: DarcyFlowMH

Mixed-Hybrid solver for STEADY saturated Darcy flow.

`TYPE` = *<selection: DarcyFlowMH_TYPE_selection>*

Default: Steady_MH []

Sub-record selection.

`n_schurs` = *<Integer [0, 2]>*

Default: 2 []

Number of Schur complements to perform when solving MH sytem.

`solver` = *<abstract type: Solver>*

Default: *<obligatory>* []

Linear solver for MH problem.

`output` = *<record: DarcyMHOutput>*

Default: *<obligatory>* []

Parameters of output form MH module.

`mortar_method` = *<selection: MH_MortarMethod>*

Default: None []

Method for coupling Darcy flow between dimensions.

`mortar_sigma` = *<Double [0,]>*

Default: 1.0 []

Conductivity between dimensions.

`bc_data` = *<Array of record: DarcyFlowMH_Steady_BoundaryData>*

Default: *<obligatory>* []

`bulk_data` = *<Array of record: DarcyFlowMH_Steady_BulkData>*

Default: *<obligatory>* []

abstract type: **Solver**

Descendants:

Solver setting.

Petsc

Bddc

---

record: **Petsc** implements abstract type: Solver

Solver setting.

`TYPE` = *<selection: Solver_TYPE_selection>*

Default: Petsc []

Sub-record selection.

`a_tol` = *<Double [0,]>*

Default: 1.0e-9 []

Absolute residual tolerance.

`r_tol` = *<Double [0, 1]>*

Default: 1.0e-7 []

Relative residual tolerance (to initial error).

`max_it` = *<Integer [0,]>*

Default: 10000 []

Maximum number of outer iterations of the linear solver.

`options` = *<String (generic)>*

Default: []

Options passed to the petsc instead of default setting.

---

record: **Bddc** implements abstract type: Solver

Solver setting.

`TYPE` = *<selection: Solver_TYPE_selection>*

Default: Bddc []

Sub-record selection.

`a_tol` = *<Double [0,]>*

Default: 1.0e-9 []

Absolute residual tolerance.

`r_tol` = *<Double [0, 1]>*

Default: 1.0e-7 []

Relative residual tolerance (to initial error).

`max_it` = *<Integer [0,]>*

Default: 10000 []

Maximum number of outer iterations of the linear solver.

record: **DarcyMHOutput**

Parameters of MH output.

`save_step` = <*Double [0,]*>

Default: 1.0 []

Regular step between MH outputs.

`output_stream` = <*record: OutputStrem*>

Default: <*obligatory*> []

Parameters of output stream.

`velocity_p0` = <*String (generic)*>

Default: <*optional*> []

Output stream for P0 approximation of the velocity field.

`pressure_p0` = <*String (generic)*>

Default: <*optional*> []

Output stream for P0 approximation of the pressure field.

`pressure_p1` = <*String (generic)*>

Default: <*optional*> []

Output stream for P1 approximation of the pressure field.

`piezo_head_p0` = <*String (generic)*>

Default: <*optional*> []

Output stream for P0 approximation of the piezometric head field.

`balance_output` = <*output file name*>

Default: water_balance.txt []

Output file for water balance table.

`raw_flow_output` = <*output file name*>

Default: <*optional*> []

Output file with raw data form MH module.

record: **OutputStrem**

Parameters of output.

`name` = <*String (generic)*>

Default: <*obligatory*> []

The name of this stream. Used to reference the output stream.

`file` = <*output file name*>

Default: <*obligatory*> []

File path to the output stream.

`format` = <*abstract type: OutputFormat*>

Default: <*optional*> []

Format of output stream and possible parameters.

---

abstract type: **OutputFormat**

Descendants:

Format of output stream and possible parameters.

`vtk`

`gmsh`

---

record: **vtk** implements abstract type: OutputFormat

Parameters of vtk output format.

`TYPE` = <*selection: OutputFormat_TYPE_selection*>

Default: vtk []

Sub-record selection.

`variant` = <*selection: VTK variant (ascii or binary)*>

Default: ascii []

Variant of output stream file format.

`parallel` = <*Bool*>

Default: false []

Parallel or serial version of file format.

`compression` = <*selection: Type of compression of VTK file format*>

Default: none []

Compression used in output stream file format.

---

selection type: **VTK variant (ascii or binary)**

Possible values:

`ascii` : ASCII variant of VTK file format

`binary` : Binary variant of VTK file format (not supported yet)

---

selection type: **Type of compression of VTK file format**

Possible values:

`none` : Data in VTK file format are not compressed

`zlib` : Data in VTK file format are compressed using zlib (not supported yet)

record: **gmsh** implements abstract type: OutputFormat

Parameters of gmsh output format.

`TYPE` = <*selection: OutputFormat_TYPE_selection*>

Default: gmsh []

Sub-record selection.

selection type: **MH_MortarMethod**

Possible values:

`None` : Mortar space: P0 on elements of lower dimension.

`P0` : Mortar space: P0 on elements of lower dimension.

`P1` : Mortar space: P1 on intersections, using non-conforming pressures.

record: **DarcyFlowMH_Steady_BoundaryData**

Record to set BOUNDARY fields of the equation 'DarcyFlowMH_Steady'. The fields are set only on the domain specified by one of the keys: 'region', 'rid', 'r_set' and after the time given by the key 'time'. The field setting can be overridden by any DarcyFlowMH_Steady_BoundaryData record that comes later in the boundary data array.

`r_set` = <*String (generic)*>

Default: <*optional*> []

Name of region set where to set fields.

`region` = <*String (generic)*>

Default: <*optional*> []

Label of the region where to set fields.

`rid` = <*Integer [0,]*>

Default: <*optional*> []

ID of the region where to set fields.

`time` = <*Double [0,]*>

Default: 0.0 []

Apply field setting in this record after this time. These times has to form an increasing sequence.

`bc_type` = <*abstract type: Field:R3 → Enum*>

Default: <*optional*> []

Boundary condition type, possible values:

`bc_pressure` = <*abstract type: Field:R3 → Real*>

Default: <*optional*> []

Dirichlet BC condition value for pressure.

`bc_flux` = <*abstract type: Field:R3 → Real*>

Default: <*optional*> []

Flux in Neumman or Robin boundary condition.

`bc_robin_sigma` = <*abstract type: Field:R3 → Real*>

Default: <*optional*> []

Conductivity coefficient in Robin boundary condition.

`bc_piezo_head` = <*abstract type: Field:R3 → Real*>

Default: <*optional*> []

Boundary condition for pressure as piezometric head.

`flow_old_bcd_file` = <*input file name*>

Default: <*optional*> []

---

abstract type: **Field:R3 → Enum** default descendant: FieldConstant

---

Descendants:

Abstract record for all time-space functions.

`FieldConstant`

`FieldFormula`

`FieldPython`

`FieldElementwise`

---

record: **FieldConstant** implements abstract type: Field:R3 → Enum constructible from key: value

---

R3 → Enum Field constant in space.

`TYPE` = <*selection: Field:R3 → Enum_TYPE_selection*>

Default: FieldConstant []

Sub-record selection.

`value` = <*selection: EqData_bc_Type*>

Default: <*obligatory*> []

Value of the constant field. For vector values, you can use scalar value to enter constant vector. For square NxN-matrix values, you can use: * vector of size N to enter diagonal matrix * vector of size (N+1)*N/2 to enter symmetric matrix (upper triangle, row by row) * scalar to enter multiple of the unit matrix.

---

selection type: **EqData_bc_Type**

Possible values:

`none` :

`dirichlet` :

`neumann` :

`robin` :

`total_flux` :

---

record: **FieldFormula** implements abstract type: Field:R3 → Enum

---

R3 → Enum Field given by runtime interpreted formula.

`TYPE` = *<selection: Field:R3 → Enum_TYPE_selection>*

Default: FieldFormula []

Sub-record selection.

`value` = *<String (generic)>*

Default: *<obligatory>* []

String, array of strings, or matrix of strings with formulas for individual entries of scalar, vector, or tensor value respectively. For vector values, you can use just one string to enter homogeneous vector. For square NxN-matrix values, you can use: * array of strings of size N to enter diagonal matrix * array of strings of size (N+1)*N/2 to enter symmetric matrix (upper triangle, row by row) * just one string to enter (spatially variable) multiple of the unit matrix. Formula can contain variables x,y,z,t and usual operators and functions.

---

record: **FieldPython** implements abstract type: Field:R3 → Enum

---

R3 → Enum Field given by a Python script.

`TYPE` = *<selection: Field:R3 → Enum_TYPE_selection>*

Default: FieldPython []

Sub-record selection.

`script_string` = *<String (generic)>*

Default: "Obligatory if 'script_file' is not given." []

Python script given as in place string

`script_file` = *<input file name>*

Default: "Obligatory if 'script_striong' is not given." []

Python script given as external file

`function` = *<String (generic)>*

Default: *<obligatory>* []

Function in the given script that returns tuple containing components of the return type. For NxM tensor values: tensor(row,col) = tuple(M*row + col).

---

record: **FieldElementwise** implements abstract type: Field:R3 → Enum

R3 → Enum Field constant in space.

`TYPE` = <*selection: Field:R3 → Enum_TYPE_selection*>

Default: FieldElementwise []

Sub-record selection.

`gmsh_file` = <*input file name*>

Default: <*obligatory*> []

Input file with ASCII GMSH file format.

`field_name` = <*String (generic)*>

Default: <*obligatory*> []

The values of the Field are read from the $ElementData section with field name given by this key.

---

abstract type: **Field:R3 → Real** default descendant: FieldConstant

Descendants:

Abstract record for all time-space functions.

`FieldConstant`

`FieldPython`

`FieldFormula`

`FieldElementwise`

`FieldInterpolatedP0`

---

record: **FieldConstant** implements abstract type: Field:R3 → Real constructible from key: value

R3 → Real Field constant in space.

`TYPE` = <*selection: Field:R3 → Real_TYPE_selection*>

Default: FieldConstant []

Sub-record selection.

`value` = <*Double* >

Default: <*obligatory*> []

Value of the constant field. For vector values, you can use scalar value to enter constant vector. For square NxN-matrix values, you can use: * vector of size N to enter diagonal matrix * vector of size (N+1)*N/2 to enter symmetric matrix (upper triangle, row by row) * scalar to enter multiple of the unit matrix.

---

record: **FieldPython** implements abstract type: Field:R3 → Real

R3 → Real Field given by a Python script.

`TYPE` = <*selection: Field:R3 → Real_TYPE_selection*>

Default: FieldPython []

Sub-record selection.

`script_string` = <*String (generic)*>

Default: "Obligatory if 'script_file' is not given." []

Python script given as in place string

`script_file` = <*input file name*>

Default: "Obligatory if 'script_striong' is not given." []

Python script given as external file

`function` = <*String (generic)*>

Default: <*obligatory*> []

Function in the given script that returns tuple containing components of the return type. For NxM tensor values: tensor(row,col) = tuple(M*row + col).

---

record: **FieldFormula** implements abstract type: Field:R3 → Real

---

R3 → Real Field given by runtime interpreted formula.

`TYPE` = <*selection: Field:R3 → Real_TYPE_selection*>

Default: FieldFormula []

Sub-record selection.

`value` = <*String (generic)*>

Default: <*obligatory*> []

String, array of strings, or matrix of strings with formulas for individual entries of scalar, vector, or tensor value respectively. For vector values, you can use just one string to enter homogeneous vector. For square NxN-matrix values, you can use: * array of strings of size N to enter diagonal matrix * array of strings of size (N+1)*N/2 to enter symmetric matrix (upper triangle, row by row) * just one string to enter (spatially variable) multiple of the unit matrix. Formula can contain variables x,y,z,t and usual operators and functions.

---

record: **FieldElementwise** implements abstract type: Field:R3 → Real

---

R3 → Real Field constant in space.

`TYPE` = <*selection: Field:R3 → Real_TYPE_selection*>

Default: FieldElementwise []

Sub-record selection.

`gmsh_file` = <*input file name*>

Default: <*obligatory*> []

Input file with ASCII GMSH file format.

`field_name` = *<String (generic)>*

Default: *<obligatory>* []

The values of the Field are read from the $ElementData section with field name given by this key.

---

record: **FieldInterpolatedP0** implements abstract type: Field:R3 → Real

---

Field given by P0 data on another mesh. Currently defined only on boundary.

`TYPE` = *<selection: Field:R3 → Real_TYPE_selection>*

Default: FieldInterpolatedP0 []

Sub-record selection.

`mesh` = *<input file name>*

Default: *<obligatory>* []

File with the mesh from which we interpolate. (currently only GMSH supported)

`raw_data` = *<input file name>*

Default: *<obligatory>* []

File with raw output from flow calculation. Currently we can interpolate only pressure.

---

abstract type: **Field:R3 → Real** default descendant: FieldConstant

---

Descendants:

Abstract record for all time-space functions.

`FieldConstant`

`FieldFormula`

`FieldPython`

`FieldElementwise`

---

record: **FieldConstant** implements abstract type: Field:R3 → Real constructible from key: value

---

R3 → Real Field constant in space.

`TYPE` = *<selection: Field:R3 → Real_TYPE_selection>*

Default: FieldConstant []

Sub-record selection.

`value` = *<Double >*

Default: *<obligatory>* []

Value of the constant field. For vector values, you can use scalar value to enter

constant vector. For square NxN-matrix values, you can use: * vector of size N to enter diagonal matrix * vector of size (N+1)*N/2 to enter symmetric matrix (upper triangle, row by row) * scalar to enter multiple of the unit matrix.

---

record: **FieldFormula** implements abstract type: Field:R3 → Real

---

R3 → Real Field given by runtime interpreted formula.

`TYPE` = <*selection: Field:R3 → Real_TYPE_selection*>

Default: FieldFormula []

Sub-record selection.

`value` = <*String (generic)*>

Default: <*obligatory*> []

String, array of strings, or matrix of strings with formulas for individual entries of scalar, vector, or tensor value respectively. For vector values, you can use just one string to enter homogeneous vector. For square NxN-matrix values, you can use: * array of strings of size N to enter diagonal matrix * array of strings of size (N+1)*N/2 to enter symmetric matrix (upper triangle, row by row) * just one string to enter (spatially variable) multiple of the unit matrix. Formula can contain variables x,y,z,t and usual operators and functions.

---

record: **FieldPython** implements abstract type: Field:R3 → Real

---

R3 → Real Field given by a Python script.

`TYPE` = <*selection: Field:R3 → Real_TYPE_selection*>

Default: FieldPython []

Sub-record selection.

`script_string` = <*String (generic)*>

Default: "Obligatory if 'script_file' is not given." []

Python script given as in place string

`script_file` = <*input file name*>

Default: "Obligatory if 'script_striong' is not given." []

Python script given as external file

`function` = <*String (generic)*>

Default: <*obligatory*> []

Function in the given script that returns tuple containing components of the return type. For NxM tensor values: tensor(row,col) = tuple(M*row + col).

---

record: **FieldElementwise** implements abstract type: Field:R3 → Real

---

R3 → Real Field constant in space.

`TYPE` = <*selection: Field:R3 → Real_TYPE_selection*>

Default: FieldElementwise []

Sub-record selection.

`gmsh_file` = <*input file name*>

Default: <*obligatory*> []

Input file with ASCII GMSH file format.

`field_name` = <*String (generic)*>

Default: <*obligatory*> []

The values of the Field are read from the $ElementData section with field name given by this key.

---

record: **DarcyFlowMH_Steady_BulkData**

Record to set BULK fields of the equation 'DarcyFlowMH_Steady'. The fields are set only on the domain specified by one of the keys: 'region', 'rid', 'r_set' and after the time given by the key 'time'. The field setting can be overridden by any DarcyFlowMH_Steady_BulkData record that comes later in the bulk data array.

`r_set` = <*String (generic)*>

Default: <*optional*> []

Name of region set where to set fields.

`region` = <*String (generic)*>

Default: <*optional*> []

Label of the region where to set fields.

`rid` = <*Integer [0,]*>

Default: <*optional*> []

ID of the region where to set fields.

`time` = <*Double [0,]*>

Default: 0.0 []

Apply field setting in this record after this time. These times has to form an increasing sequence.

`cond_anisothropy` = <*abstract type: Field:R3 → Real[3,3]*>

Default: <*optional*> []

Anisothropic conductivity tensor.

`cross_section` = <*abstract type: Field:R3 → Real*>

Default: <*optional*> []

Complement dimension parameter (cross section for 1D, thickness for 2D).

`conductivity` = <*abstract type: Field:R3 → Real*>

Default: <*optional*> []

Isothropic conductivity scalar.

`water_source_density` = <*abstract type: Field:R3 → Real*>

Default: <*optional*> []

Water source density.

`init_pressure` = <*abstract type: Field:R3 → Real*>

Default: <*optional*> []

Initial condition as pressure

`storativity` = <*abstract type: Field:R3 → Real*>

Default: <*optional*> []

Storativity.

`init_piezo_head` = <*abstract type: Field:R3 → Real*>

Default: <*optional*> []

Initial condition for pressure as piezometric head.

---

abstract type: **Field:R3 → Real[3,3]** default descendant: FieldConstant

---

Descendants:

Abstract record for all time-space functions.

`FieldConstant`

`FieldPython`

`FieldFormula`

`FieldElementwise`

`FieldInterpolatedP0`

---

record: **FieldConstant** implements abstract type: Field:R3 → Real[3,3] constructible from key: value

---

R3 → Real[3,3] Field constant in space.

`TYPE` = <*selection: Field:R3 → Real[3,3]_TYPE_selection*>

Default: FieldConstant []

Sub-record selection.

`value` = <*Array [1,] of Array [1,] of Double* >

Default: <*obligatory*> []

Value of the constant field. For vector values, you can use scalar value to enter constant vector. For square NxN-matrix values, you can use: * vector of size N to enter diagonal matrix * vector of size (N+1)*N/2 to enter symmetric matrix (upper triangle, row by row) * scalar to enter multiple of the unit matrix.

record: **FieldPython** implements abstract type: Field:R3 → Real[3,3]

R3 → Real[3,3] Field given by a Python script.

`TYPE` = *<selection: Field:R3 → Real[3,3]_TYPE_selection>*

Default: FieldPython []

Sub-record selection.

`script_string` = *<String (generic)>*

Default: "Obligatory if 'script_file' is not given." []

Python script given as in place string

`script_file` = *<input file name>*

Default: "Obligatory if 'script_striong' is not given." []

Python script given as external file

`function` = *<String (generic)>*

Default: *<obligatory>* []

Function in the given script that returns tuple containing components of the return type. For NxM tensor values: tensor(row,col) = tuple(M*row + col).

record: **FieldFormula** implements abstract type: Field:R3 → Real[3,3]

R3 → Real[3,3] Field given by runtime interpreted formula.

`TYPE` = *<selection: Field:R3 → Real[3,3]_TYPE_selection>*

Default: FieldFormula []

Sub-record selection.

`value` = *<Array [1,] of Array [1,] of String (generic)>*

Default: *<obligatory>* []

String, array of strings, or matrix of strings with formulas for individual entries of scalar, vector, or tensor value respectively. For vector values, you can use just one string to enter homogeneous vector. For square NxN-matrix values, you can use: * array of strings of size N to enter diagonal matrix * array of strings of size (N+1)*N/2 to enter symmetric matrix (upper triangle, row by row) * just one string to enter (spatially variable) multiple of the unit matrix. Formula can contain variables x,y,z,t and usual operators and functions.

record: **FieldElementwise** implements abstract type: Field:R3 → Real[3,3]

R3 → Real[3,3] Field constant in space.

`TYPE` = *<selection: Field:R3 → Real[3,3]_TYPE_selection>*

Default: FieldElementwise []

Sub-record selection.

`gmsh_file` = *<input file name>*

Default: *<obligatory>* []

Input file with ASCII GMSH file format.

`field_name` = *<String (generic)>*

Default: *<obligatory>* []

The values of the Field are read from the $ElementData section with field name given by this key.

---

record: **FieldInterpolatedP0** implements abstract type: Field:R3 → Real[3,3]

Field given by P0 data on another mesh. Currently defined only on boundary.

`TYPE` = *<selection: Field:R3 → Real[3,3]_TYPE_selection>*

Default: FieldInterpolatedP0 []

Sub-record selection.

`mesh` = *<input file name>*

Default: *<obligatory>* []

File with the mesh from which we interpolate. (currently only GMSH supported)

`raw_data` = *<input file name>*

Default: *<obligatory>* []

File with raw output from flow calculation. Currently we can interpolate only pressure.

---

abstract type: **Field:R3 → Real** default descendant: FieldConstant

---

Descendants:

Abstract record for all time-space functions.

`FieldConstant`

`FieldFormula`

`FieldPython`

`FieldElementwise`

---

record: **FieldConstant** implements abstract type: Field:R3 → Real constructible from key: value

---

R3 → Real Field constant in space.

`TYPE` = *<selection: Field:R3 → Real_TYPE_selection>*

Default: FieldConstant []

Sub-record selection.

`value` = *<Double >*

Default: <*obligatory*> []

Value of the constant field. For vector values, you can use scalar value to enter constant vector. For square NxN-matrix values, you can use: * vector of size N to enter diagonal matrix * vector of size (N+1)*N/2 to enter symmetric matrix (upper triangle, row by row) * scalar to enter multiple of the unit matrix.

---

record: **FieldFormula** implements abstract type: Field:R3 → Real

---

R3 → Real Field given by runtime interpreted formula.

`TYPE` = <*selection: Field:R3 → Real_TYPE_selection*>

Default: FieldFormula []

Sub-record selection.

`value` = <*String (generic)*>

Default: <*obligatory*> []

String, array of strings, or matrix of strings with formulas for individual entries of scalar, vector, or tensor value respectively. For vector values, you can use just one string to enter homogeneous vector. For square NxN-matrix values, you can use: * array of strings of size N to enter diagonal matrix * array of strings of size (N+1)*N/2 to enter symmetric matrix (upper triangle, row by row) * just one string to enter (spatially variable) multiple of the unit matrix. Formula can contain variables x,y,z,t and usual operators and functions.

---

record: **FieldPython** implements abstract type: Field:R3 → Real

---

R3 → Real Field given by a Python script.

`TYPE` = <*selection: Field:R3 → Real_TYPE_selection*>

Default: FieldPython []

Sub-record selection.

`script_string` = <*String (generic)*>

Default: "Obligatory if 'script_file' is not given." []

Python script given as in place string

`script_file` = <*input file name*>

Default: "Obligatory if 'script_striong' is not given." []

Python script given as external file

`function` = <*String (generic)*>

Default: <*obligatory*> []

Function in the given script that returns tuple containing components of the return type. For NxM tensor values: tensor(row,col) = tuple(M*row + col).

---

record: **FieldElementwise** implements abstract type: Field:R3 → Real

---

R3 → Real Field constant in space.

`TYPE` = *<selection: Field:R3 → Real_TYPE_selection>*

Default: FieldElementwise []

Sub-record selection.

`gmsh_file` = *<input file name>*

Default: *<obligatory>* []

Input file with ASCII GMSH file format.

`field_name` = *<String (generic)>*

Default: *<obligatory>* []

The values of the Field are read from the $ElementData section with field name given by this key.

---

record: **Unsteady_MH** implements abstract type: DarcyFlowMH

Mixed-Hybrid solver for unsteady saturated Darcy flow.

`TYPE` = *<selection: DarcyFlowMH_TYPE_selection>*

Default: Unsteady_MH []

Sub-record selection.

`n_schurs` = *<Integer [0, 2]>*

Default: 2 []

Number of Schur complements to perform when solving MH sytem.

`solver` = *<abstract type: Solver>*

Default: *<obligatory>* []

Linear solver for MH problem.

`output` = *<record: DarcyMHOutput>*

Default: *<obligatory>* []

Parameters of output form MH module.

`mortar_method` = *<selection: MH_MortarMethod>*

Default: None []

Method for coupling Darcy flow between dimensions.

`mortar_sigma` = *<Double [0,]>*

Default: 1.0 []

Conductivity between dimensions.

`time` = *<record: TimeGovernor>*

Default: *<obligatory>* []

Time governor setting for the unsteady Darcy flow model.

`bc_data` = <*Array of record: DarcyFlowMH_Steady_BoundaryData*>

Default: <*obligatory*> []

`bulk_data` = <*Array of record: DarcyFlowMH_Steady_BulkData*>

Default: <*obligatory*> []

---

record: **DarcyFlowMH_Steady_BoundaryData**

Record to set BOUNDARY fields of the equation 'DarcyFlowMH_Steady'. The fields are set only on the domain specified by one of the keys: 'region', 'rid', 'r_set' and after the time given by the key 'time'. The field setting can be overridden by any DarcyFlowMH_Steady_BoundaryData record that comes later in the boundary data array.

`r_set` = <*String (generic)*>

Default: <*optional*> []

Name of region set where to set fields.

`region` = <*String (generic)*>

Default: <*optional*> []

Label of the region where to set fields.

`rid` = <*Integer [0,]*>

Default: <*optional*> []

ID of the region where to set fields.

`time` = <*Double [0,]*>

Default: 0.0 []

Apply field setting in this record after this time. These times has to form an increasing sequence.

`bc_type` = <*abstract type: Field:R3 → Enum*>

Default: <*optional*> []

Boundary condition type, possible values:

`bc_pressure` = <*abstract type: Field:R3 → Real*>

Default: <*optional*> []

Dirichlet BC condition value for pressure.

`bc_flux` = <*abstract type: Field:R3 → Real*>

Default: <*optional*> []

Flux in Neumman or Robin boundary condition.

`bc_robin_sigma` = <*abstract type: Field:R3 → Real*>

Default: <*optional*> []

Conductivity coefficient in Robin boundary condition.

`bc_piezo_head` = <*abstract type: Field:R3 → Real*>

Default: <*optional*> []

Boundary condition for piezometric head.

`flow_old_bcd_file` = <*input file name*>

Default: <*optional*> []

---

abstract type: **Field:R3 → Enum** default descendant: FieldConstant

---

Descendants:

Abstract record for all time-space functions.

`FieldConstant`

`FieldFormula`

`FieldPython`

`FieldElementwise`

---

record: **FieldConstant** implements abstract type: Field:R3 → Enum constructible from key: value

---

R3 → Enum Field constant in space.

`TYPE` = <*selection: Field:R3 → Enum_TYPE_selection*>

Default: FieldConstant []

Sub-record selection.

`value` = <*selection: EqData_bc_Type*>

Default: <*obligatory*> []

Value of the constant field. For vector values, you can use scalar value to enter constant vector. For square NxN-matrix values, you can use: * vector of size N to enter diagonal matrix * vector of size (N+1)*N/2 to enter symmetric matrix (upper triangle, row by row) * scalar to enter multiple of the unit matrix.

---

record: **FieldFormula** implements abstract type: Field:R3 → Enum

---

R3 → Enum Field given by runtime interpreted formula.

`TYPE` = <*selection: Field:R3 → Enum_TYPE_selection*>

Default: FieldFormula []

Sub-record selection.

`value` = <*String (generic)*>

Default: <*obligatory*> []

String, array of strings, or matrix of strings with formulas for individual entries of scalar, vector, or tensor value respectively. For vector values, you can use just one string to enter homogeneous vector. For square NxN-matrix values, you can

use: * array of strings of size N to enter diagonal matrix * array of strings of size (N+1)*N/2 to enter symmetric matrix (upper triangle, row by row) * just one string to enter (spatially variable) multiple of the unit matrix. Formula can contain variables x,y,z,t and usual operators and functions.

---

record: **FieldPython** implements abstract type: Field:R3 → Enum

---

R3 → Enum Field given by a Python script.

`TYPE` = *<selection: Field:R3 → Enum_TYPE_selection>*

Default: FieldPython []

Sub-record selection.

`script_string` = *<String (generic)>*

Default: "Obligatory if 'script_file' is not given." []

Python script given as in place string

`script_file` = *<input file name>*

Default: "Obligatory if 'script_striong' is not given." []

Python script given as external file

`function` = *<String (generic)>*

Default: *<obligatory>* []

Function in the given script that returns tuple containing components of the return type. For NxM tensor values: tensor(row,col) = tuple(M*row + col).

---

record: **FieldElementwise** implements abstract type: Field:R3 → Enum

---

R3 → Enum Field constant in space.

`TYPE` = *<selection: Field:R3 → Enum_TYPE_selection>*

Default: FieldElementwise []

Sub-record selection.

`gmsh_file` = *<input file name>*

Default: *<obligatory>* []

Input file with ASCII GMSH file format.

`field_name` = *<String (generic)>*

Default: *<obligatory>* []

The values of the Field are read from the $ElementData section with field name given by this key.

---

abstract type: **Field:R3 → Real** default descendant: FieldConstant

---

Descendants:

Abstract record for all time-space functions.

FieldConstant

FieldFormula

FieldPython

FieldElementwise

---

record: **FieldConstant** implements abstract type: Field:R3 → Real constructible from key: value

---

R3 → Real Field constant in space.

`TYPE` = *<selection: Field:R3 → Real_TYPE_selection>*

Default: FieldConstant []

Sub-record selection.

`value` = *<Double >*

Default: *<obligatory>* []

Value of the constant field. For vector values, you can use scalar value to enter constant vector. For square NxN-matrix values, you can use: * vector of size N to enter diagonal matrix * vector of size (N+1)*N/2 to enter symmetric matrix (upper triangle, row by row) * scalar to enter multiple of the unit matrix.

---

record: **FieldFormula** implements abstract type: Field:R3 → Real

---

R3 → Real Field given by runtime interpreted formula.

`TYPE` = *<selection: Field:R3 → Real_TYPE_selection>*

Default: FieldFormula []

Sub-record selection.

`value` = *<String (generic)>*

Default: *<obligatory>* []

String, array of strings, or matrix of strings with formulas for individual entries of scalar, vector, or tensor value respectively. For vector values, you can use just one string to enter homogeneous vector. For square NxN-matrix values, you can use: * array of strings of size N to enter diagonal matrix * array of strings of size (N+1)*N/2 to enter symmetric matrix (upper triangle, row by row) * just one string to enter (spatially variable) multiple of the unit matrix. Formula can contain variables x,y,z,t and usual operators and functions.

---

record: **FieldPython** implements abstract type: Field:R3 → Real

---

R3 → Real Field given by a Python script.

`TYPE` = *<selection: Field:R3 → Real_TYPE_selection>*

Default: FieldPython []

Sub-record selection.

`script_string` = *<String (generic)>*

Default: "Obligatory if 'script_file' is not given." []

Python script given as in place string

`script_file` = *<input file name>*

Default: "Obligatory if 'script_striong' is not given." []

Python script given as external file

`function` = *<String (generic)>*

Default: *<obligatory>* []

Function in the given script that returns tuple containing components of the return type. For NxM tensor values: tensor(row,col) = tuple(M*row + col).

---

record: **FieldElementwise** implements abstract type: Field:R3 → Real

---

R3 → Real Field constant in space.

`TYPE` = *<selection: Field:R3 → Real_TYPE_selection>*

Default: FieldElementwise []

Sub-record selection.

`gmsh_file` = *<input file name>*

Default: *<obligatory>* []

Input file with ASCII GMSH file format.

`field_name` = *<String (generic)>*

Default: *<obligatory>* []

The values of the Field are read from the $ElementData section with field name given by this key.

---

record: **DarcyFlowMH_Steady_BulkData**

Record to set BULK fields of the equation 'DarcyFlowMH_Steady'. The fields are set only on the domain specified by one of the keys: 'region', 'rid', 'r_set' and after the time given by the key 'time'. The field setting can be overridden by any DarcyFlowMH_Steady_BulkData record that comes later in the bulk data array.

`r_set` = *<String (generic)>*

Default: *<optional>* []

Name of region set where to set fields.

`region` = *<String (generic)>*

Default: *<optional>* []

Label of the region where to set fields.

`rid` = <*Integer [0,]*>

Default: <*optional*> []

ID of the region where to set fields.

`time` = <*Double [0,]*>

Default: 0.0 []

Apply field setting in this record after this time. These times has to form an increasing sequence.

`cond_anisothropy` = <*abstract type: Field:R3 → Real[3,3]*>

Default: <*optional*> []

Anisothropic conductivity tensor.

`cross_section` = <*abstract type: Field:R3 → Real*>

Default: <*optional*> []

Complement dimension parameter (cross section for 1D, thickness for 2D).

`conductivity` = <*abstract type: Field:R3 → Real*>

Default: <*optional*> []

Isothropic conductivity scalar.

`water_source_density` = <*abstract type: Field:R3 → Real*>

Default: <*optional*> []

Water source density.

`init_pressure` = <*abstract type: Field:R3 → Real*>

Default: <*optional*> []

Initial condition as pressure

`storativity` = <*abstract type: Field:R3 → Real*>

Default: <*optional*> []

Storativity.

`init_piezo_head` = <*abstract type: Field:R3 → Real*>

Default: <*optional*> []

Initial piezometric head.

---

abstract type: **Field:R3 → Real** default descendant: FieldConstant

---

Descendants:

Abstract record for all time-space functions.

`FieldConstant`

`FieldFormula`

`FieldPython`

FieldElementwise

---

record: **FieldConstant** implements abstract type: Field:R3 → Real constructible from key: value

---

R3 → Real Field constant in space.

`TYPE` = *<selection: Field:R3 → Real_TYPE_selection>*

Default: FieldConstant []

Sub-record selection.

`value` = *<Double >*

Default: *<obligatory>* []

Value of the constant field. For vector values, you can use scalar value to enter constant vector. For square NxN-matrix values, you can use: * vector of size N to enter diagonal matrix * vector of size (N+1)*N/2 to enter symmetric matrix (upper triangle, row by row) * scalar to enter multiple of the unit matrix.

---

record: **FieldFormula** implements abstract type: Field:R3 → Real

---

R3 → Real Field given by runtime interpreted formula.

`TYPE` = *<selection: Field:R3 → Real_TYPE_selection>*

Default: FieldFormula []

Sub-record selection.

`value` = *<String (generic)>*

Default: *<obligatory>* []

String, array of strings, or matrix of strings with formulas for individual entries of scalar, vector, or tensor value respectively. For vector values, you can use just one string to enter homogeneous vector. For square NxN-matrix values, you can use: * array of strings of size N to enter diagonal matrix * array of strings of size (N+1)*N/2 to enter symmetric matrix (upper triangle, row by row) * just one string to enter (spatially variable) multiple of the unit matrix. Formula can contain variables x,y,z,t and usual operators and functions.

---

record: **FieldPython** implements abstract type: Field:R3 → Real

---

R3 → Real Field given by a Python script.

`TYPE` = *<selection: Field:R3 → Real_TYPE_selection>*

Default: FieldPython []

Sub-record selection.

`script_string` = *<String (generic)>*

Default: "Obligatory if 'script_file' is not given." []

Python script given as in place string

`script_file` = *<input file name>*

Default: "Obligatory if 'script_striong' is not given." []

Python script given as external file

`function` = *<String (generic)>*

Default: *<obligatory>* []

Function in the given script that returns tuple containing components of the return type. For NxM tensor values: tensor(row,col) = tuple(M*row + col).

---

record: **FieldElementwise** implements abstract type: Field:R3 → Real

---

R3 → Real Field constant in space.

`TYPE` = *<selection: Field:R3 → Real_TYPE_selection>*

Default: FieldElementwise []

Sub-record selection.

`gmsh_file` = *<input file name>*

Default: *<obligatory>* []

Input file with ASCII GMSH file format.

`field_name` = *<String (generic)>*

Default: *<obligatory>* []

The values of the Field are read from the $ElementData section with field name given by this key.

---

record: **Unsteady_LMH** implements abstract type: DarcyFlowMH

Lumped Mixed-Hybrid solver for unsteady saturated Darcy flow.

`TYPE` = *<selection: DarcyFlowMH_TYPE_selection>*

Default: Unsteady_LMH []

Sub-record selection.

`n_schurs` = *<Integer [0, 2]>*

Default: 2 []

Number of Schur complements to perform when solving MH sytem.

`solver` = *<abstract type: Solver>*

Default: *<obligatory>* []

Linear solver for MH problem.

`output` = *<record: DarcyMHOutput>*

Default: *<obligatory>* []

Parameters of output form MH module.

`mortar_method` = <*selection:* *MH_MortarMethod*>

Default: None []

Method for coupling Darcy flow between dimensions.

`mortar_sigma` = <*Double [0,]*>

Default: 1.0 []

Conductivity between dimensions.

`time` = <*record:* *TimeGovernor*>

Default: <*obligatory*> []

Time governor setting for the unsteady Darcy flow model.

`bc_data` = <*Array of record:* *DarcyFlowMH_Steady_BoundaryData*>

Default: <*obligatory*> []

`bulk_data` = <*Array of record:* *DarcyFlowMH_Steady_BulkData*>

Default: <*obligatory*> []

---

record: **DarcyFlowMH_Steady_BoundaryData**

Record to set BOUNDARY fields of the equation 'DarcyFlowMH_Steady'. The fields are set only on the domain specified by one of the keys: 'region', 'rid', 'r_set' and after the time given by the key 'time'. The field setting can be overridden by any DarcyFlowMH_Steady_BoundaryData record that comes later in the boundary data array.

`r_set` = <*String (generic)*>

Default: <*optional*> []

Name of region set where to set fields.

`region` = <*String (generic)*>

Default: <*optional*> []

Label of the region where to set fields.

`rid` = <*Integer [0,]*>

Default: <*optional*> []

ID of the region where to set fields.

`time` = <*Double [0,]*>

Default: 0.0 []

Apply field setting in this record after this time. These times has to form an increasing sequence.

`bc_type` = <*abstract type:* *Field:R3 → Enum*>

Default: <*optional*> []

Boundary condition type, possible values:

`bc_pressure` = <*abstract type: Field:R3 → Real*>

Default: <*optional*> []

Dirichlet BC condition value for pressure.

`bc_flux` = <*abstract type: Field:R3 → Real*>

Default: <*optional*> []

Flux in Neumman or Robin boundary condition.

`bc_robin_sigma` = <*abstract type: Field:R3 → Real*>

Default: <*optional*> []

Conductivity coefficient in Robin boundary condition.

`bc_piezo_head` = <*abstract type: Field:R3 → Real*>

Default: <*optional*> []

Boundary condition for piezometric head.

`flow_old_bcd_file` = <*input file name*>

Default: <*optional*> []

---

abstract type: **Field:R3 → Enum** default descendant: FieldConstant

---

Descendants:

Abstract record for all time-space functions.

`FieldConstant`

`FieldFormula`

`FieldPython`

`FieldElementwise`

---

record: **FieldConstant** implements abstract type: Field:R3 → Enum constructible from key: value

---

R3 → Enum Field constant in space.

`TYPE` = <*selection: Field:R3 → Enum_TYPE_selection*>

Default: FieldConstant []

Sub-record selection.

`value` = <*selection: EqData_bc_Type*>

Default: <*obligatory*> []

Value of the constant field. For vector values, you can use scalar value to enter constant vector. For square NxN-matrix values, you can use: * vector of size N to enter diagonal matrix * vector of size (N+1)*N/2 to enter symmetric matrix (upper triangle, row by row) * scalar to enter multiple of the unit matrix.

---

record: **FieldFormula** implements abstract type: Field:R3 → Enum

R3 → Enum Field given by runtime interpreted formula.

`TYPE` = *<selection: Field:R3 → Enum_TYPE_selection>*

Default: FieldFormula []

Sub-record selection.

`value` = *<String (generic)>*

Default: *<obligatory>* []

String, array of strings, or matrix of strings with formulas for individual entries of scalar, vector, or tensor value respectively. For vector values, you can use just one string to enter homogeneous vector. For square NxN-matrix values, you can use: * array of strings of size N to enter diagonal matrix * array of strings of size (N+1)*N/2 to enter symmetric matrix (upper triangle, row by row) * just one string to enter (spatially variable) multiple of the unit matrix. Formula can contain variables x,y,z,t and usual operators and functions.

record: **FieldPython** implements abstract type: Field:R3 → Enum

R3 → Enum Field given by a Python script.

`TYPE` = *<selection: Field:R3 → Enum_TYPE_selection>*

Default: FieldPython []

Sub-record selection.

`script_string` = *<String (generic)>*

Default: "Obligatory if 'script_file' is not given." []

Python script given as in place string

`script_file` = *<input file name>*

Default: "Obligatory if 'script_striong' is not given." []

Python script given as external file

`function` = *<String (generic)>*

Default: *<obligatory>* []

Function in the given script that returns tuple containing components of the return type. For NxM tensor values: tensor(row,col) = tuple(M*row + col).

record: **FieldElementwise** implements abstract type: Field:R3 → Enum

R3 → Enum Field constant in space.

`TYPE` = *<selection: Field:R3 → Enum_TYPE_selection>*

Default: FieldElementwise []

Sub-record selection.

`gmsh_file` = *<input file name>*

Default: <*obligatory*> []

Input file with ASCII GMSH file format.

`field_name` = <*String (generic)*>

Default: <*obligatory*> []

The values of the Field are read from the $ElementData section with field name given by this key.

---

abstract type: **Field:R3 → Real** default descendant: FieldConstant

---

Descendants:

Abstract record for all time-space functions.

`FieldConstant`

`FieldFormula`

`FieldPython`

`FieldElementwise`

---

record: **FieldConstant** implements abstract type: Field:R3 → Real constructible from key: value

---

R3 → Real Field constant in space.

`TYPE` = <*selection: Field:R3 → Real_TYPE_selection*>

Default: FieldConstant []

Sub-record selection.

`value` = <*Double* >

Default: <*obligatory*> []

Value of the constant field. For vector values, you can use scalar value to enter constant vector. For square NxN-matrix values, you can use: * vector of size N to enter diagonal matrix * vector of size (N+1)*N/2 to enter symmetric matrix (upper triangle, row by row) * scalar to enter multiple of the unit matrix.

---

record: **FieldFormula** implements abstract type: Field:R3 → Real

---

R3 → Real Field given by runtime interpreted formula.

`TYPE` = <*selection: Field:R3 → Real_TYPE_selection*>

Default: FieldFormula []

Sub-record selection.

`value` = <*String (generic)*>

Default: <*obligatory*> []

String, array of strings, or matrix of strings with formulas for individual entries of scalar, vector, or tensor value respectively. For vector values, you can use just

one string to enter homogeneous vector. For square NxN-matrix values, you can use: * array of strings of size N to enter diagonal matrix * array of strings of size (N+1)*N/2 to enter symmetric matrix (upper triangle, row by row) * just one string to enter (spatially variable) multiple of the unit matrix. Formula can contain variables x,y,z,t and usual operators and functions.

---

record: **FieldPython** implements abstract type: Field:R3 → Real

---

R3 → Real Field given by a Python script.

`TYPE` = <*selection: Field:R3 → Real_TYPE_selection*>

Default: FieldPython []

Sub-record selection.

`script_string` = <*String (generic)*>

Default: "Obligatory if 'script_file' is not given." []

Python script given as in place string

`script_file` = <*input file name*>

Default: "Obligatory if 'script_striong' is not given." []

Python script given as external file

`function` = <*String (generic)*>

Default: <*obligatory*> []

Function in the given script that returns tuple containing components of the return type. For NxM tensor values: tensor(row,col) = tuple(M*row + col).

---

record: **FieldElementwise** implements abstract type: Field:R3 → Real

---

R3 → Real Field constant in space.

`TYPE` = <*selection: Field:R3 → Real_TYPE_selection*>

Default: FieldElementwise []

Sub-record selection.

`gmsh_file` = <*input file name*>

Default: <*obligatory*> []

Input file with ASCII GMSH file format.

`field_name` = <*String (generic)*>

Default: <*obligatory*> []

The values of the Field are read from the $ElementData section with field name given by this key.

---

record: **DarcyFlowMH_Steady_BulkData**

Record to set BULK fields of the equation 'DarcyFlowMH_Steady'. The fields

are set only on the domain specified by one of the keys: 'region', 'rid', 'r_set' and after the time given by the key 'time'. The field setting can be overridden by any DarcyFlowMH_Steady_BulkData record that comes later in the bulk data array.

`r_set` = *<String (generic)>*

Default: *<optional>* []

Name of region set where to set fields.

`region` = *<String (generic)>*

Default: *<optional>* []

Label of the region where to set fields.

`rid` = *<Integer [0,]>*

Default: *<optional>* []

ID of the region where to set fields.

`time` = *<Double [0,]>*

Default: 0.0 []

Apply field setting in this record after this time. These times has to form an increasing sequence.

`cond_anisothropy` = *<abstract type: Field:R3 → Real[3,3]>*

Default: *<optional>* []

Anisothropic conductivity tensor.

`cross_section` = *<abstract type: Field:R3 → Real>*

Default: *<optional>* []

Complement dimension parameter (cross section for 1D, thickness for 2D).

`conductivity` = *<abstract type: Field:R3 → Real>*

Default: *<optional>* []

Isothropic conductivity scalar.

`water_source_density` = *<abstract type: Field:R3 → Real>*

Default: *<optional>* []

Water source density.

`init_pressure` = *<abstract type: Field:R3 → Real>*

Default: *<optional>* []

Initial condition as pressure

`storativity` = *<abstract type: Field:R3 → Real>*

Default: *<optional>* []

Storativity.

`init_piezo_head` = *<abstract type: Field:R3 → Real>*

Default: <*optional*> []

Initial piezometric head.

---

abstract type: **Field:R3 → Real** default descendant: FieldConstant

---

Descendants:

Abstract record for all time-space functions.

`FieldConstant`

`FieldFormula`

`FieldPython`

`FieldElementwise`

---

record: **FieldConstant** implements abstract type: Field:R3 → Real constructible from key: value

---

R3 → Real Field constant in space.

`TYPE` = <*selection: Field:R3 → Real_TYPE_selection*>

Default: FieldConstant []

Sub-record selection.

`value` = <*Double* >

Default: <*obligatory*> []

Value of the constant field. For vector values, you can use scalar value to enter constant vector. For square NxN-matrix values, you can use: * vector of size N to enter diagonal matrix * vector of size (N+1)*N/2 to enter symmetric matrix (upper triangle, row by row) * scalar to enter multiple of the unit matrix.

---

record: **FieldFormula** implements abstract type: Field:R3 → Real

---

R3 → Real Field given by runtime interpreted formula.

`TYPE` = <*selection: Field:R3 → Real_TYPE_selection*>

Default: FieldFormula []

Sub-record selection.

`value` = <*String (generic)*>

Default: <*obligatory*> []

String, array of strings, or matrix of strings with formulas for individual entries of scalar, vector, or tensor value respectively. For vector values, you can use just one string to enter homogeneous vector. For square NxN-matrix values, you can use: * array of strings of size N to enter diagonal matrix * array of strings of size (N+1)*N/2 to enter symmetric matrix (upper triangle, row by row) * just one string to enter (spatially variable) multiple of the unit matrix. Formula can contain variables x,y,z,t and usual operators and functions.

record: **FieldPython** implements abstract type: Field:R3 → Real

R3 → Real Field given by a Python script.

`TYPE` = *<selection: Field:R3 → Real_TYPE_selection>*

Default: FieldPython []

Sub-record selection.

`script_string` = *<String (generic)>*

Default: "Obligatory if 'script_file' is not given." []

Python script given as in place string

`script_file` = *<input file name>*

Default: "Obligatory if 'script_striong' is not given." []

Python script given as external file

`function` = *<String (generic)>*

Default: *<obligatory>* []

Function in the given script that returns tuple containing components of the return type. For NxM tensor values: tensor(row,col) = tuple(M*row + col).

record: **FieldElementwise** implements abstract type: Field:R3 → Real

R3 → Real Field constant in space.

`TYPE` = *<selection: Field:R3 → Real_TYPE_selection>*

Default: FieldElementwise []

Sub-record selection.

`gmsh_file` = *<input file name>*

Default: *<obligatory>* []

Input file with ASCII GMSH file format.

`field_name` = *<String (generic)>*

Default: *<obligatory>* []

The values of the Field are read from the $ElementData section with field name given by this key.

abstract type: **Transport**

Descendants:

Secondary equation for transport of substances.

`TransportOperatorSplitting`

`AdvectionDiffusion_DG`

record: **TransportOperatorSplitting** implements abstract type: Transport

Explicit FVM transport (no diffusion) coupled with reaction and sorption model (ODE per element) via. operator splitting.

`TYPE` = <*selection: Transport_TYPE_selection*>

Default: TransportOperatorSplitting []

Sub-record selection.

`time` = <*record: TimeGovernor*>

Default: <*obligatory*> []

Time governor setting for the transport model.

`substances` = <*Array of String (generic)*>

Default: <*obligatory*> []

Names of transported substances.

`sorption_enable` = <*Bool*>

Default: false []

Model of sorption.

`dual_porosity` = <*Bool*>

Default: false []

Dual porosity model.

`sources_file` = <*input file name*>

Default: <*optional*> []

File with data for the source term in the transport equation.

`output` = <*record: TransportOutput*>

Default: <*obligatory*> []

Parameters of output stream.

`reactions` = <*abstract type: Reactions*>

Default: <*optional*> []

Initialization of per element reactions.

`bc_data` = <*Array of record: TransportOperatorSplitting_BoundaryData*>

Default: <*obligatory*> []

`bulk_data` = <*Array of record: TransportOperatorSplitting_BulkData*>

Default: <*obligatory*> []

record: **TransportOutput**

Output setting for transport equations.

`output_stream` = <*record: OutputStrem*>

Default: <*obligatory*> []

Parameters of output stream.

`save_step` = <*Double [0,]*>

Default: <*obligatory*> []

Interval between outputs.

`output_times` = <*Array of Double [0,]*>

Default: <*optional*> []

Explicit array of output times (can be combined with 'save_step'.

`conc_mobile_p0` = <*String (generic)*>

Default: <*optional*> []

Name of output stream for P0 approximation of the concentration in mobile phase.

`conc_immobile_p0` = <*String (generic)*>

Default: <*optional*> []

Name of output stream for P0 approximation of the concentration in immobile phase.

`conc_mobile_sorbed_p0` = <*String (generic)*>

Default: <*optional*> []

Name of output stream for P0 approximation of the surface concentration of sorbed mobile phase.

`conc_immobile_sorbed_p0` = <*String (generic)*>

Default: <*optional*> []

Name of output stream for P0 approximation of the surface concentration of sorbed immobile phase.

---

abstract type: **Reactions**

Descendants:

Equation for reading information about simple chemical reactions.

`LinearReactions`

`PadeApproximant`

`Isotope`

---

record: **LinearReactions** implements abstract type: Reactions

---

Information for a decision about the way to simulate radioactive decay.

`TYPE` = <*selection: Reactions_TYPE_selection*>

Default: LinearReactions []

Sub-record selection.

`decays` = <*Array of record: Substep*>

Default: <*obligatory*> []

Description of particular decay chain substeps.

`matrix_exp_on` = <*Bool*>

Default: false []

Enables to use Pade approximant of matrix exponential.

---

record: **Substep**

Equation for reading information about radioactive decays.

`parent` = <*String (generic)*>

Default: <*obligatory*> []

Identifier of an isotope.

`half_life` = <*Double* >

Default: <*optional*> []

Half life of the parent substance.

`kinetic` = <*Double* >

Default: <*optional*> []

Kinetic constants describing first order reactions.

`products` = <*Array of String (generic)*>

Default: <*obligatory*> []

Identifies isotopes which decays parental atom to.

`branch_ratios` = <*Array of Double* >

Default: 1.0 []

Decay chain branching percentage.

---

record: **PadeApproximant** implements abstract type: Reactions

---

Abstract record with an information about pade approximant parameters.

`TYPE` = <*selection: Reactions_TYPE_selection*>

Default: PadeApproximant []

Sub-record selection.

`decays` = <*Array of record: Substep*>

Default: <*obligatory*> []

Description of particular decay chain substeps.

`nom_pol_deg` = <*Integer* >

Default: 2 []

Polynomial degree of the nominator of Pade approximant.

`den_pol_deg` = <*Integer* >

Default: 2 []

Polynomial degree of the nominator of Pade approximant

---

record: **Substep**

Equation for reading information about radioactive decays.

`parent` = <*String (generic)*>

Default: <*obligatory*> []

Identifier of an isotope.

`half_life` = <*Double* >

Default: <*optional*> []

Half life of the parent substance.

`kinetic` = <*Double* >

Default: <*optional*> []

Kinetic constants describing first order reactions.

`products` = <*Array of String (generic)*>

Default: <*obligatory*> []

Identifies isotopes which decays parental atom to.

`branch_ratios` = <*Array of Double* >

Default: 1.0 []

Decay chain branching percentage.

---

record: **Isotope** implements abstract type: Reactions

Definition of information about a single isotope.

`TYPE` = <*selection: Reactions_TYPE_selection*>

Default: Isotope []

Sub-record selection.

`identifier` = <*Integer* >

Default: <*obligatory*> []

Identifier of the isotope.

`half_life` = <*Double* >

Default: <*obligatory*> []

Half life parameter.

record: **TransportOperatorSplitting_BoundaryData**

Record to set BOUNDARY fields of the equation 'TransportOperatorSplitting'. The fields are set only on the domain specified by one of the keys: 'region', 'rid', 'r_set' and after the time given by the key 'time'. The field setting can be overridden by any TransportOperatorSplitting_BoundaryData record that comes later in the boundary data array.

`r_set` = *<String (generic)>*

Default: *<optional>* []

Name of region set where to set fields.

`region` = *<String (generic)>*

Default: *<optional>* []

Label of the region where to set fields.

`rid` = *<Integer [0,]>*

Default: *<optional>* []

ID of the region where to set fields.

`time` = *<Double [0,]>*

Default: 0.0 []

Apply field setting in this record after this time. These times has to form an increasing sequence.

`bc_conc` = *<abstract type: Field:R3 → Real[n]>*

Default: *<optional>* []

Boundary conditions for concentrations.

`old_boundary_file` = *<input file name>*

Default: *<optional>* []

Input file with boundary conditions (obsolete).

`bc_times` = *<Array of Double >*

Default: *<optional>* []

Times for changing the boundary conditions (obsolete).

abstract type: **Field:R3 → Real[n]** default descendant: FieldConstant

Descendants:

Abstract record for all time-space functions.

`FieldConstant`

`FieldPython`

`FieldFormula`

FieldElementwise

FieldInterpolatedP0

---

record: **FieldConstant** implements abstract type: Field:R3 → Real[n] constructible from key: value

---

R3 → Real[n] Field constant in space.

`TYPE` = <*selection: Field:R3 → Real[n]_TYPE_selection*>

Default: FieldConstant []

Sub-record selection.

`value` = <*Array [1,] of Double* >

Default: <*obligatory*> []

Value of the constant field. For vector values, you can use scalar value to enter constant vector. For square NxN-matrix values, you can use: * vector of size N to enter diagonal matrix * vector of size (N+1)*N/2 to enter symmetric matrix (upper triangle, row by row) * scalar to enter multiple of the unit matrix.

---

record: **FieldPython** implements abstract type: Field:R3 → Real[n]

R3 → Real[n] Field given by a Python script.

`TYPE` = <*selection: Field:R3 → Real[n]_TYPE_selection*>

Default: FieldPython []

Sub-record selection.

`script_string` = <*String (generic)*>

Default: "Obligatory if 'script_file' is not given." []

Python script given as in place string

`script_file` = <*input file name*>

Default: "Obligatory if 'script_striong' is not given." []

Python script given as external file

`function` = <*String (generic)*>

Default: <*obligatory*> []

Function in the given script that returns tuple containing components of the return type. For NxM tensor values: tensor(row,col) = tuple(M*row + col).

---

record: **FieldFormula** implements abstract type: Field:R3 → Real[n]

R3 → Real[n] Field given by runtime interpreted formula.

`TYPE` = <*selection: Field:R3 → Real[n]_TYPE_selection*>

Default: FieldFormula []

Sub-record selection.

`value` = *<Array [1,] of String (generic)>*

Default: *<obligatory>* []

String, array of strings, or matrix of strings with formulas for individual entries of scalar, vector, or tensor value respectively. For vector values, you can use just one string to enter homogeneous vector. For square NxN-matrix values, you can use: * array of strings of size N to enter diagonal matrix * array of strings of size (N+1)*N/2 to enter symmetric matrix (upper triangle, row by row) * just one string to enter (spatially variable) multiple of the unit matrix. Formula can contain variables x,y,z,t and usual operators and functions.

---

record: **FieldElementwise** implements abstract type: Field:R3 → Real[n]

R3 → Real[n] Field constant in space.

`TYPE` = *<selection: Field:R3 → Real[n]_TYPE_selection>*

Default: FieldElementwise []

Sub-record selection.

`gmsh_file` = *<input file name>*

Default: *<obligatory>* []

Input file with ASCII GMSH file format.

`field_name` = *<String (generic)>*

Default: *<obligatory>* []

The values of the Field are read from the $ElementData section with field name given by this key.

---

record: **FieldInterpolatedP0** implements abstract type: Field:R3 → Real[n]

Field given by P0 data on another mesh. Currently defined only on boundary.

`TYPE` = *<selection: Field:R3 → Real[n]_TYPE_selection>*

Default: FieldInterpolatedP0 []

Sub-record selection.

`mesh` = *<input file name>*

Default: *<obligatory>* []

File with the mesh from which we interpolate. (currently only GMSH supported)

`raw_data` = *<input file name>*

Default: *<obligatory>* []

File with raw output from flow calculation. Currently we can interpolate only pressure.

Record to set BULK fields of the equation 'TransportOperatorSplitting'. The fields are set only on the domain specified by one of the keys: 'region', 'rid', 'r_set' and after the time given by the key 'time'. The field setting can be overridden by any TransportOperatorSplitting_BulkData record that comes later in the bulk data array.

`r_set` = <*String (generic)*>

Default: <*optional*> []

Name of region set where to set fields.

`region` = <*String (generic)*>

Default: <*optional*> []

Label of the region where to set fields.

`rid` = <*Integer [0,]*>

Default: <*optional*> []

ID of the region where to set fields.

`time` = <*Double [0,]*>

Default: 0.0 []

Apply field setting in this record after this time. These times has to form an increasing sequence.

`init_conc` = <*abstract type: Field:R3 → Real[n]*>

Default: <*optional*> []

Initial concentrations.

`por_m` = <*abstract type: Field:R3 → Real*>

Default: <*optional*> []

Mobile porosity

`por_imm` = <*abstract type: Field:R3 → Real*>

Default: <*optional*> []

Immobile porosity

`alpha` = <*abstract type: Field:R3 → Real[n]*>

Default: <*optional*> []

Coefficients of non-equilibrium exchange.

`sorp_type` = <*abstract type: Field:R3 → Real[n]*>

Default: <*optional*> []

Type of sorption.

`sorp_coef0` = <*abstract type: Field:R3 → Real[n]*>

Default: <*optional*> []

Coefficient of sorption.

`sorp_coef1` = <*abstract type: Field:R3 → Real[n]*>

Default: <*optional*> []

Coefficient of sorption.

`phi` = <*abstract type: Field:R3 → Real*>

Default: <*optional*> []

Solid / solid mobile.

`sources_density` = <*abstract type: Field:R3 → Real[n]*>

Default: <*optional*> []

Density of transport sources.

`sources_sigma` = <*abstract type: Field:R3 → Real[n]*>

Default: <*optional*> []

`sources_conc` = <*abstract type: Field:R3 → Real[n]*>

Default: <*optional*> []

Concentration sources.

---

record: **AdvectionDiffusion_DG** implements abstract type: Transport

---

DG solver for transport with diffusion.

`TYPE` = <*selection: Transport_TYPE_selection*>

Default: AdvectionDiffusion_DG []

Sub-record selection.

`time` = <*record: TimeGovernor*>

Default: <*obligatory*> []

Time governor setting for the transport model.

`substances` = <*Array of String (generic)*>

Default: <*obligatory*> []

Names of transported substances.

`sorption_enable` = <*Bool*>

Default: false []

Model of sorption.

`dual_porosity` = <*Bool*>

Default: false []

Dual porosity model.

`sources_file` = <*input file name*>

Default: <*optional*> []

File with data for the source term in the transport equation.

`output` = *<record: TransportOutput>*

Default: *<obligatory>* []

Parameters of output stream.

`sigma` = *<Double >*

Default: 0 []

Coefficient of diffusive transfer through fractures.

`alpha_l` = *<Double >*

Default: 0 []

Longitudal dispersivity.

`alpha_t` = *<Double >*

Default: 0 []

Transversal dispersivity.

`d_m` = *<Double >*

Default: 1e-6 []

Molecular diffusivity.

`dg_penalty` = *<Double [0,]>*

Default: 0 []

Penalty parameter influencing the discontinuity of the solution.

`solver` = *<abstract type: Solver>*

Default: *<obligatory>* []

Linear solver for MH problem.

`bc_data` = *<Array of record: TransportDG_BoundaryData>*

Default: *<obligatory>* []

`bulk_data` = *<Array of record: TransportDG_BulkData>*

Default: *<obligatory>* []

---

record: **TransportDG_BoundaryData**

Record to set BOUNDARY fields of the equation 'TransportDG'. The fields are set only on the domain specified by one of the keys: 'region', 'rid', 'r_set' and after the time given by the key 'time'. The field setting can be overridden by any TransportDG_BoundaryData record that comes later in the boundary data array.

`r_set` = *<String (generic)>*

Default: *<optional>* []

Name of region set where to set fields.

`region` = *<String (generic)>*

Default: *<optional>* []

Label of the region where to set fields.

`rid` = *<Integer [0,]>*

Default: *<optional>* []

ID of the region where to set fields.

`time` = *<Double [0,]>*

Default: 0.0 []

Apply field setting in this record after this time. These times has to form an increasing sequence.

`bc_conc` = *<abstract type: Field:R3 → Real[n]>*

Default: *<optional>* []

Boundary conditions for concentrations.

`old_boundary_file` = *<input file name>*

Default: *<optional>* []

Input file with boundary conditions (obsolete).

`bc_times` = *<Array of Double >*

Default: *<optional>* []

Times for changing the boundary conditions (obsolete).

---

record: **TransportDG_BulkData**

Record to set BULK fields of the equation 'TransportDG'. The fields are set only on the domain specified by one of the keys: 'region', 'rid', 'r_set' and after the time given by the key 'time'. The field setting can be overridden by any TransportDG_BulkData record that comes later in the bulk data array.

`r_set` = *<String (generic)>*

Default: *<optional>* []

Name of region set where to set fields.

`region` = *<String (generic)>*

Default: *<optional>* []

Label of the region where to set fields.

`rid` = *<Integer [0,]>*

Default: *<optional>* []

ID of the region where to set fields.

`time` = *<Double [0,]>*

Default: 0.0 []

Apply field setting in this record after this time. These times has to form an increasing sequence.

`init_conc` = <*abstract type: Field:R3 → Real[n]*>

Default: <*optional*> []

Initial concentrations.

`por_m` = <*abstract type: Field:R3 → Real*>

Default: <*optional*> []

Mobile porosity