Technical university of Liberec

Faculty of mechatronics, informatics
and interdisciplinary studies

Flow123d

version 1.7.0

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

Liberec, 2012

Authors (of version 1.7.0)

Jan Březina, Jan Stebel, Jiří Hnídek, David Flanderka, Pavel Exner, Lukáš Zedek

Acknowledgement

This work was supported by the Technology Agency of the Czech Republic under the project no. TA01021331.

Contents

1 Quick start **4**
- 1.1 Basic usage . . . 5
  - 1.1.1 How to run the simulation. . . . 5
  - 1.1.2 Tutorial problem . . . 6

2 Mathematical models of physical reality **12**
- 2.1 Darcy flow model . . . 13
- 2.2 Transport of substances . . . 14

3 File formats **17**
- 3.1 Main input file (CON file format) . . . 17
  - 3.1.1 JSON for humans . . . 17
  - 3.1.2 CON constructs . . . 18
  - 3.1.3 CON special keys . . . 19
  - 3.1.4 Record types . . . 20
- 3.2 Important Record types of Flow123d input . . . 20
  - 3.2.1 Mesh record . . . 20
  - 3.2.2 Field records . . . 21
  - 3.2.3 Field data for equations . . . 22
- 3.3 Mesh and data file format MSH ASCII . . . 23
- 3.4 Output files . . . 24
  - 3.4.1 Output data fields of water flow module . . . 24
  - 3.4.2 Output data fields of transport . . . 25
  - 3.4.3 Auxiliary output files . . . 25

4 Main input file reference **27**

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 channels and exchange between domains of different dimensions. 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 are also models 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.con
```

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.

`--help`
: Parameters interpreted by Flow123d. Remaining parameters are passed to PETSC.

`-s, --solve` *file*
: Set principal CON input file. All relative paths in the CON file are relative against current directory.

`-i, --input_dir` *directory*
: The place holder `${INPUT}` used in the path of an input file will be replaced by given *directory*.

`-o, --output_dir` *directory*
: All paths for output files will be relative to this *directory*.

`-l, --log` *file_name*
: Set base name of log files.

`--no_log`
: Turn off logging.

`--no_profiler`
: Turn off profiler output.

`--full_doc`
: Prints full structure of the main input file.

`--JSON_template`
: Prints a description of the main input file as a valid CON file template.

`--latex_doc`
: Prints a description of the main input file in LaTeX format using particular macros.

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` together with some valid input. Options for various PETSC modules are displayed when the module is used for the first time.

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.con
```

or the parallel run by the command:

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

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 Tutorial problem

CON file format

The main input file uses a slightly extended JSON file format which together with some particular constructs forms a CON (C++ object notation) file format. Main extensions of the JSON are unquoted key names (as long as they do not contain whitespaces), possibility to use `=` instead of `:` and C++ comments, i.e. `//` for a one line and `/* */` for a multi-line comment. In CON file format, we prefer to call JSON objects "records" and we introduce also "abstract records" that mimic C++ abstract classes, arrays of a CON file have only elements of the same type (possibly using abstract record types for polymorphism). The usual keys are in lower case and without spaces (using underscores instead), there are few special upper case keys that are interpreted by the reader: `REF` key for references, `TYPE` key for specifing actual type of an abstract record. For detailed description see Section 3.1.

Geometry

In the following, we shall provide a commented input for the tutorial problem:

`tests/03_transport_small_12d/flow_vtk.con`

We consider a simple 2D problem with a branching 1D fracture (see Figure 1.1 for the geometry). To prepare a mesh file we use the GMSH software. First, we construct a geometry file. In our case the geometry consists of:

- one physical 2D domain corresponding to the whole square
- three 1D physical domains of the fracture
- four 1D boundary physical domains of the 2D domain
- three 0D boundary physical domains of the 1D domain

In this simple example, we can in fact combine physical domains in every group, however we use this more complex setting for demonstration purposes. Using GMSH graphical interface we can prepare the GEO file where physical domains are referenced by numbers, then we use any text editor and replace numbers with string labels in such a way that the labels of boundary physical domains start with the dot character. These are the domains where we will not do any calculations but we will use them for setting boundary conditions. Finally, we get the GEO file like this:

```
1  cl1 = 0.16;
2  Point(1) = {0, 1, 0, cl1};
3  Point(2) = {1, 1, 0, cl1};
4  Point(3) = {1, 0, 0, cl1};
5  Point(4) = {0, 0, 0, cl1};
6  Point(6) = {0.25, -0, 0, cl1};
7  Point(7) = {0, 0.25, 0, cl1};
8  Point(8) = {0.5, 0.5, -0, cl1};
9  Point(9) = {0.75, 1, 0, cl1};
10 Line(19) = {9, 8};
11 Line(20) = {7, 8};
12 Line(21) = {8, 6};
13 Line(22) = {2, 3};
14 Line(23) = {2, 9};
15 Line(24) = {9, 1};
16 Line(25) = {1, 7};
17 Line(26) = {7, 4};
18 Line(27) = {4, 6};
19 Line(28) = {6, 3};
20 Line Loop(30) = {20, -19, 24, 25};
21 Plane Surface(30) = {30};
22 Line Loop(32) = {23, 19, 21, 28, -22};
23 Plane Surface(32) = {32};
24 Line Loop(34) = {26, 27, -21, -20};
25 Plane Surface(34) = {34};
26 Physical Point(".1d_top") = {9};
27 Physical Point(".1d_left") = {7};
28 Physical Point(".1d_bottom") = {6};
29 Physical Line("1d_upper") = {19};
30 Physical Line("1d_lower") = {21};
31 Physical Line("1d_left_branch") = {20};
32 Physical Line(".2d_top") = {23, 24};
33 Physical Line(".2d_right") = {22};
34 Physical Line(".2d_bottom") = {27, 28};
35 Physical Line(".2d_left") = {25, 26};
36 Physical Surface("2d") = {30, 32, 34};
```

Notice the labeled physical domains on lines 26 – 36. Then we just set the discretization step `cl1` and use GMSH to create the mesh file. The mesh file contains both the 'bulk' elements where we perform calculations and the 'boundary' elements (on the boundary physical domains) where we only set the boundary conditions.

Having the computational mesh, we can create the main input file with the description of our problem.

```
{
  problem = {
    TYPE = "SequentialCoupling",
    description = "Transport 1D-2D, (convection, dual porosity, sorption)",
    mesh = {
      mesh_file = "./input/mesh_with_boundary.msh",
      sets = [
          { name="1d_domain",
            region_labels = [ "1d_upper", "1d_lower", "1d_left_branch" ]
          }
        ]
    },
```

The file starts with a particular problem type selection, currently only the type `SequentialCoupling` is supported, and a textual problem description. Next, we specify the computational mesh, here it consists of the name of the mesh file and the declaration of one *region set* composed of all 1D regions i.e. representing the whole fracture. Other keys of the mesh record allow labeling regions given only by numbers, defining new regions in terms of element numbers (e.g to have leakage on single element), defining boundary regions, and set operations with region sets, see Section 3.2.1 for details.

Flow setting

Next, we setup the flow problem. We shall consider a flow driven only by the pressure gradient (no gravity), setting the Dirichlet boundary condition on the whole boundary with the pressure head equal to $x+y$. The conductivity will be 1 on the 2D domain and 10 on the 1D domain. The fracture width will be $\delta_1 = 1$ (quite unnatural) as well as the transition parameter $\sigma_2 = 1$ which describes a "conductivity" between dimensions. These are currently the default values.

```
    primary_equation = {
      TYPE = "Steady_MH",

      bulk_data = [
        { r_set = "1d_domain", conductivity = 10 },
        { region = "2d",       conductivity = 1  }
      ],

      bc_data = [
        { r_set = "BOUNDARY",
          bc_type = "dirichlet",
          bc_pressure = { TYPE="FieldFormula", value = "x+y" }
        }
      ],

```

```
28        output = {
29          output_stream = { REF = "/system/output_streams/0" },
30          pressure_p0 = "flow_output_stream",
31          pressure_p1 = "flow_output_stream",
32          velocity_p0 = "flow_output_stream"
33        },
34
35        solver = { TYPE = "Petsc", accuracy = 1e-07 }
36      }, // primary equation
```

On line 11, we specify particular implementation (numerical method) of the flow solver, in this case the Mixed-Hybrid solver for unsteady problems. On lines 16 – 19, we set mathematical fields that live on the computational domain (i.e. the bulk domain), we set only the conductivity field since other bulk fields have appropriate default values. On lines 21 – 26, we set fields for boundary conditions (`bc_data`). We use implicitely defined set "BOUNDARY" that contains all boundary regions and set there dirichlet boundary condition in terms of the pressure head. In this case, the field is not of the implicit type `FieldConstant`, so we must specify the type of the field `TYPE=FieldFormula`. See Section **??** for other field types. On lines 28 – 33, we specify which output fields should be written into which output stream (that means particular output file, with given format). Currently, we support only one output stream per equation, so this allows at least switching individual output fields on or off. Notice the reference used on line 29 pointing to the definition of the output streams at the end of the file. Finally, we specify type of the linear solver and its tolerance.

Transport setting

We also consider subsequent transport problem with the porosity $\theta = 0.25$ and zero initial concentration. The boundary condition is equal to 1 and is automatically applied only on the inflow part of the boundary. There are also some adsorption and dual porosity models in this particular test case, but we do not discuss this topic here for the sake of simplicity.

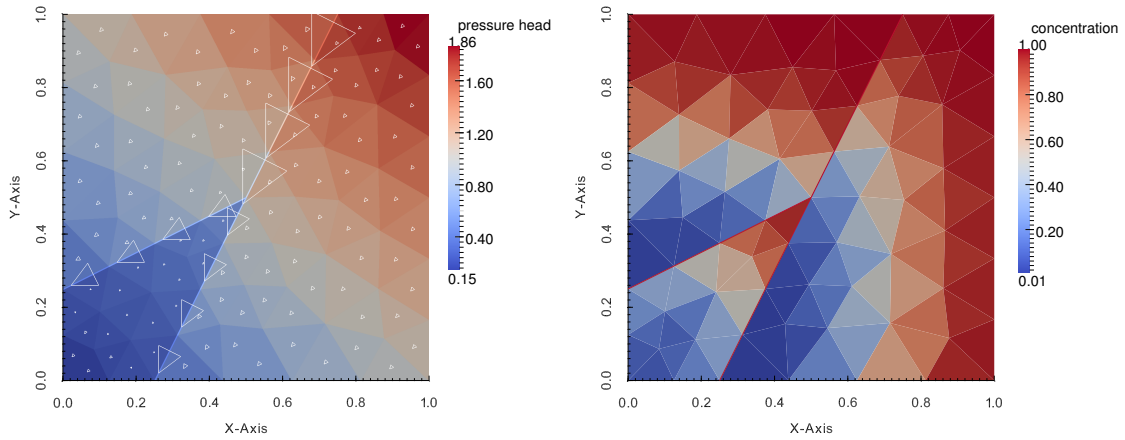
```
37      secondary_equation = {
38        TYPE = "TransportOperatorSplitting",
39
40        dual_porosity = true,
41        sorption_enable = true,
42        substances = [ "age", "U235" ],
43
44        bulk_data = [
45          { r_set = "ALL",
46            init_conc = 0,
47            por_m = 0.25,
48            por_imm = 0.25,
49            alpha = [0.01, 0.01],
50            phi = 0.5,
51            sorp_type = [1, 2],
```

```
52            sorp_coef0 = [0.02, 0.02],
53            sorp_coef1 = [0, 0.5]
54          }
55        ],
56
57        bc_data = [
58          { r_set = "BOUNDARY",
59            bc_conc = 1.0
60          }
61        ],
62
63        output = {
64          output_stream = { REF = "/system/output_streams/1" },
65          save_step = 0.01,
66          mobile_p0 = "transport_output_stream"
67        },
68
69        time = { end_time = 1.0 }
70      } // secondary_equation
71    }, // problem
```

For the transport problem we use implementation called "TransportOperatorSplitting" which is explicit finite volume solver of the convection equation (without diffusion), the operator splitting is used for the equilibrium adsorption as well as for the dual porosity model. Both of these are switched on as we can see on lines 40, 41. On the next line, we set names of transported substances, here it is the age of the water and the uranium 235. On lines 44 – 55, we set the bulk fields in particular the porosity 'por_m' and the initial concentrations (one for every substance). However, on line 46, we see only single value since an automatic conversion is applied to turn the scalar zero into the zero vector (of size 2). On line 53, we can see vector that set different adsorption coefficients for the two substances. Then, on lines 57 – 61, we set the boundary fields namely the concentration on the inflow part of the boundary. We need not to specify type of the condition since currently this is the only one available. In the output record we have to specify the save step (line 65) for the output fields. And finally, we have to set the time setting, here only the end time of the simulation since the step size is determined from the CFL condition, however you can set smaller time step if you want.

Output streams and results

```
72    system = {
73      output_streams = [
74        {
75          file = "test3.pvd",
76          format = { TYPE = "vtk", variant = "ascii" },
77          name = "flow_output_stream"
78        },
79        {
80          file = "test3-transport.pvd",
```



(a) Elementwise pressure head and velocity field (triangles).

(b) Propagation of U235 from the inflow part of the boundary.

Figure 1.1: Results of the tutorial problem.

```
81            format = { TYPE = "vtk", variant = "ascii" },
82            name = "transport_output_stream"
83          }
84        ]
85    }
86  }
```

The end of the input file contains declaration of two output streams, one for the flow problem and one for the transport problem. Currently, we support output into VTK format and GMSH data format. On Figure 1.1 you can see the results, the pressure and the velocity field on the left and the concentration of U235 at time $t = 0.9$ on the right. Even if the pressure gradient is the same on the 2D domain as on the fracture, the velocity field is ten times faster on the fracture. Since porosity is same, the substance is transported faster by the fracture and then appears in the bottom left 2D domain before the main wave propagating solely through the 2D domain.

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

Flow123d provides models for Dary flow in porous media as well as for the transport and reactions of soluted substances. In this section, we describe mathematical formulations of these models together with physical meaning and units of all involved quantities. Common and unique feature of all models is support of domains with mixed dimension. Let $\Omega_3 \subset \mathbf{R}^3$ be an open set representing continuum approximation of porous and fractured medium. Similarly, we consider open set $\Omega_2 \subset \mathbf{R}^2$ representing 2D fractures and open set $\Omega_1 \subset \mathbf{R}^3$ of 1D channels or preferential paths (see Fig 2.1). We assume that Ω_2 and Ω_1 are polygonal. For every dimension $d = 1, 2, 3$, we introduce a triangulation $\mathcal{T}_d$ of the open set Ω_d that consists of finite elements T_d^i, $i = 1, \ldots, N_E^d$. The elements are simplexes that is tetrahedrons, triangles and lines.

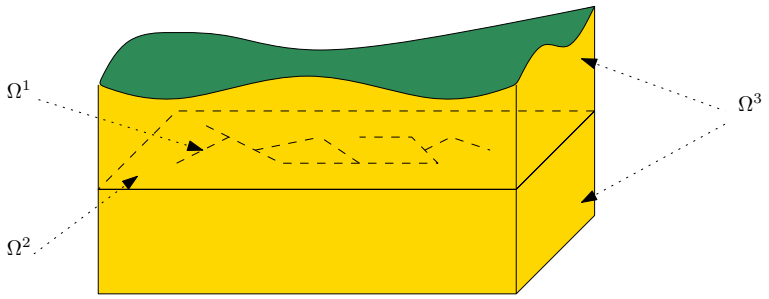


Figure 2.1: Scheme of a problem with domains of multiple dimensions.

Present numerical methods requires meshes satisfying the compatibility conditions

$$T_{d-1}^i \cap T_d \subset \mathcal{F}_d, \qquad \text{where } \mathcal{F}_d = \bigcup_k \partial T_d^k \tag{2.1}$$

and

$$T_{d-1}^i \cap \mathcal{F}_d \text{ is either } T_{d-1}^i \text{ or } \emptyset \tag{2.2}$$

for every $i \in \{1, \ldots, N_E^{d-1}\}$, $j \in \{1, \ldots, N_E^d\}$, and $d = 2, 3$. That is the $(d-1)$-dimensional elements are either between d-dimensional elements and match their sides or they poke out of Ω_d.

2.1 Darcy flow model

We consider simplest model for the velocity of the steady or unsteady flow in porous and fractured medium given by Darcy low:

$$\boldsymbol{w} = -\mathbb{K}\nabla H \quad \text{on } \Omega_d, \text{ for } d = 1, 2, 3. \tag{2.3}$$

We drop the dimension index of quantities in equations if it is same as the dimension of the set where the equation holds. In (2.3), $\boldsymbol{w}_d$ [ms^{-1}] is the superficial velocity, $\mathbb{K}_d$ is the conductivity tensor, and H_d [m] is the piezometric head. The velocity is related to the flux $\boldsymbol{q}_d$ with units [m^{4-d}s^{-1}] through

$$\boldsymbol{q}_d = \delta_d \boldsymbol{w}_d.$$

where δ_d [m^{3-d}] is a cross section coefficient, in particular $\delta_3 = 1$, δ_2 [m] is the thickness of a fracture, and δ_1 [m^2] is the cross-section of a channel. The flux q_d is the volume of the liquid (water) that pass through a unit square ($d = 3$), unit line ($d = 2$), or through a point ($d = 1$) per one second. The conductivity tensor is given by the product $\mathbb{K}_d = k_d \mathbb{A}_d$, where $k_d > 0$ is the hydraulic conductivity [ms^{-1}] and A_d is $3x3$ dimensionless anisotropy tensor which has to be symmetric and positive definite. The piezometric-head H_d has units [m] and is related to the pressure head h_d by $H_d = h_d + z$ assuming that the gravity force acts in negative direction of the z-axes. Combining these relations we get Darcy low in the form:

$$\boldsymbol{q} = -\delta k \mathbb{A} \nabla (h + z) \qquad \text{on } \Omega_d, \text{ for } d = 1, 2, 3. \tag{2.4}$$

Next, we employ continuity equation for saturated porous medium:

$$\partial_t (S\, h) + \operatorname{div} \boldsymbol{q} = F \qquad \text{on } \Omega_d, \text{ for } d = 1, 2, 3, \tag{2.5}$$

where S_d is the storativity and F_d is a source term. In our setting the principal unknowns of the system (2.4, 2.5) are the pressure head h_d and the flux $\boldsymbol{q}_d$.

The storativity $S_d > 0$ or the volumetric specific storage [m^{-1}] can be expressed as

$$S_d = \gamma_w (\beta_r + \nu \beta_w), \tag{2.6}$$

where γ_w [kgm^{-2}s^{-2}] is the specific weight of water, ν is the porosity [$-$], β_r is compressibility of the bulk material of the pores (rock) and β_w is compressibility of the water both with units [kg^{-1}ms^{-2}]. For steady problems we set $S_d = 0$ for all dimensions $d = 1, 2, 3$. The source term F_d [m^{3-d}s^{-1}] on the right hand side of (2.5) consists of the volume density of prescribed sources f_d [s^{-1}] and flux from higher dimension. Exact formula is slightly different for every dimension and will be discussed presently.

On Ω_3 we simply have $F_3 = f_3$ [s^{-1}].

On the set $\Omega_2 \cap \Omega_3$ the fracture is surrounded by one 3D surface from every side (or just one surface since we allow also 2D models on the boundary). On $\partial\Omega_3 \cap \Omega_2$ we prescribe boundary condition of Robin type

$$\begin{aligned} \boldsymbol{q}_3 \cdot \boldsymbol{n}^+ &= q_{32}^+ = \sigma_3^+ (h_3^+ - h_2), \\ \boldsymbol{q}_3 \cdot \boldsymbol{n}^- &= q_{32}^- = \sigma_3^- (h_3^- - h_2), \end{aligned}$$

where $\boldsymbol{q}_3 \cdot \boldsymbol{n}^{+/-}$ [ms^{-1}] is the outflow from Ω_3, $h_3^{+/-}$ is a trace of the pressure head on Ω_3, h_2 is the pressure head on Ω_2, and $\sigma_3^{+/-} = \sigma_{32}$ [s^{-1}] is the transition coefficient that will be discussed later. On the other hand, the sum of the interchange fluxes $\boldsymbol{q}_{32}^{+/-}$ forms a volume source on Ω_2. Therefore F_2 [ms^{-1}] on the right hand side of (2.5) is given by

$$F_2 = \delta_2 f_2 + (q_{32}^+ + q_{32}^-). \tag{2.7}$$

The communication between Ω_2 and Ω_1 is similar. However, in the 3D ambient space, an 1D channel can join multiple 2D fractures $1, \ldots, n$. Therefore, we have n independent outflows from Ω_2:

$$\boldsymbol{q}_2 \cdot \boldsymbol{n}^i = q_{21}^i = \sigma_2^i (h_2^i - h_1),$$

where $\sigma_2^i = \delta_2^i \sigma_{21}$ [ms^{-1}] is the transition coefficient integrated over the width of the fracture i. Sum of the fluxes forms part of F_1 [m^{2}s^{-1}]

$$F_1 = \delta_1 f_1 + \sum_i q_{21}^i. \tag{2.8}$$

The transition coefficients σ_d [m^{3-d}s^{-1}] are independent parameters in our setting however in practice they should be related to the conductivity in direction (or plane) perpendicular to the fracture (channel). According to [4] one can use

$$\sigma_3 = \frac{2\mathbb{K}_2 : \boldsymbol{n}_2 \otimes \boldsymbol{n}_2}{\delta_2}, \sigma_2^i = \frac{2\delta_2 \mathbb{K}_1 : \boldsymbol{n}_1^i \otimes \boldsymbol{n}_1^i}{\delta_1}$$

where $\boldsymbol{n}_2$ is normal to the fracture (sign doesn't matter) and $\boldsymbol{n}_1^i$ is normal to the channel that is tangential to the fracture i.

In order to obtain unique solution we have to prescribe boundary conditions. Currently we support three basic types of boundary condition. Consider disjoint decomposition of the boundary

$$\partial\Omega_d = \Gamma_d^D \cap \Gamma_d^N \cap \Gamma_d^R$$

into Dirichlet, Neumann, and Robin parts. We prescribe

$$h_d = h_d^D \qquad \text{on } \Gamma_d^D, \tag{2.9}$$

$$\boldsymbol{q}_d \cdot \boldsymbol{n} = q_d^N \qquad \text{on } \Gamma_d^N, \tag{2.10}$$

$$\boldsymbol{q}_d \cdot \boldsymbol{n} = \sigma_d^R (h_d - h_d^R) \qquad \text{on } \Gamma_d^R. \tag{2.11}$$

where h_d^D, h_d^R is the prescribed pressure head [m], which alternatively can be prescribed through the piezometric head H_d^D, H_d^R respectively. q_d^N is the prescribed surface density of the boundary outflow [m^{4-d}s^{-1}], and σ_d^R is the transition coefficient [m^{3-d}s^{-1}]. The problem is well posed only if there is Dirichlet or Robin boundary condition on every component of the set $\Omega_1 \cup \Omega_2 \cup \Omega_3$ and $\sigma_d > 0$ for $d = 2, 3$.

For unsteady problems one has to specify initial condition in terms of initial pressure head h_d^0 or initial piezometric head H_d^0.

2.2 Transport of substances

Flow123d can simulate transport of substances dissolved in water. The transport mechanism is governed by the *advection*, and the *hydrodynamic dispersion*. Moreover the substances can move between ground and fractures.

In the domain Ω_d of dimension $d \in \{1, 2, 3\}$, we consider a system of mass balance equations in the following form:

$$\delta_d \partial_t(\vartheta c^i) + \operatorname{div}(\boldsymbol{q}_d c^i) - \operatorname{div}(\vartheta \delta_d \mathbb{D}^i \nabla c^i) = F_S + F_C(c^i) + F_R(c^1, \dots, c^s). \tag{2.12}$$

The principal unknown is the concentration c^i [kgm^{-3}] of a substance $i \in \{1, \dots, s\}$, which means weight of the substance in unit volume of the water. Other quantities are:

- ϑ [] is the porosity, i.e. fraction of space occupied by water and the total volume.
- The hydrodynamic dispersivity tensor $\mathbb{D}^i$ [m^{2}s^{-1}] has the form

$$\mathbb{D}^i = D_m^i \tau \mathbb{I} + |\boldsymbol{v}| \left(\alpha_T^i \mathbb{I} + (\alpha_L^i - \alpha_T^i) \frac{\boldsymbol{v} \times \boldsymbol{v}}{|\boldsymbol{v}|^2} \right),$$

  which represents (isotropic) molecular diffusion, and mechanical dispersion in longitudal and transversal direction to the flow. Here D_m^i [m^{2}s^{-1}] is the molecular diffusion coefficient of the i-th substance (usual magnitude in clear water is 10^{-9}), $\tau = \vartheta^{1/3}$ is the tortuosity (by [5]), α_L^i [m] and α_T^i [m] is the longitudal dispersivity and the transversal dispersivity, respectively. Finally, $\boldsymbol{v}$ [ms^{-1}] is the *microscopic* water velocity, related to the Darcy flux $\boldsymbol{q}_d$ by the relation $\boldsymbol{q}_d = \vartheta \delta_d \boldsymbol{v}$. The value of D_m^i for specific substances can be found in literature (see e.g. [1]). For instructions on how to determine α_L^i, α_T^i we refer to [2, 3].
- F_S [kgm^{-d}s^{-1}] is the density of concentration sources.
- $F_C(c^i)$ [kgm^{-d}s^{-1}] is the density of concentration sources due to exchange between regions with different dimensions, see (2.15) below.
- The reaction term $F_R(\dots)$ [kgm^{-d}s^{-1}] is currently neglected.

Initial and boundary conditions. At time $t = 0$ the concentration is determined by the initial condition

$$c^i(0, \boldsymbol{x}) = c_0^i(\boldsymbol{x}).$$

The physical boundary $\partial\Omega_d$ is decomposed into two parts:

$$\Gamma_D(t) = \{\boldsymbol{x} \in \partial\Omega_d \,|\, \boldsymbol{q}(t, \boldsymbol{x}) \cdot \boldsymbol{n}(\boldsymbol{x}) < 0\},$$
$$\Gamma_N(t) = \{\boldsymbol{x} \in \partial\Omega_d \,|\, \boldsymbol{q}(t, \boldsymbol{x}) \cdot \boldsymbol{n}(\boldsymbol{x}) \geq 0\},$$

where $\boldsymbol{n}$ stands for the unit outward normal vector to $\partial\Omega_d$. On the inflow part Γ_D, the user must provide Dirichlet boundary condition for concentrations:

$$c^i(t, \boldsymbol{x}) = c_D^i(t, \boldsymbol{x}) \text{ on } \Gamma_D(t),$$

while on Γ_N we impose homogeneous Neumann boundary condition:

$$-\vartheta \delta_d \mathbb{D}^i(t, \boldsymbol{x}) \nabla c^i(t, \boldsymbol{x}) \cdot \boldsymbol{n}(\boldsymbol{x}) = 0 \text{ on } \Gamma_N(t).$$

Communication between dimensions. Transport of substances is considered also on interfaces of physical domains with adjacent dimensions (i.e. 3D-2D and 2D-1D, but not 3D-1D). Denoting c_{d+1}, c_d the concentration of a given substance in Ω_{d+1} and Ω_d, respectively, the comunication on the interface between Ω_{d+1} and Ω_d is described by:

$$q^c = \delta_{d+1}\sigma^c(\vartheta_{d+1}c_{d+1} - \vartheta_d c_d) + \begin{cases} q^w c_{d+1} & \text{if } q^w \geq 0, \\ q^w c_d & \text{if } q^w < 0, \end{cases} \tag{2.13}$$

where

- q^c [kgm^{-d}s^{-1}] is the density of concentration flux from Ω_{d+1} to Ω_d,
- σ^c [ms^{-1}] is a transition parameter. Its nonzero value causes mass exchange between dimensions whenever the concentrations differ. It is recommended to set either $\sigma^c = 0$ (exchange due to water flux only) or, similarly as in (2.1),

$$\sigma^c \approx \frac{\delta_{d+1}}{\delta_d}\mathbb{D} : \boldsymbol{n} \otimes \boldsymbol{n}.$$

- q^w [m^{3-d}s^{-1}] is the water flux from Ω_{d+1} to Ω_d, i.e. $q^w = \boldsymbol{q}_{d+1} \cdot \boldsymbol{n}_{d+1}$.

Equation (2.13) is incorporated as the total flux boundary condition for the problem on Ω_{d+1} and a source term in Ω_d:

$$-\vartheta\delta_{d+1}\mathbb{D}\nabla c_{d+1} \cdot \boldsymbol{n} + q^w c_{d+1} = q^c, \tag{2.14}$$

$$F_C^d = q^c. \tag{2.15}$$

Chapter 3

File formats

3.1 Main input file (CON file format)

In this section, we shall describe structure of the main input file that is given through the parameter `-s` on the command line. The file formats of other files that are referenced from the main input file and used for input of the mesh or large field data (e.g. the GMSH file format) are described in following sections. The input subsystem was designed with the aim to provide uniform initialization of C++ classes and data structures. Its structure is depicted on Figure 3.1. The structure of the input is described by the Input Types Tree (ITT) of (usually static) objects which follows the structure of the classes. The data from an input file are read by apropriate reader, their structure is checked against ITT and they are pushed into the Internal Storage Buffer (ISB). An accessor object to the root data record is the result of the file reading. The data can be retrieved through accessors which combine raw data stored in in IBS with their meaning described in ITT. ITT can be printed out in various formats providing description of the input structure both for humans and other software.

Currently, the JSON input file format is only implemented and in fact it is slight extension of the JSON file format. On the other hand the data for initialization of the C++ data structures are coded in particular way. Combination of this extension and restriction of the JSON file format produce what we call CON (C++ object notation) file format.

3.1.1 JSON for humans

Basic syntax of the CON file is very close to the JSON file format with only few extensions, namely:

- You can use C++ (or JavaScript) comments. One line comments `//` and multi-line comments `/* */`.
- The quoting of the keys is optional if they do not contain spaces (holds for all CON keys).
- You can use equality sign `=` instead of colon `:` for separation of keys and values in JSON objects.

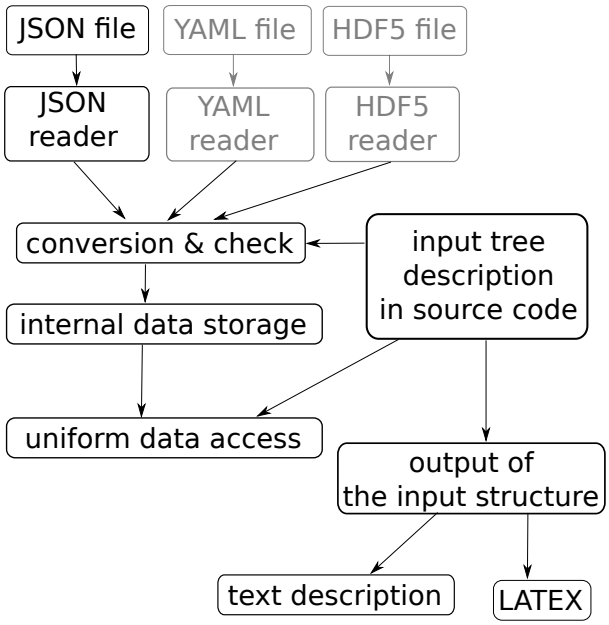


Figure 3.1: Sturucture of the input subsystem. Grey boxes are not implemented yet.

- 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. For the description of the JSON format we refer to `http://www.json.org/`.

3.1.2 CON constructs

The CON file format constructs are designed for initialization of C++ strongly typed variables. The primitive data types can be initialized from the primitive CON constructs:

- *Bool* — initialized from the JSON keywords `true` and `false`.
- *Double*, *Integer* — initialized from JSON numeric data.
- *String*, *FileName*, *Selections* — initialized from JSON strings

Selections are typed like the C++ enum types that are initialized from them. Various kind of containers can be initialized by the *Array* construct, that is an JSON array with elements of the same CON type. The C++ structures and classes can be initialize from the *Record* construct, which is represented by a JSON object. However, in constrast to JSON, these Records have different types in similar way as the strong typed C++ structures. The types are described by ITT of the particular program which can be printed out in several formats, in particular description of ITT for Flow123d forms content of Chapter 4. In order to allow certain kind of polymorphism, we introduce also the *AbstractRecord* construct, where the type of the record is not given by ITT but can be chosen as part of the input.

3.1.3 CON special keys

All keys in Records should be in lower case, possibly using digits and underscore. The keys all in upper case are reserved for special function in the CON file. These are:

TYPE key :

```
TYPE=<Selection of AbstractRecord>
```

Is used to specify particular type of an AbstractRecord. This way you can choose which particular implementation of an abstract C++ class should be instantiated. The value of the key is a string from the Selection that consists of names of Records that was declared as descendants of the AbstractRecord.

REF key :

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

The record in input file that contains only the key `REF` is replaced by the JSON entity that is referenced by the `<address>`. The address is a string with format similar to UNIX path, i.e. with grammar

```
<address> = <address> / <item>
          = <item>
          = <null>
<item> = <index>
       = <key>
       = ..
```

where `index` is non-negative integer and key is valid CON record key (lowercase, digits, underscores). 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 two dots `".."` to move to parent entity.

Example:

```
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"
}
```

3.1.4 Record types

A Record type is given by the set of key specifications, which in turn consist from: key name, type of value and default value specification. Default value specification can be:

obligatory — means no default value, which has to be specified at input.

optional — means no default value, but value is needs not to be specified. Unspecified value usually means that you turn off some functionality.

default at declaration — the default value is explicitly given in declaration and is automatically provided by the input subsystem if needed

default at read time — the default value is provided at read time, usually from some other variable. In the documentation, there is only textual description where the default value comes from.

Implicit creation of composed entities

Consider a Record type in which all keys have default values (possibly except one). Then the specification of the Record can contain a *key for default construction*. User can specify only the value of this particular key instead of the whole record, all other keys are initialized from its default values. Moreover, an AbstractRecord type may have a default value for the `TYPE` key. This allows to express simple tasks by simple inputs but still make complex inputs 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.

3.2 Important Record types of Flow123d input

3.2.1 Mesh record

The mesh record provides initialization for the computational mesh consisting of points, lines, triangles and tetrahedrons in 3D space. Currently, we support only GMSH mesh file format MSH ASCII. The input file is provided by the key `mesh_file`. The file format allows to group elements into *regions* identified either by ID number or by string label. The regions with labels starting with the dot character are treated as *boundary regions*. Their elements are removed from the computational domain, however they can be used to specify boundary conditions. Other regions are called *bulk regions*. User can create new labeled regions through the key `regions`, the new region can be specified either by its ID or by list of IDs of its elements. The latter possibility overrides original region assigned to the elements, which can be useful for specification of small areas "ad hoc". The key `sets` allows specification of sets of regions in terms of list of region IDs or labels and basic set operations. The difference between regions and sets is that regions form disjoint covering of elements, the sets, however, may overlap. There are three predefined region sets: "ALL", "BOUNDARY", "BULK".

3.2.2 Field records

A general time and space dependent, scalar, vector, or tensor valued function can be specified through the family of abstract records Field $R^m -> \mathcal{S}$, where m is currently always $m = 3$ and $\mathcal{S}$ is a specification of the target space, which can be:

- $\mathcal{T}$ — scalar valued field, with scalars of type $\mathcal{T}$
- $\mathcal{T}[d]$ — vector valued field, with vector of fixed size d and elements of type $\mathcal{T}$
- $\mathcal{T}[\mathtt{n}]$ — vector valued field, with vector of variable size (given by some input) and elements of type $\mathcal{T}$
- $\mathcal{T}[d, d]$ — tensor valued field, with square tensor of fixed size and elements of type $\mathcal{T}$

the scalar types can be

- **Real** — scalar real valued field
- **Int** — scalar integer valued field
- **Enum** — scalar non negative integer valued field, should be convertible to appropriate C++ enum type

Each of these abstract record has the same set of descendants which implement various algorithms to specify and compute values of the field. These are

FieldConstant — field that is constant in space

FieldFormula — field that is given by runtime parsed formula using x, y, z, t coordinates. The Function Parser library is used with syntax rules described here.

FieldPython — field can be implemented by Python script either specified by string (key `script_string`) or in external file (key `script_file`.

FieldElementwise — discrete field, currently only piecewise constant field on elements is supported, the field can given by the MSH ASCII file specified in key `gmsh_file` and field name in the file given by key `field_name`. The file must contain same mesh as is used for computation.

FieldInterpolated — allows interpolation between different meshes. Not yet fully supported.

Several automatic conversions are implemented. Scalar values can be used to set constant vectors or tensors. Vector value of size d can be used to set diagonal tensor $d \times d$. Vector value of size $d(d-1)/2$, e.g. 6 for $d = 3$, can be used to set symmetric tensor. These rules apply only for FieldConstant and FieldFormula. Moreover, all Field abstract types have default value `TYPE=FieldConstant`. Thus you can just use the constant value instead of the whole record.

Examples:

```
constant_scalar_function = 1.0
// is same as
constant_scalar_function = {
  TYPE=FieldConstant,
  value=1.0
}

conductivity_tensor = [1 ,2, 3]
// is same as
conductivity_tensor = {
    TYPE=FieldConstant,
    value=[[1,0,0],[0,2,0],[0,0,3]]
}

concentration = {
    TYPE=FieldFormula,
    value="x+y+z"
}
//is same as (provided the vector has 2 elements)
concentration = {
    TYPE=FieldFormula,
    value=["x+y+z", "x+y+z"]
}
```

3.2.3 Field data for equations

Every equation record has keys `bulk\_data` and `bc_data`. Both have the same structure, however, the first one is intended to set the bulk fields (on bulk regions) while the second serves for initialization of the boundary fields (on boundary regions). These keys contains an array of region-time initialization records like the BulkData record of the DarcyFlow equation. Every such record specify fields on particular region (keys `region` and `rid`) or on a region set (key `r_set`) starting from the time specified by the key `time`. The array is processed sequentially and latter values overwrites the previous ones. Times should form a non-decreasing sequence.

Example:

```
bulk_data = [
    { // time=0.0  - default value
        r_set="BULK"
        conductivity=1   // setting the conductivity field on all regions
    }
    {
        region="2d_part"
        conductivity=2  // overwriting the previous value
    }
    {   time=1.0
        region="2d_part"
        conductivity={
```

```
            // from time=1.0 we switch to the linear function in time
            TYPE=FieldFormula
            value="2+t"
        }
    }
    {   time=2.0
        region="2d_part"
        conductivity={
            // from time=2.0 we switch to elementwise field, but only
            // on the region "2d_part"
            TYPE=FieldElementwise
            gmsh_file="./input/data.msh"
            field_name="conductivity"
        }
    }
]
```

3.3 Mesh and data file format MSH ASCII

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

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

The scheme of the file is as follows:

```
$MeshFormat
<format version>
$EndMeshFormat

$PhysicalNames
<number of items>
<dimension>     <region ID>     <region label>
...
$EndPhysicalNames

$Nodes
<number of nodes>
<node ID> <X coord> <Y coord> <Z coord>
...
$EndNodes

$Elements
<number of elements>
<element ID> <element shape> <n of tags> <tags> <nodes>
...
$EndElements

$ElementData
```

```
<n of string tags>
    <field name>
    <interpolation scheme>
<n of double tags>
    <time>
<n of integer tags>
    <time step index>
    <n of components>
    <n of items>
    <partition index>
<element ID> <component 1> <component 2> ...
...
$EndElementData
```

Detailed description of individual sections:

`PhysicalNames` — assign labels to region IDs

`Nodes` — `<number of nodes>` is also number of data lines that follows. Node IDs are unique but need not to form an aritmetic sequance. Coordinates are float numbers.

`Elements` — Element IDs are unique but need not to form an aritmetic sequence. `<element shape>` is integer code of the shape, we support only points (15), lines (1), triangles (2), and tetrahedrons (4). Default number of tags is 3. The first is the region ID, the second is ID of the geometrical entity (that was used in original geometry file from which the mesh was generated), and the third tag is the partition number. `nodes` is list of node IDs with size according to the element shape.

`ElementData` — the header has 2 string tags, 1 double tag, and 4 integer tags with default meaning. For the purpose of the `FieldElementwise` the tags `<field name>`, `<n of components>`, and `<n of items>` are obligatory.

3.4 Output files

Flow123d support output of scalar and vector data fields into two formats. The first is the native format of the GMSH software (usually with extension `msh`) which contains computational mesh followed by data fields for sequence of time levels. The second is the XML version of VTK files. These files can be viewed and post-processed by several visualization softwares. However, our primal goal is to support data transfer into the Paraview visualization software. See key `format`.

3.4.1 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 `piezo_head_p0`).

3.4.2 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 $[M/L^3]$.

3.4.3 Auxiliary output files

Profiling information

On every run we collect some basic profiling 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 4

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: OutputStream*>

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: **OutputStream**

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 connected output file.

`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` : Homogeneous Neoumann BC.

`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.

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

Default: <*optional*>

Anisotropy of the 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*>

Isotropic conductivity scalar.

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

Default: <*optional*>

Transition coefficient between dimensions.

`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.

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

Default: <*optional*>

Anisotropy of the 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*>

Isotropic conductivity scalar.

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

Default: <*optional*>

Transition coefficient between dimensions.

`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.

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

Default: <*optional*>

Anisotropy of the 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*>

Isotropic conductivity scalar.

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

Default: <*optional*>

Transition coefficient between dimensions.

`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: OutputStream*>

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

Sorptions

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: **Sorptions** implements abstract type: Reactions

Information about all the limited solubility affected sorptions.

`TYPE` = *<selection: Reactions_TYPE_selection>*

Default: Sorptions

Sub-record selection.

`solvent_dens` = *<Double >*

Default: 1.0

Density of the solvent.

`substeps` = *<Integer >*

Default: 10

Number of equidistant substeps, molar mass and isotherm intersections

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

Default: *<obligatory>*

Names of all the sorbing species

`molar_masses` = *<Array of Double >*

Default: *<obligatory>*

Specifies molar masses of all the sorbing species

`solubility` = *<Array of Double >*

Default: *<obligatory>*

Specifies solubility limits of all the sorbing species

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

Default: *<obligatory>*

Containes region specific data necessery to construct isotherms.

---

record: **Sorption_BulkData**

Record to set BULK fields of the equation 'Sorption'. 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 Sorption_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.

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

Default: <*optional*>

Rock matrix density.

`sorption_types` = <*abstract type: Field:R3 → Enum[n]*>

Default: <*optional*>

Considered adsorption is described by selected isotherm.

`mult_coefs` = <*abstract type: Field:R3 → Real[n]*>

Default: <*optional*>

Multiplication parameters (k, omega) in either Langmuir c_s = omega * (alpha*c_a)/(1- alpha*c_a) or in linear c_s = k * c_a isothermal description.

`second_params` = <*abstract type: Field:R3 → Real[n]*>

Default: <*optional*>

Second parameters (alpha, ...) defining isotherm c_s = omega * (alpha*c_a)/(1- alpha*c_a).

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

Default: <*optional*>

Mobile porosity of the rock matrix.

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

Default: <*optional*>

Immobile porosity of the rock matrix.

---

abstract type: **Field:R3 → Enum[n]** default descendant: FieldConstant

---

Descendants:

Abstract record for all time-space functions.

`FieldConstant`

`FieldFormula`

`FieldPython`

`FieldElementwise`

---

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

---

R3 → Enum[n] Field constant in space.

`TYPE` = *<selection: Field:R3 → Enum[n]_TYPE_selection>*

Default: FieldConstant

Sub-record selection.

`value` = *<Array [1,] of selection: SorptionType>*

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: **SorptionType**

Possible values:

`none` : No sorption considered

`linear` : Linear isotherm described sorption considered.

`langmuir` : Langmuir isotherm described sorption considered

`freundlich` : Freundlich isotherm described sorption considered

---

record: **FieldFormula** implements abstract type: Field:R3 → Enum[n]

R3 → Enum[n] Field given by runtime interpreted formula.

`TYPE` = *<selection: Field:R3 → Enum[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: **FieldPython** implements abstract type: Field:R3 → Enum[n]

R3 → Enum[n] Field given by a Python script.

`TYPE` = *<selection: Field:R3 → Enum[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: **FieldElementwise** implements abstract type: Field:R3 → Enum[n]

R3 → Enum[n] Field constant in space.

`TYPE` = *<selection: Field:R3 → Enum[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.

---

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: **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).

record: **TransportOperatorSplitting_BulkData**

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.

`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

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

Default: <*optional*>

Longitudal dispersivity.

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

Default: <*optional*>

Transversal dispersivity.

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

Default: <*optional*>

Molecular diffusivity.

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

Default: <*optional*>

Coefficient of diffusive transfer through fractures.

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

Default: <*optional*>

Penalty parameter influencing the discontinuity of the solution.

Bibliography

[1] Milena Číslerová and Tomáš Vogel. *Transportní procesy*. ČVUT, 1998.

[2] Ghislain De Marsily. *Quantitative hydrogeology: Groundwater hydrology for engineers*. Academic Press, New York, 1986.

[3] Patrick A Domenico and Franklin W Schwartz. *Physical and chemical hydrogeology*, volume 824. Wiley New York, 1990.

[4] Vincent Martin, Jeroˆme Jaffre, and Jean E. Roberts. Modeling fractures and barriers as interfaces for flow in porous media. *SIAM Journal on Scientific Computing*, 26(5):1667, 2005.

[5] RJ Millington and JP Quirk. Permeability of porous solids. *Transactions of the Faraday Society*, 57:1200–1207, 1961.