Technical university of Liberec

Faculty of mechatronics, informatics
and interdisciplinary studies

Flow123d

version 1.8.3

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

Liberec, 2015

Authors:

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

Acknowledgement

This work was supported by the Ministry of Industry and Trade of the Czech Republic under the project no. FR-TI3/579.

Contents

Chapter 1

Getting Started

1.1 Introduction

Flow123D is a software for simulation of water flow, reactionary solute transport and heat transfer 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 a collection of tetrahedra, triangles and line segments.

The water flow model assumes a saturated medium described by the Darcy law. For discretization, we use lumped mixed-hybrid finite element method. We support both steady and unsteady water flow. The water flow model can be sequentially coupled with two different models for a solute transport or with a heat transfer model.

The first solute transport model can deal only with pure advection of several substances without any diffusion-dispersion term. It uses explicit Euler method for time discretization and finite volume method for space discretization and operator splitting method to couple with various processes described by the reaction term. The reaction term can treat any meaningful combination of the dual porosity, sorptions, decays and linear reactions. Alternatively, one can use interface to the experimental SEMCHEM package for more complex geochemistry.

The second solute transport model describes general advection with hydrodynamic dispersion for several substances. It uses implicit Euler method for time discretization and discontinuous Galerkin method of the first, second or third order for the discretization in space. Currently there is no support for reaction term, the operator splitting approach (although it is not suited for implicit time schemes) is planned for the next version.

The heat transfer model assumes equilibrium between temperature of the rock and the fluid phase. It uses the same numerical scheme as the second transport model.

The program support output of all input and many output fields into 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 the VTK data.

The program is implemented in C/C++ using essentially PETSC library for linear algebra. All models can run in parallel using MPI environment, however, the scalability

of the whole program is limited due to serial mesh and serial outputs.

The program is distributed under GNU GPL v. 3 license and is available on the project web page: http://flow123d.github.io

with sources on the GitHub: https://github.com/flow123d/flow123d.

1.2 Reading Documentation

The Flow123d documentation has two main parts. The first three chapters form a user manual which starts with getting and running the program and tutorial problem in chapter 1. The second chapter 2 provides detailed description of mathematical models of physical reality. The third chapter 4 documents all file types used by Flow123d, including mesh files, input and output files.

The second main part, consisting only of the chapter 5, is automatically generated. It mirrors directly the code and contains the whole input tree of the main input file. Description of input records, their structure and default values are supplied there and bidirectional links to the user manual are provided.

1.3 Running Flow123d

On the Linux system the program can be started either directly or through a script `flow123d.sh`, both placed in the `bin` directory of the installation package or of the source tree. 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 placeholder `${INPUT}` used in the path of an input file will be replaced by the `<directory>`. Default value is `input`.

`-o, --output_dir <directory>`
: All paths for output files will be relative to this `<directory>`. Default value is `output`.

`-l, --log <file_name>`
: Set base name of log files. Default value is `flow123d`. The log files are individual for every MPI process, placed in the output directory. The MPI rank of the process and the `log` suffix are appended to the base name.

`--no_log`
: Turn off logging.

`--no_profiler`
: Turn off profiler output.

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

`--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 solvers. 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. The syntax is as follows:

```
flow123d.sh [OPTIONS] -- [FLOW_PARAMS]
```

where everything after double dash is passed as parameters to the flow123d binary. The script accepts following options:

`-h, --help`
: Usage overview.

`--host <hostname>`
: Default value is the host name obtained by system `hostname` command, this argument can be used to override it. Resulting value is used to select a backend script `config/<hostname>.sh`, which describes particular method how to start parallel jobs, usually through some sort of PBS job queue system. If the script is not found, we try to start parallel processes directly on the actual host."

`-t, --walltime <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 MPI parallel processes for calculation.

`-m, --mem <memory limit>`
: Limits total available memory to `<memory limit>` bytes per process.

`-n, --nice <niceness>`
: Change priority of the calculation, higher values means lower priority. See the `nice` command.

`-ppn <processes per node>`
: Set number of processes started on one node for multicore systems. Number of processes set by `-np` parameter should be divisible by `<processes per node>`.

`-q, --queue <queue>`
: Select particular job queue on PBS systems. If running without PBS, it redirects stdout and stderr to the file `<queue>.<date>`, which appended date and time of the start of the job.

On the windows operating systems, we use Cygwin libraries in order to emulate Linux API. Therefore you have to keep the Cygwin libraries within the same directory 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 based 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 operating systems.

1.4 Tutorial Problem

In the following section, we shall provide an example cook book for preparing and running a model. It is one of the test problem with the main input file:

`tests/03_transport_small_12d/flow_vtk.con`

We shall start with preparation of the geometry using an external software and then we shall go thoroughly through the commented main input file. The problem includes steady Darcy flow, transport of two substances with explicit time discretization and a reaction term consisting of dual porosity and sorption model.

1.4.1 Geometry

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.

1.4.2 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 4.1.

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

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

The file starts with a selection of problem type (`SequentialCoupling`), 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 4.2.1 for details.

1.4.3 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 $k_2 = 10^{-7}$ ms^{-1} on the 2D domain and $k_1 = 10^{-6}$ ms^{-1} on the 1D domain. Both 2D domain and 1D domain `cross_section` will be set by default, meaning that the thickness of 2D domain is $\delta_2 = 1$ m and the fracture cross section is $\delta_1 = 1$ m^2. The transition coefficient σ_2 between dimensions can be scaled by setting the dimensionless parameter σ_{21} (`sigma`). This can be used for simulating additional effects which prevent the liquid transition from/to a fracture, like a thin resistance layer. Read Section 2.3 for more details.

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

      input_fields = [
        { r_set = "1d_domain", conductivity = 1e-6,
                               cross_section = 0.04,
                               sigma = 0.9 },
        { region = "2d",       conductivity = 1e-7  },
        { r_set = "BOUNDARY",
          bc_type = "dirichlet",
          bc_pressure = { TYPE="FieldFormula", value = "x+y" }
        }
      ],

```

```
28      output = {
29         output_stream = {
30           file = "flow.pvd",
31           format = { TYPE = "vtk", variant = "ascii" }
32         },
33         output_fields = [ "pressure_p0", "pressure_p1", "velocity_p0" ]
34       },
35
36       solver = {
37         TYPE = "Petsc",
38         a_tol = 1e-12,
39         r_tol = 1e-12
40       }
41     }, // primary equation
```

On line 15, we specify particular implementation (numerical method) of the flow solver, in this case the Mixed-Hybrid solver for steady problems. On lines 17 – 24, we set both mathematical fields that live on the computational domain and those defining the boundary conditions. We set only the conductivity field since other `input_fields` have appropriate default values. We use implicitly 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 4.2.2 for other field types. On lines 26 – 32, we specify which output fields should be written to the 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. See Section 4.4 for the list of available `output_fields`. Finally, we specify type of the linear solver and its tolerances.

1.4.4 Transport Setting

The flow model is followed by a transport model in the `secondary_equation` beginning on line 40. For the transport problem, we use an implementation called TransportOperatorSplitting which stands for an explicit finite volume solver of the convection equation (without diffusion). The operator splitting method is used for equilibrium sorption as well as for dual porosity model and first order reactions simulation.

```
42     secondary_equation = {
43       TYPE = "TransportOperatorSplitting",
44
45       substances = [
46         {name = "age", molar_mass = 0.018},     // water age
47         {name = "U235", molar_mass = 0.235}     // uranium 235
48       ],
49
50       input_fields= [
51         { r_set = "ALL",
52           init_conc = 0,
```

```
53            porosity= 0.25,
54            sources_density = [1.0, 0]
55          },
56          { r_set = "BOUNDARY",
57            bc_conc = [0.0, 1.0]
58          }
59        ],
60
61        time = { end_time = 1e6 },
62        mass_balance = { cumulative = true },
```

On lines 43 – 46, we set the transported `substances`, which are identified by their names. Here, the first one is the `age` of the water, with the molar mass of water, and the second one `U235` is the uranium isotope 235. On lines 48 – 57, we set the input fields, in particular zero initial concentration for all substances, `porosity` $\theta = 0.25$ and sources of concentration by `sources_density`. Notice line 50 where we can see only single value since an automatic conversion is applied to turn the scalar zero into the zero vector (of size 2 according to the number of substances).

The boundary fields are set on lines 54 – 56. We need not to specify the type of the condition since there is only one type in the current transport model. The boundary condition is equal to 1 for the uranium 235 and 0 for the age of the water and is automatically applied only on the inflow part of the boundary.

We also have to prescribe the `time` setting, here only the end time of the simulation (in seconds: 10^6 s ≈ 11.57 days) is required since the step size is determined from the CFL condition. However, a smaller time step can be enforced if necessary.

Reaction term of the transport model is described in the next subsection, including dual porosity and sorption.

1.4.5 Reaction Term

The input information for dual porosity, equilibrial sorption and possibly first order reations are enclosed in the record `reaction_term`, lines 61 – 100. Go to section 2.5 to see how the models can be chained.

The type of the first process is determined by `TYPE="DualPorosity"`, on line 62. The `input_fields` of dual porosity model are set on lines 64 – 71 and the output is disabled by setting an empty array on line 73.

```
63        reaction_term = {
64          TYPE = "DualPorosity",
65
66          input_fields= [
67            {
68              r_set="ALL",
69              diffusion_rate_immobile = [0.01,0.01],
70              porosity_immobile = 0.25,
71              init_conc_immobile = [0.0, 0.0]
```

```
72            }
73          ],
74
75          output_fields = [],
76
77          reaction_mobile = {
78            TYPE = "SorptionMobile",
79            solvent_density = 1000.0,     // water
80            substances = ["age", "U235"],
81            solubility = [1.0, 1.0],
82
83            input_fields= [
84              {
85                r_set="ALL",
86                rock_density = 2800.0,    // granit
87                sorption_type =  ["none", "freundlich"],
88                isotherm_mult = [0, 0.68],
89                isotherm_other = [0, 1.0]
90              }
91            ],
92            output_fields = []
93          },
94          reaction_immobile = {
95            TYPE = "SorptionImmobile",
96            solvent_density = 1000.0,     // water
97            substances = ["age", "U235"],
98            solubility = [1.0, 1.0],
99            input_fields = { REF="../../reaction_mobile/input_fields" },
100           output_fields = []
101         }
102       },
103
104       output_stream = {
105         file = "transport.pvd",
106         format = { TYPE = "vtk", variant = "ascii" },
107         time_step = 1e5
108       }
109
110     } // secondary_equation
111   } // problem
112 }
```

Next, we define the equilibrial sorption model such that `SorptionMobile` type takes place in the mobile zone of the dual porosity model while `SorptionImmobile` type takes place in its immobile zone, see lines 76 and 93. Isothermally described sorption simulation can be used in the case of low concentrated solutions without competition between multiple dissolved species.

On lines 77 – 89, we set the sorption related input information. The solvent is water so

the solvent_density is supposed to be constant all over the simulated area. The vector substances contains the list of names of soluted substances which are considered to be affected by the sorption. Solubility is a material characteristic of a sorbing substance related to the solvent. Elements of the vector solubility define the upper bound of aqueous concentration which can appear. This constrain is necessary because some substances might have limited solubility and if the solubility exceeds its limit they start to precipitate. `solubility` is a crucial parameter for solving a set of nonlinear equations, described further.

The record input_fields covers the region specific parameters. All implemented types of sorption can take the rock density in the region into account. The value of rock_density is a constant in our case. The sorption_type represents the empirically determined isotherm type and can have one of four possible values: {`"none"`, `"linear"`, `"freundlich"`, `"langmuir"`}. Linear isotherm needs just one parameter given whereas Freundlichs' and Langmuirs' isotherms require two parameters. We will use Freundlich's isotherm for demonstration but we will set the other parameter (exponent) $\alpha = 1$ which means it will be the same as the linear type.

Let suppose we have a sorption coefficient for uranium $K_d = 1.6{\cdot}10^{-4}\,\mathrm{kg}^{-1}\mathrm{m}^3$ (www.skb.se, report R-10-48 by James Crawford, 2010) and we want to use. We need to convert it to dimensionless value of isotherm_mult in the following way: $k_l = K_d M_s^{-1} \rho_l = K_d \frac{1000}{0.235} \approx 0.68$. For further details, see mathematical description in Section 2.5.2.

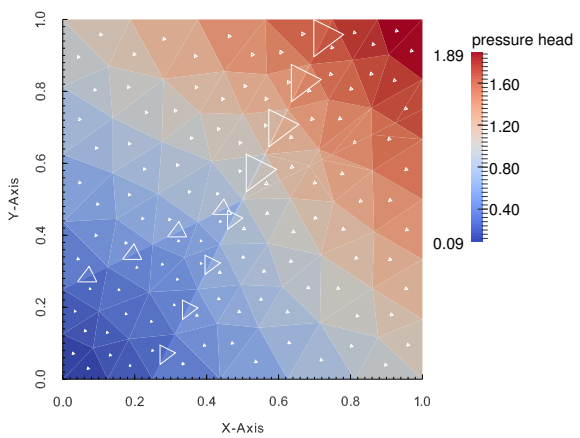
On line 97, notice the reference pointing to the definition of input fields on lines 81 – 89. Only entire records can be referenced which is why we have to repeat parts of the input such as solvent density and solubility (records for reaction mobile and reaction immobile have different types).

On lines 90 and 98, we define which sorption specific outputs are to be written to the output file. An implicit set of outputs exists. In this case we define an empty set of outputs thus overriding the implicit one. This means that no sorption specific outputs will be written to the output file. On lines 102 – 106 we specify which output fields should be written to the output stream. Currently, we support output into VTK and GMSH data format. In the output record for time-dependent process we have to specify the `time_step` (line 105) which determines the frequency of saving.

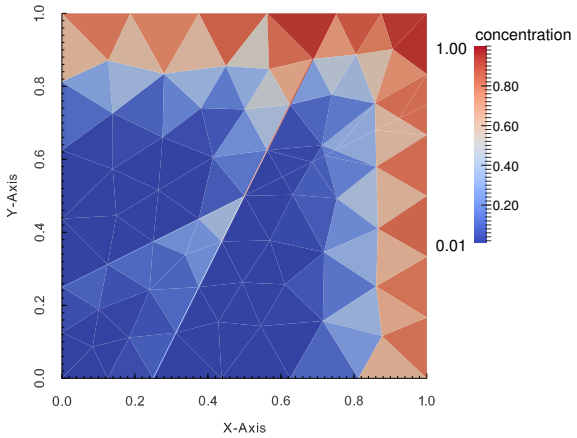
1.4.6 Results

In Figure 1.1 one can see the results: the pressure and the velocity field on the left and the concentration of U235 at time $t = 9 \cdot 10^5$ s on the right. Even if the pressure gradient is the same in the 2D domain and in the fracture, due to higher conductivity the velocity field is ten times faster in the fracture. Since porosity is the 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.

In the following chapter we describe mathematical models used in Flow123d. Then in chapter 4 we briefly describe structure of individual input files, in particular the main CON file. The complete description of the CON format is given in chapter 5.



(a) Elementwise pressure head and velocity field denoted by triangles. (Steady flow.)



(b) Propagation of U235 from the inflow part of the boundary. (At the time $9 \cdot 10^5$ s.)

Figure 1.1: Results of the tutorial problem.

Chapter 2

Mathematical Models of Physical Reality

Flow123d provides models for Darcy flow in porous media as well as for the transport and reactions of solutes. In this section, we describe mathematical formulations of these models together with physical meaning and units of all involved quantities. In the first section we present basic notation and assumptions about computational domains and meshes that combine different dimensions. In the next section we derive approximation of thin fractures by lower dimensional interfaces for a general transport process. Latter sections describe details for models of particular physical processes.

2.1 Meshes of Mixed Dimension

Unique feature common to all models in Flow123d is the support of domains with mixed dimension. Let $\Omega_3 \subset \mathbf{R}^3$ be an open set representing continuous approximation of porous and fractured medium. Similarly, we consider a set of 2D manifolds $\Omega_2 \subset \overline{\Omega}_3$, representing the 2D fractures and a set of 1D manifolds $\Omega_1 \subset \overline{\Omega}_2$ representing the 1D channels or preferential paths (see Fig 2.1). We assume that Ω_2 and Ω_1 are polytopic (i.e. polygonal and piecewise linear, respectively). 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 simplices, i.e. lines, triangles and tetrahedra, respectively.

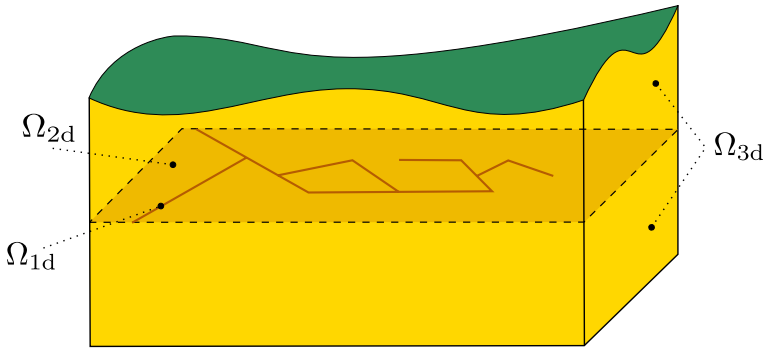


Figure 2.1: Scheme of a problem with domains of multiple dimensions.

Present numerical methods used by the software require meshes satisfying the compat-

ibility 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. Support for a coupling between non-compatible meshes of different dimesion is in developement and partly supported by the Darcy Flow model.

2.2 Advection-Diffusion Processes on Fractures

This section presents derivation of an abstract advection-diffusion process on 2D and 1D manifolds and its coupling with the higher dimensional domains. The reader not interested in the details of this approximation may skip directly to the later sections describing mathematical models of individual physical processes.

As was already mentioned, the unique feature of Flow123d is support of models living on 2D and 1D manifolds. The aim is to capture features significantly influencing the solution despite of their small cross-section. Such a tiny features are challenging for numerical simulations since a direct discretization requires highly refined computational mesh. One possible solution is to model these features (fractures, channels) as lower dimensional objects (2D and 1D manifolds) and introduce their coupling with the surrounding continuum. The equations modeling a physical process on a manifold as well as its coupling to the model in the surrounding continuum has to be derived from the model on the 3D continuum. This section presents such a procedure for the case of the abstract advection-diffusion process inspired by the paper [7]. Later, we this abstract approach to particular advection-diffusion processes: Darcian flow, solute transport, and heat transfer.

Let us consider a fracture as a strip domain

$$\Omega_f \subset [0, \delta] \times \mathbf{R}^{d-1}$$

for $d = 2$ or $d = 3$ and surrounding continuum domains

$$\Omega_1 \subset (-\infty, 0) \times \mathbf{R}^{d-1}, \Omega_2 \subset (\delta, \infty) \times \mathbf{R}^{d-1}.$$

Further, we denote by γ_i, $i = 1, 2$ the fracture faces common with domains Ω_1 and Ω_2 respectively. By x, $\boldsymbol{y}$ we denote normal and tangential coordinate of a point in Ω_f. We consider the normal vector $\boldsymbol{n} = \boldsymbol{n}_1 = -\boldsymbol{n}_2 = (1, 0, 0)^\top$. An advection-diffusion process is given by equations:

$$\partial_t w_i + \operatorname{div} \boldsymbol{j}_i = f_i \qquad \text{on } \Omega_i,\ i = 1, 2, f, \tag{2.3}$$

$$\boldsymbol{j}_i = -\mathbb{A}_i \nabla u_i + \boldsymbol{b}_i w_i \qquad \text{on } \Omega_i,\ i = 1, 2, f, \tag{2.4}$$

$$u_i = u_f \qquad \text{on } \gamma_i,\ i = 1, 2, \tag{2.5}$$

$$\boldsymbol{j}_i \cdot \boldsymbol{n} = \boldsymbol{j}_f \cdot \boldsymbol{n} \qquad \text{on } \gamma_i,\ i = 1, 2, \tag{2.6}$$

where $w_i = w_i(u_i)$ is the conservative quantity and u_i is the principal unknown, $\boldsymbol{j}_i$ is the flux of w_i, f_i is the source term, $\mathbb{A}_i$ is the diffusivity tensor and $\boldsymbol{b}_i$ is the velocity

field. We assume that the tensor $\mathbb{A}_f$ is symmetric positive definite with one eigenvector in the direction $\boldsymbol{n}$. Consequently the tensor has the form:

$$A_f = \begin{pmatrix} a_n & 0 \\ 0 & \mathbb{A}_t \end{pmatrix}$$

Furthermore, we assume that $\mathbb{A}_f(x, \boldsymbol{y}) = \mathbb{A}_f(\boldsymbol{y})$ is constant in the normal direction.

Our next aim is to integrate equations on the fracture Ω_f in the normal direction and obtain their approximations on the surface $\gamma = \Omega_f \cap \{x = \delta/2\}$ running through the middle of the fracture. For the sake of clarity, we will not write subscript f for quantities on the fracture. To make the following procedure mathematicaly correct we have to assume that functions $\partial_x w$, $\partial_x \nabla_{\boldsymbol{y}} u$, $\partial_x \boldsymbol{b}_{\boldsymbol{y}}$ are continuous and bounded on Ω_f. Here and later on $\boldsymbol{b}_x = (\boldsymbol{b} \cdot \boldsymbol{n})\,\boldsymbol{n}$ is the normal part of the velocity field and $\boldsymbol{b}_{\boldsymbol{y}} = \boldsymbol{b} - \boldsymbol{b}_x$ is the tangential part. The same notation will be used for normal and tangential part of the field $\boldsymbol{q}$.

We integrate (2.3) over the fracture opening $[0, \delta]$ and use approximations to get

$$\partial_t(\delta W) - \boldsymbol{j}_2 \cdot \boldsymbol{n}_2 - \boldsymbol{j}_1 \cdot \boldsymbol{n}_1 + \operatorname{div} \boldsymbol{J} = \delta F, \tag{2.7}$$

where for the first term, we have used mean value theorem, first order Taylor expansion, and boundedness of $\partial_x w$ to obtain approximation:

$$\int_0^\delta w(x, \boldsymbol{y})\,\mathrm{d}x = \delta w(\xi_{\boldsymbol{y}}, \boldsymbol{y}) = \delta W(\boldsymbol{y}) + O(\delta^2 |\partial_x w|),$$

where

$$W(\boldsymbol{y}) = w(\delta/2, \boldsymbol{y}) = w(u(\delta/2, \boldsymbol{y})) = w(U(\boldsymbol{y})).$$

Next two terms in (2.7) come from the exact integration of the divergence of the normal flux $\boldsymbol{j}_x$. Integration of the divergence of the tangential flux $\boldsymbol{j}_{\boldsymbol{y}}$ gives the fourth term, where we introduced

$$\boldsymbol{J}(\boldsymbol{y}) = \int_0^\delta \boldsymbol{j}_{\boldsymbol{y}}(x, \boldsymbol{y})\,\mathrm{d}x.$$

In fact, this flux on γ is scalar for the case $d = 2$. Finally, we integrate the right-hand side to get

$$\int_0^\delta f(x, \boldsymbol{y})\,\mathrm{d}x = \delta F(\boldsymbol{y}) + O(\delta^2 |\partial_x f|), \quad F(\boldsymbol{y}) = f(\delta/2, \boldsymbol{y}).$$

Due to the particular form of the tensor $\mathbb{A}_f$, we can separately integrate tangential and normal part of the flux given by (2.4). Integrating the tangential part and using approximations

$$\int_0^\delta \nabla_{\boldsymbol{y}} u(x, \boldsymbol{y})\,\mathrm{d}x = \delta \nabla_{\boldsymbol{y}} u(\xi_{\boldsymbol{y}}, \boldsymbol{y}) = \delta \nabla_{\boldsymbol{y}} U(\boldsymbol{y}) + O\big(\delta^2 |\partial_x \nabla_{\boldsymbol{y}} u|\big)$$

and

$$\int_0^\delta \big(\boldsymbol{b}_{\boldsymbol{y}} w\big)(x, \boldsymbol{y})\,\mathrm{d}x = \delta \boldsymbol{B}(\boldsymbol{y}) W(\boldsymbol{y}) + O\big(\delta^2 |\partial_x (\boldsymbol{b}_{\boldsymbol{y}} w)|\big)$$

where

$$\boldsymbol{B}(\boldsymbol{y}) = \boldsymbol{b}_{\boldsymbol{y}}(\delta/2, \boldsymbol{y}),$$

we obtain

$$\boldsymbol{J} = -\mathbb{A}_t \delta \nabla_{\boldsymbol{y}} U + \delta \boldsymbol{B} W + O\big(\delta^2(|\partial_x \nabla_{\boldsymbol{y}} u| + |\partial_x(\boldsymbol{b}_{\boldsymbol{y}} w)|)\big). \tag{2.8}$$

So far, we have derived equations for the state quantities U and $\boldsymbol{J}$ on the fracture manifold γ. In order to get a well possed problem, we have to prescribe two conditions for boundaries γ_i, $i = 1, 2$. To this end, we perform integration of the normal flux $\boldsymbol{j}_x$, given by (2.4), separately for the left and right half of the fracture. Similarly as before we use approximations

$$\int_0^{\delta/2} \boldsymbol{j}_x \,\mathrm{d}x = (\boldsymbol{j}_1 \cdot \boldsymbol{n}_1)\frac{\delta}{2} + O(\delta^2 |\partial_x \boldsymbol{j}_x|)$$

and

$$\int_0^{\delta/2} \boldsymbol{b}_x w \,\mathrm{d}x = (\boldsymbol{b}_1 \cdot \boldsymbol{n}_1)\tilde{w}_1 \frac{\delta}{2} + O(\delta^2 |\partial_x \boldsymbol{b}_x||w| + \delta^2 |\boldsymbol{b}_x||\partial_x w|)$$

and their counter parts on the interval $(\delta/2, \delta)$ to get

$$\boldsymbol{j}_1 \cdot \boldsymbol{n}_1 = -\frac{2a_n}{\delta}(U - u_1) + \boldsymbol{b}_1 \cdot \boldsymbol{n}_1 \tilde{w}_1 \tag{2.9}$$

$$\boldsymbol{j}_2 \cdot \boldsymbol{n}_2 = -\frac{2a_n}{\delta}(U - u_2) + \boldsymbol{b}_2 \cdot \boldsymbol{n}_2 \tilde{w}_2 \tag{2.10}$$

where $\tilde{w}_i$ can be any convex combination of w_i and W. Equations (2.9) and (2.10) have meaning of a semi-discretized flux from domains Ω_i into fracture. In order to get a stable numerical scheme, we introduce a kind of upwind already on this level using a different convex combination for each flow direction:

$$\begin{aligned} \boldsymbol{j}_i \cdot \boldsymbol{n}_i = &- \sigma_i(U - u_i) \\ &+ \big[\boldsymbol{b}_i \cdot \boldsymbol{n}_i\big]^+ \big(\xi w_i + (1-\xi)W\big) \\ &+ \big[\boldsymbol{b}_i \cdot \boldsymbol{n}_i\big]^- \big((1-\xi)w_i + \xi W\big), \qquad i = 1, 2 \end{aligned} \tag{2.11}$$

where $\sigma_i = \frac{2a_n}{\delta}$ is the transition coefficient and the parameter $\xi \in [\frac{1}{2}, 1]$ can be used to interpolate between upwind ($\xi = 1$) and central difference ($\xi = \frac{1}{2}$) scheme. Equations (2.7), (2.8), and (2.11) describe the general form of the advection-diffusion process on the fracture and its communication with the surrounding continuum which we shall later apply to individual processes.

2.3 Darcy Flow Model

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

$$\boldsymbol{w} = -\mathbb{K}\nabla H \quad \text{in } \Omega_d, \text{ for } d = 1, 2, 3. \tag{2.12}$$

Here and later on, we drop the dimension index d of the quantities if it can be deduced from the context. In (2.12), $\boldsymbol{w}$ [ms^{-1}] is the superficial velocity, $\mathbb{K}_d$ is the conductivity tensor, and H [m] is the piezometric head. The velocity $\boldsymbol{w}_d$ is related to the flux $\boldsymbol{q}_d$ [m^{4-d}s^{-1}] through

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

where δ_d [m^{3-d}] is the 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 $\boldsymbol{q}_d \cdot \boldsymbol{n}$ is the volume of the liquid (water) that passes 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 $k_d > 0$ [ms^{-1}] is the hydraulic conductivity and $\mathbb{A}_d$ is the 3×3 dimensionless anisotropy tensor which has to be symmetric and positive definite. The piezometric-head H_d is related to the pressure head h_d through

$$H_d = h_d + z \tag{2.13}$$

assuming that the gravity force acts in the negative direction of the z-axis. Combining these relations, we get the Darcy law in the form:

$$\boldsymbol{q} = -\delta k \mathbb{A} \nabla (h + z) \qquad \text{in } \Omega_d, \text{ for } d = 1, 2, 3. \tag{2.14}$$

Next, we employ the continuity equation for saturated porous medium and the dimensional reduction from the preceding section (with $w = u := H$, $\boldsymbol{j} := \boldsymbol{w}$, $\mathbb{A} := \mathbb{K}$ and $\boldsymbol{b} := \boldsymbol{0}$), which yields:

$$\partial_t (\delta S\, h) + \mathrm{div} \boldsymbol{q} = F \qquad \text{in } \Omega_d, \text{ for } d = 1, 2, 3, \tag{2.15}$$

where S_d [m^{-1}] is the storativity and F_d [$m^{3-d} s^{-1}$] is the source term. In our setting the principal unknowns of the system (2.14, 2.15) are the pressure head h_d and the flux $\boldsymbol{q}_d$.

The storativity (or the volumetric specific storage) $S_d > 0$ can be expressed as

$$S_d = \gamma_w (\beta_r + \vartheta \beta_w), \tag{2.16}$$

where γ_w [$kgm^{-2}s^{-2}$] is the specific weight of water, ϑ [$-$] 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 on the right hand side of (2.15) consists of the volume density of the water source f_d[s^{-1}] and flux from the from the higher dimension. Precise form of F_d slightly differs for every dimension and will be discussed presently.

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

In the set $\Omega_2 \cap \Omega_3$ the fracture is surrounded by at most one 3D surface from every side. On $\partial\Omega_3 \cap \Omega_2$ we prescribe a boundary condition of the 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 in Ω_3, h_2 is the pressure head in Ω_2, and σ_3 [s^{-1}] is the transition coefficient given by (see section 2.2 and [7])

$$\sigma_3 = \sigma_{32} \frac{2\mathbb{K}_2 : \boldsymbol{n}_2 \otimes \boldsymbol{n}_2}{\delta_2}.$$

Here $\boldsymbol{n}_2$ is the unit normal to the fracture (sign does not matter). On the other hand, the sum of the interchange fluxes $q_{32}^{+/-}$ forms a volume source in Ω_2. Therefore F_2 [ms^{-1}] on the right hand side of (2.15) is given by

$$F_2 = \delta_2 f_2 + (q_{32}^+ + q_{32}^-). \tag{2.17}$$

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

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

where σ_2 [ms^{-1}] is the transition coefficient integrated over the width of the fracture i:

$$\sigma_2 = \sigma_{21} \frac{2\delta_2^2 \mathbb{K}_1 : \boldsymbol{n}_1^i \otimes \boldsymbol{n}_1^i}{\delta_1}.$$

Here $\boldsymbol{n}_1^i$ is the unit normal to the channel that is tangential to the fracture i. Sum of the fluxes forms a part of F_1 [m^{2}s^{-1}]:

$$F_1 = \delta_1 f_1 + \sum_{i=1}^{n} q_{21}^i. \tag{2.18}$$

We remark that the direct communication between 3D and 1D (e.g. model of a well) is not supported yet. The transition coefficients σ_{32} [$-$] and σ_{21} [$-$] are independent scaling parameters which represent the ratio of the crosswind and the tangential conductivity in the fracture. For example, in the case of impermeable film on the fracture walls one may choice $\sigma_{32} < 1$.

In order to obtain unique solution we have to prescribe boundary conditions. Currently we support three basic types of boundary conditions:

Dirichlet boundary condition

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

where h_d^D [m] is the boundary pressure head . Alternatively one can prescribe the boundary piezometric head H_d^D [m] related to the pressure head through (2.13).

Neuman boundary condition

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

where q_d^N [m^{4-d}s^{-1}] is the surface density of the water outflow.

Robin boundary condition

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

where h_d^R is the boundary pressure head and σ_d^R [m^{3-d}s^{-1}] is the transition coefficient. As before one can also prescribe the boundary piezo head H_d^R to specify h_d^R.

We consider a disjoint decomposition of the boundary

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

into Dirichlet, Neumann, and Robin part. In order to obtain well posed steady state problem one have to prescribe Dirichlet or Robin boundary condition on part of boundary that is connected (geometricaly of through the inter dimensional coupling) to hte rest of the domain.

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

Volume balance. The equation (2.15) satisfies the volume balance of the liquid in the following form:

$$V(0) + \int_0^t s(\tau)\,d\tau - \int_0^t f(\tau)\,d\tau = V(t)$$

for any instant t in the computational time interval. Here

$$V(t) := \sum_{d=1}^{3} \int_{\Omega^d} (\delta Sh)(t, \boldsymbol{x})\,d\boldsymbol{x},$$

$$s(t) := \sum_{d=1}^{3} \int_{\Omega^d} F(t, \boldsymbol{x})\,d\boldsymbol{x},$$

$$f(t) := \sum_{d=1}^{3} \int_{\partial\Omega^d} \boldsymbol{q}(t, \boldsymbol{x}) \cdot \boldsymbol{n}(\boldsymbol{x})\,d\boldsymbol{x}$$

is the volume [m^3], the volume source [m^3s^{-1}] and the volume flux [m^3s^{-1}] of the liquid at time t, respectively. The volume, flux and source on every geometrical region is calculated at each output time and the values together with the control sums are written to the file `water_balance.{dat|txt}`. If, in addition, cumulative is set to true then the time-integrated flux and source is written.

2.4 Transport of Substances

The motion of substances dissolved in water is governed by the *advection*, and the *hydrodynamic dispersion*. In Ω_d, $d \in \{1, 2, 3\}$, we consider the following system of mass balance equations[1]:

$$\partial_t(\delta\vartheta c^i) + \operatorname{div}(\boldsymbol{q}c^i) - \operatorname{div}(\vartheta\delta\mathbb{D}^i\nabla c^i) = F_S^i + F_C^c + F_R(c^1, \ldots, c^s). \tag{2.19}$$

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

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

$$\mathbb{D}^i = D_m^i\tau\mathbb{I} + |\boldsymbol{v}|\left(\alpha_T^i\mathbb{I} + (\alpha_L^i - \alpha_T^i)\frac{\boldsymbol{v}\otimes\boldsymbol{v}}{|\boldsymbol{v}|^2}\right), \tag{2.20}$$

  which represents (isotropic) molecular diffusion, and mechanical dispersion in longitudal and transversal direction to the flow. Here D_m^i [m^2s^{-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 [8]), α_L^i [m] and α_T^i [m] is the longitudal dispersivity and the transverse dispersivity, respectively. Note that although we allow

[1] For $d \in \{1, 2\}$ this form can be derived as in Section 2.2 using $w := \delta\vartheta c^i$, $u := c^i$, $\mathbb{A} := \delta\vartheta\mathbb{D}^i$, $\boldsymbol{b} := \boldsymbol{v} = \frac{\boldsymbol{q}}{\vartheta\delta}$.

dispersivities to have different values for different substances, it is often assumed that they are intrinsic parameters of the porous medium. Finally, $\boldsymbol{v}$ [ms^{-1}] is the *microscopic* water velocity, related to the Darcy flux $\boldsymbol{q}$ by the relation $\boldsymbol{q} = \vartheta\delta\boldsymbol{v}$. The value of D_m^i for specific substances can be found in literature (see e.g. [2]). For instructions on how to determine α_L^i, α_T^i we refer to [3, 4].

- F_S^i [kgm^{-d}s^{-1}] represents the density of concentration sources in the porous medium. Its form is:
$$F_S^i = \delta f_S^i + \delta(c_S^i - c^i)\sigma_S^i. \tag{2.21}$$
Here f_S^i [kgm^{-3}s^{-1}] is the density of concentration sources, c_S^i [kgm^{-3}] is an equilibrium concentration and σ_S^i [s^{-1}] is the concentration flux. One has to pay attention when prescribing the source, namely to determine whether it is related to the *liquid* or the *porous medium*. We mention several examples:
  - extraction of solution: $f_S^i = 0$, $c_S^i = 0$, $\sigma_S^i > 0$ is the intensity of extraction, i.e. volume of liquid extracted from a unit volume of porous medium per second;
  - injection of solution: $f_S^i = 0$, c_S^i is the concentration of the substance in the injected liquid, $\sigma_S^i > 0$ is the intensity of injection (volume of liquid injected into a unit volume of porous medium per second);
  - production or degradation of substances due to bacteria present in liquid: $f_S^i = \vartheta p^i$, where p^i is the production/degradation rate in a unit volume of liquid;
  - age of liquid: if $f_S^i = \vartheta$ then c^i is the age of liquid, i.e. the time spent in the domain.
- F_C^c [kgm^{-d}s^{-1}] is the density of concentration sources due to exchange between regions with different dimensions, see (2.23) below.
- The reaction term $F_R(\dots)$ [kgm^{-d}s^{-1}] is thoroughly described in the next section 2.5, see also paragraph "Two transport models" below.

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 the parts $\Gamma_I \cup \Gamma_D \cup \Gamma_N \cup \Gamma_R$, which may change during simulation time. The first part Γ_I is further divided into two segments:
$$\Gamma_I^+(t) = \{\boldsymbol{x} \in \partial\Omega_d \,|\, \boldsymbol{q}(t, \boldsymbol{x}) \cdot \boldsymbol{n}(\boldsymbol{x}) < 0\},$$
$$\Gamma_I^-(t) = \{\boldsymbol{x} \in \partial\Omega_d \,|\, \boldsymbol{q}(t, \boldsymbol{x}) \cdot \boldsymbol{n}(\boldsymbol{x}) \geq 0\},$$
where $\boldsymbol{n}$ stands for the unit outward normal vector to $\partial\Omega_d$. We prescribe the following boundary conditions: On Γ_D, the Dirichlet condition is imposed via prescribed concentration c_D^i:
$$c^i = c_D^i \text{ on } \Gamma_D.$$
On the inflow Γ_I^+ the reference concentration c_D^i is enforced through total flux:
$$(\boldsymbol{q}c^i - \vartheta\delta\mathbb{D}^i\nabla c^i) \cdot \boldsymbol{n} = \boldsymbol{q} \cdot \boldsymbol{n}c_D^i \text{ on } \Gamma_I^+,$$

on Γ_I^- we impose homogeneous Neumann condition:

$$-\vartheta\delta\mathbb{D}^i\nabla c^i\cdot\boldsymbol{n}=0 \text{ on } \Gamma_I^-,$$

on Γ_N we impose Neumann condition with user-defined concentration flux f_N^i:

$$-\vartheta\delta\mathbb{D}^i\nabla c^i\cdot\boldsymbol{n}=f_N^i \text{ on } \Gamma_N,$$

and finally on Γ_R we impose Robin condition through transition parameter σ_R^i and reference concentration c_D^i:

$$-\vartheta\delta\mathbb{D}^i\nabla c^i\cdot\boldsymbol{n}=\sigma_R^i(c^i-c_D^i) \text{ on } \Gamma_R.$$

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 the quantity

$$q_{d+1,d}^c=\sigma_{d+1,d}^c\frac{\delta_{d+1}^2}{\delta_d}2\vartheta_d\mathbb{D}_d:\boldsymbol{n}\otimes\boldsymbol{n}(c_{d+1}-c_d)+\begin{cases}q_{d+1,d}^l c_{d+1} & \text{if } q_{d+1,d}^l\geq 0,\\ q_{d+1,d}^l\frac{\vartheta_d}{\vartheta_{d+1}}c_d & \text{if } q_{d+1,d}^l<0,\end{cases}\tag{2.22}$$

where

- $q_{d+1,d}^c$ [kgm^{-d}s^{-1}] is the density of concentration flux from Ω_{d+1} to Ω_d,
- $\sigma_{d+1,d}^c$ [$-$] is a transition parameter. Its value determines the mass exchange between dimensions whenever the concentrations differ. In general, it is recommended to leave the default value $\sigma^c=1$ or to set $\sigma^c=0$ (when exchange is due to water flux only).
- $q_{d+1,d}^l$ [m^{3-d}s^{-1}] is the water flux from Ω_{d+1} to Ω_d, i.e. $q_{d+1,d}^l=\boldsymbol{q}_{d+1}\cdot\boldsymbol{n}_{d+1}$.

The communication between dimensions is incorporated as the total flux boundary condition for the problem on Ω_{d+1}:

$$-\vartheta\delta\mathbb{D}\nabla c\cdot\boldsymbol{n}+q^l c=q^c\tag{2.23}$$

and a source term in Ω_d:

$$F_{C3}^c=0,\quad F_{C2}^c=q_{32}^c,\quad F_{C1}^c=q_{21}^c.\tag{2.24}$$

Two transport models. Within the above presented model, Flow123d presents two possible approaches to solute transport.

- For modelling pure advection ($\mathbb{D}=0$) one can choose `TransportOperatorSplitting` method, which represents an explicit in time finite volume solver. Only the inflow/outflow boundary condition is available and the source term has the form

  $$F_S^i=\delta f_S^i+\delta(c_S^i-c^i)^+\sigma_S^i.$$

  The solution process for one time step is faster, but the maximal time step is restricted. The resulting concentration is piecewise constant on mesh elements. This solver supports reaction term (involving simple chemical reactions, dual porosity and sorption).

- The full model including dispersion is solved by `SoluteTransport_DG`, an implicit in time discontinuous Galerkin solver. It has no restriction of the computational time step and the space approximation is piecewise polynomial, currently up to order 3. Reaction term is implemented only for the case of linear sorption, i.e.

$$F_R^i = -\partial_t \left((1-\vartheta)\delta M^i \varrho_s c_s^i \right), \quad c_s^i = \frac{k_l^i}{\varrho_l} c,$$

where c_s^i [mol kg^{-1}] is the concentration of sorbed substance, k_l^i [mol kg^{-1}] is the sorption coefficient, ϱ_s and ϱ_l [kgm^{-3}] is the density of the solid (rock) and of the liquid (solvent), respectively, and M^i [kg mol^{-1}] denotes the molar mass of the i-th substance. The initial concentration in solid is assumed to be in equilibrium with the concentration in liquid.

Mass balance. The advection-dispersion equation satisfies the balance of mass in the following form:

$$m^i(0) + \int_0^t s^i(\tau)\, d\tau - \int_0^t f^i(\tau)\, d\tau = m^i(t)$$

for any instant t in the computational time interval and any substance i. Here

$$m^i(t) := \sum_{d=1}^{3} \int_{\Omega^d} (\delta\vartheta c^i)(t, \boldsymbol{x})\, d\boldsymbol{x},$$

$$s^i(t) := \sum_{d=1}^{3} \int_{\Omega^d} F_S^i(t, \boldsymbol{x})\, d\boldsymbol{x},$$

$$f^i(t) := \sum_{d=1}^{3} \int_{\partial\Omega^d} \left(\boldsymbol{q} c^i - \vartheta\delta \mathbb{D}^i \nabla c^i \right)(t, \boldsymbol{x}) \cdot \boldsymbol{n}\, d\boldsymbol{x}$$

is the mass [kg], the volume source [kgs^{-1}] and the mass flux [kgs^{-1}] of i-th substance at time t, respectively. The mass, flux and source on every geometrical region is calculated at each output time and the values are written to the file `mass_balance.{dat|txt}`. If, in addition, cumulative is set to true then the time-integrated flux and source is written. In that case the cumulative source contains also contribution due to reactions.

2.5 Reaction Term in Transport

The `TransportOperatorSplitting` method supports the reaction term $F_R(c^1, \ldots, c^s)$ on the right hand side of the equation (2.19). It can represent several models of chemical or physical nature. Figure 2.2 shows all possible reactional models that we support in combination with the transport process. The Operator Splitting method enables us to deal with the convection part and reaction term side by side. The convected quantities do not influence each other in the convectional process and are balanced over the elements. On the other hand the reaction term relates the convected quantities and can be computed separately on each element.

We move now to the description of the reaction models which can be seen again in Figure 2.2. The convected quantity is considered to be the concentration of substances.

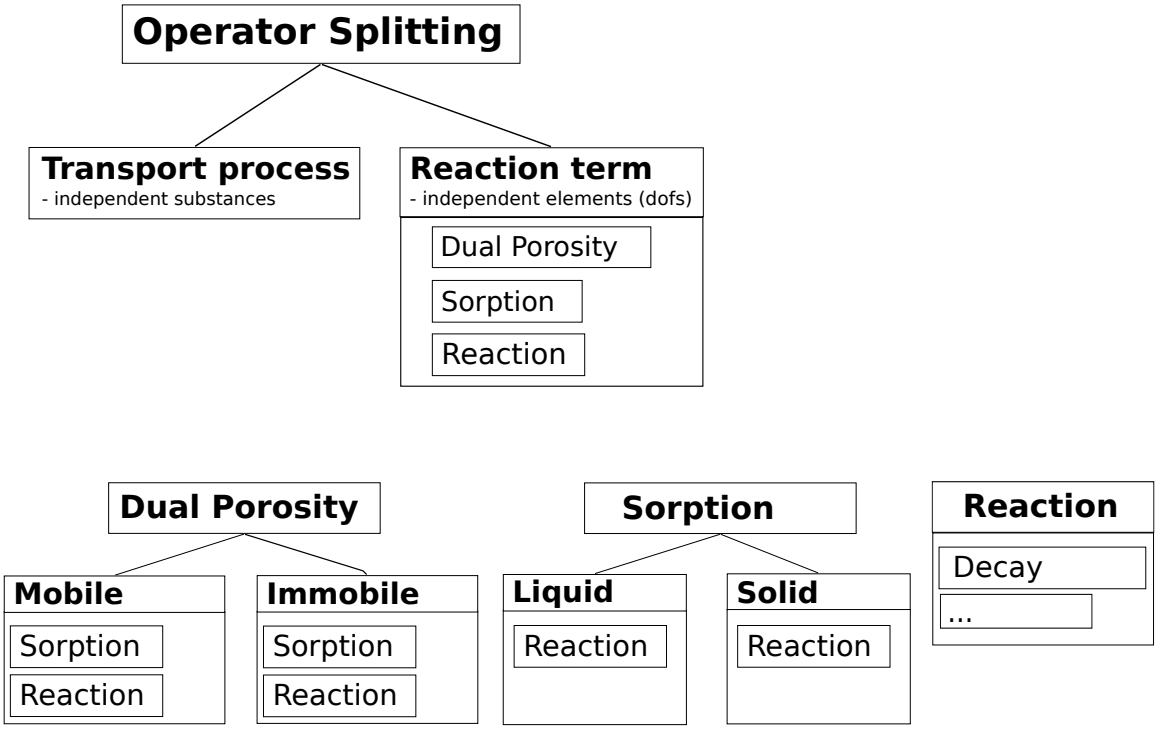


Figure 2.2: The scheme of the reaction term objects. The lines represents connections between different models. The tables under model name include the possible models which can be connected to the model above.

Up to now we can have *dual porosity*, *sorption* (these two are more of a physical nature) and (chemical) *reaction* models in the reaction term.

The *reaction* model acts only on the specified substances and computes exchange of concentration among them. It does not have its own output because it only changes the concentration of substances in the specified zone where the reaction takes place.

The *sorption* model describes the exchange of concentration of the substances between liquid and solid. It can be followed by another *reaction* that can run in both phases. The concentration in solid is an additional output of this model. See Subsection 2.5.2.

The *dual porosity* model, described in Subsection 2.5.1, introduces the so called immobile (or dead-end) pores in the matrix. The convection process operates only on the concentration of the substances in the mobile zone (open pores) and the exchange of concentrations from/to immobile zone is governed by molecular diffusion. This process can be followed by *sorption* model and/or chemical *reaction*, both in mobile and immobile zone. The immobile concentration is an additional output.

2.5.1 Dual Porosity

Up to now, we have described the transport equation for the single porosity model. The dual porosity model splits the mass into two zones – the mobile zone and the immobile zone. Both occupy the same macroscopic volume, however on the microscopic scale, the immobile zone is formed by the dead-end pores, where the liquid is trapped and cannot pass through. The rest of the pore volume is occupied by the mobile zone. Since the liquid in the immobile pores is immobile, the exchange of the substance is only due to molecular diffusion. We consider simple nonequilibrium linear model:

$$\vartheta_m \partial_t c_m = D_{dp}(c_i - c_m), \tag{2.25a}$$

$$\vartheta_i \partial_t c_i = D_{dp}(c_m - c_i), \tag{2.25b}$$

where c_m is the concentration in the mobile zone, c_i is the concentration in the immobile zone and D_{dp} is a diffusion rate between the zones. ϑ_i denotes porosity of the immobile zone and $\vartheta_m = \vartheta$ the mobile porosity from transport equation (2.19). One can also set non-zero initial concentration in the immobile zone $c_i(0)$.

To solve the system of first order differential equation, we use analytic solution or Euler's method, which are switched according to a given error tolerance. See subsection 3.4.1 in numerical methods.

2.5.2 Equilibrial Sorption

The simulation of monolayer, equilibrial sorption is based on the solution of two algebraic equations, namely the mass balance (in unit volume)

$$\vartheta \varrho_l c_l + (1 - \vartheta)\varrho_s M_s c_s = c_T = const. \tag{2.26}$$

and an empirical sorption law

$$c_s = f(c_l), \tag{2.27}$$

given in terms of the so-called isotherm f. Its form is determined by the parameter sorption type:

- "*none*": $f(c_l) = 0$ (the sorption model returns zero concentration in solid);
- "*linear*": $f(c_l) = k_l c_l$;
- "$freundlich$": $f(c_l) = k_F c_l^\alpha$;
- "*langmuir*": $f(c_l) = k_L \frac{\alpha c_l}{1+\alpha c_l}$. Langmuir isotherm has been derived from thermodynamic laws. k_L denotes the maximal amount of sorbing specie which can be kept in an unit volume of a bulk matrix. Coefficient α is a fraction of sorption and desorption rate constant $\alpha = \frac{k_a}{k_d}$.

Notation:

- In solid, $c_s = \frac{n}{m_s}$ $[\mathrm{mol\,kg^{-1}}]$ is the fraction of the molar amount of the solute adsorbed n and the amount of the adsorbent m_s (mass of solid), all in unit volume. The concentration in solid can be selected for output.
- In liquid, $c_l = \frac{m}{m_l}$ $[-]$ is the fraction of the amount of the solute m and the mass of liquid m_l, all in unit volume. The relation between c_l and the concentration c from transport equation (2.19) is $c = c_l \varrho_l$.
- ϱ_l, ϱ_s is the liquid (solvent) density and the solid (rock) density, respectively.
- M_s denotes the molar mass of a substance.
- Multiplication parameters are $k_i, i \in \{l, F, L\}$ $[\mathrm{mol\,kg^{-1}}]$.
- Additional parameter $[\alpha] = 1$ can be set.

Non-zero initial concentration in the solid phase $c_s(0)$ can be set in the input record. Now, further denoting

$$\mu_l = \varrho_l \vartheta, \quad \mu_s = M_s \varrho_s \cdot (1 - \vartheta),$$

and using (2.27), the mass balance (2.26) reduces to the equation

$$c_T = \mu_l c_l + \mu_s f(c_l), \tag{2.28}$$

which can be either solved iteratively or using interpolation. See subsection 3.4.2 in numerical methods for details.

The units of c_l, c_s and k_i can vary in literature. To avoid misinterpretation, we derive (according to Bowman [1]) a conversion rule for Freundlich isotherm which will lead the user also in other cases, we believe.

Units conversion. Let us have c $[\mathrm{kgm^{-3}}]$, the mass concentration in liquid, and s $[\mathrm{kg\,kg^{-1}}]$, the fraction of the amount of the solute adsorbed and the amount of the adsorbent in solid. The unit of K follows from the dimensional analysis of $s = Kc^\alpha$:

$$[K] = \frac{\mathrm{kg}^{1-\alpha}\mathrm{m}^{3\alpha}}{\mathrm{kg}},$$

which we want to convert to k_F $[\mathrm{mol\,kg^{-1}}]$ in the formula $c_s = k_F c_l^\alpha$.

The first step is a conversion of the mass of the solute to moles by dividing it by the molar mass M_s. We then have the formula

$$\begin{aligned} s &= Kc^\alpha \\ \frac{s}{M_s} &= K'\left(\frac{c}{M_s}\right)^\alpha, \\ s &= K'M_s^{1-\alpha}c^\alpha, \end{aligned} \tag{2.29}$$

where $s = c_s M_s$ and $K' = KM_s^{\alpha-1}$ $[\mathrm{mol}^{1-\alpha}\mathrm{kg}^{-1}\mathrm{m}^{3\alpha}]$ is a new constant, distributing the molar concentration in liquid to the ratio of the molar mass and the amount of sorbent in solid.

The second step is introducing $c_l = \frac{c}{\varrho_l}$ into the formula (2.29)

$$c_s = K'\left(\frac{c_l\rho_l}{M_s}\right)^\alpha = K'M_s^{-\alpha}\rho_l^\alpha c_l^\alpha = \left(KM_s^{-1}\rho_l^\alpha\right)c_l^\alpha, \tag{2.30}$$

where we can denote

$$k_F = KM_s^{-1}\rho_l^\alpha, \tag{2.31}$$

which is the constant we are looking for. This can be also translated to the case of the linear isotherm, where $\alpha = 1$ and $[K] = \mathrm{kg}^{-1}\mathrm{m}^3$, and we get the conversion rule

$$k_l = KM_s^{-1}\rho_l. \tag{2.32}$$

The conversion of different prefixes of units are left on the user. One should be careful using the Freundlich isotherm, though, where the exponent α must not be forgotten.

2.5.3 Sorption in Dual Porosity Model

There are two parameters μ_l and μ_s, scale of aqueous concentration and scale of sorbed concentration, respectively. There is a difference in computation of these in the dual porosity model because both work on different concentrations and different zones.

Let c_{ml} and c_{ms} be concentration in liquid and in solid in the mobile zone, c_{il} and c_{is} be concentration in liquid and in solid in the immobile zone, ϑ_m and ϑ_i be the mobile and the immobile porosity, and φ be the sorbing surface.

The sorbing surface in the mobile zone is given by

$$\varphi = \frac{\vartheta_m}{\vartheta_m + \vartheta_i}, \tag{2.33}$$

while in the immobile zone it becomes

$$1 - \varphi = 1 - \frac{\vartheta_m}{\vartheta_m + \vartheta_i} = \frac{\vartheta_i}{\vartheta_m + \vartheta_i}.$$

Remind the mass balance equation (2.28). In the dual porosity model, the scaling parameters μ_l, μ_s are slightly different. In particular, the mass balance in the mobile zone reads:

$$\begin{aligned} c_T &= \mu_l \cdot c_{ml} + \mu_s \cdot c_{ms}, \\ \mu_l &= \varrho_l \cdot \vartheta_m, \\ \mu_s &= M_s \cdot \varrho_s \cdot (1 - \vartheta_m - \vartheta_i)\varphi, \end{aligned} \tag{2.34}$$

while in the immobile zone it has the form:

$$\begin{aligned} c_T &= \mu_l \cdot c_{il} + \mu_s \cdot c_{is}, \\ \mu_l &= \varrho_l \cdot \vartheta_i, \\ \mu_s &= M_s \cdot \varrho_s \cdot (1 - \vartheta_m - \vartheta_i)(1 - \varphi). \end{aligned} \tag{2.35}$$

2.5.4 Radioactive Decay

The radioactive decay is one of the processes that can be modelled in the reaction term of the transport model. This process is coupled with the transport using the operator splitting method. It can run throughout all the phases, including the mobile and immobile phase of the liquid and also the sorbed solid phase, as it can be seen in figure 2.2.

The radioactive decay of a parent radionuclide A to a nuclid B

$$A \xrightarrow{k} B, \qquad A \xrightarrow{t_{1/2}} B$$

is mathematicaly formulated as a system of first order differential equations

$$\frac{\mathrm{d}c_A}{\mathrm{d}\tau} = -kc_A, \tag{2.36}$$

$$\frac{\mathrm{d}c_B}{\mathrm{d}\tau} = kc_A, \tag{2.37}$$

where k is the radioactive decay rate. Usually, the half life of the parent radionuclide $t_{1/2}$ is known rather than the rate. Relation of these can be derived:

$$\begin{aligned} \frac{\mathrm{d}c_A}{\mathrm{d}\tau} &= -kc_A \\ \frac{\mathrm{d}c_A}{c_A} &= -k\,\mathrm{d}\tau \\ \int_{c_A^0}^{c_A^0/2} \frac{\mathrm{d}c_A}{c_A} &= -k \int_0^{t_{1/2}} 1\,\mathrm{d}\tau \\ \left[\ln c_A\right]_{c_A^0}^{c_A^0/2} &= -\left[k\tau\right]_0^{t_{1/2}} \\ k &= \frac{\ln 2}{t_{1/2}}. \end{aligned}$$

Let us now suppose a more complex situation. Consider substances (radionuclides) $A_1, \ldots, A_s$ which take part in a complex radioactive chain, including branches, e.g.

$$\begin{array}{lllll} A_1 \xrightarrow{k_1} A_2 \xrightarrow{k_2} & A_3 & \xrightarrow{k_{34}} & A_4 \xrightarrow{k_4} & A_8 \\ & A_3 \xrightarrow{k_{35}} & A_5 \xrightarrow{k_5} & A_4 & \\ & A_3 \xrightarrow{k_{36}} & A_6 \xrightarrow{k_6} A_7 \xrightarrow{k_7} & & A_8 \end{array}$$

Now the problem turned into a system of differential equations $\partial_t \boldsymbol{c} = \mathbf{D}\boldsymbol{c}$ with the

following matrix, generally full and nonsymmetric:

$$\mathbf{D} = \begin{pmatrix} M_1 & & & \\ & M_2 & & \\ & & \ddots & \\ & & & M_s \end{pmatrix} \begin{pmatrix} -k_1 & k_{21} & \cdots & k_{s1} \\ k_{12} & -k_2 & \cdots & k_{s2} \\ \vdots & \vdots & \ddots & \vdots \\ k_{1s} & k_{2s} & \cdots & -k_s \end{pmatrix} \begin{pmatrix} \frac{1}{M_1} & & & \\ & \frac{1}{M_2} & & \\ & & \ddots & \\ & & & \frac{1}{M_s} \end{pmatrix},$$

where M_i is molar mass. We can then write

$$d_{ij} = \begin{cases} k_{ji}\frac{M_i}{M_j}, & i \neq j, \\ -k_{ij}, & i = j. \end{cases} \tag{2.38}$$

We denote the rate constant of the i-th radionuclide

$$k_i = \sum_{j=1}^{s} k_{ij} = \sum_{j=1}^{s} b_{ij} k_i$$

which is equal to a sum of partial rate constants k_{ij}. Branching ratio $b_{ij} \in (0, 1)$ determines the distribution into different branches of the decay chain, holding $\sum_{j=1}^{s} b_{ij} = 1$.

Notice, that physically it is not possible to create a chain loop, so in fact one can permutate the vector of concentrations and rearrange the matrix D into a lower triangle matrix

$$\mathbf{D} = \begin{pmatrix} d_{11} & & & \\ d_{21} & d_{22} & & \\ \vdots & \vdots & \ddots & \\ d_{s1} & d_{s2} & \cdots & d_{ss} \end{pmatrix}.$$

However, we do not do this and we do not search the reactions for chain loops.

The system of first order differential equations with constant coefficients is solved using one of the implemented linear ODE solvers, described in section 3.4.3.

2.5.5 First Order Reaction

First order kinetic reaction is another process that can take part in the reaction term. Similarly to the radioactive decay, it is connected to transport by operator splitting method and can run in all the possible phases, see figure 2.2.

Currently, reactions with single reactant and multiple products (decays) are available in the software. The mathematical description is the same as for the radioactive decay, it only uses kinetic reaction rate coefficient k in the input instead of half life.

2.6 Heat Transfer

Under the assumption of thermal equilibrium between the solid and liquid phase, the energy balance equation has the form[2]

$$\partial_t \left(\delta \tilde{s} T\right) + \operatorname{div}(\varrho_l c_l T \boldsymbol{q}) - \operatorname{div}(\delta \Lambda \nabla T) = F^T + F_C^T.$$

[2]For lower dimensions this form can be derived as in Section 2.2 using $w := \delta\tilde{s}T$, $u := T$, $\mathbb{A} := \delta\lambda\mathbb{I}$, $\boldsymbol{b} := \frac{\varrho_l c_l}{\tilde{s}}\boldsymbol{w}$.

The principal unknown is the temperature T [K]. Other quantities are:

- ϱ_l, ϱ_s [kgm^{-3}] is the density of the fluid and solid phase, respectively.
- c_l, c_s [Jkg^{-1}K^{-1}] is the heat capacity of the fluid and solid phase, respectively.
- $\tilde{s}$ [Jm^{-3}K^{-1}] is the volumetric heat capacity of the porous medium defined as

$$\tilde{s} = \vartheta \varrho_l c_l + (1-\vartheta)\varrho_s c_s.$$

- Λ [Wm^{-1}K^{-1}] is the thermal dispersion tensor:

$$\Lambda = \Lambda^{cond} + \Lambda^{disp}$$

$$\Lambda^{cond} = \left(\vartheta \lambda_l^{cond} + (1-\vartheta)\lambda_s^{cond}\right)\mathbb{I},$$

$$\Lambda^{disp} = \vartheta \varrho_l c_l |\boldsymbol{v}| \left(\alpha_T \mathbb{I} + (\alpha_L - \alpha_T)\frac{\boldsymbol{v}\otimes\boldsymbol{v}}{|\boldsymbol{v}|^2}\right),$$

  where λ_l^{cond}, λ_s^{cond} [Wm^{-1}K^{-1}] is the thermal conductivity of the fluid and solid phase, respectively, and α_L, α_T [m] is the longitudal and transverse dispersivity in the fluid.

- F^T [Jm^{-d}s^{-1}] represents the thermal source:

$$F^T = \delta\vartheta F_l^T + \delta(1-\vartheta)F_s^T,$$

$$F_l^T = f_l^T + \varrho_l c_l \sigma_l^T (T - T_l),$$
$$F_s^T = f_s^T + \varrho_s c_s \sigma_s^T (T - T_s),$$

  where f_l^T, f_s^T [Wm^{-3}] is the density of thermal sources in fluid and solid, respectively, T_l, T_s [K] is a reference temperature and σ_l^T, σ_s^T [s^{-1}] is the heat exchange rate.

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

$$T(0, \boldsymbol{x}) = T_0(\boldsymbol{x}).$$

Given the decomposition of $\partial\Omega_d$ into $\Gamma_I \cup \Gamma_D \cup \Gamma_N \cup \Gamma_R$ (see Section 2.4), we prescribe the following boundary conditions:

- Dirichlet:

$$T = T_D \text{ on } \Gamma_I^+ \cup \Gamma_D,$$

- Homogeneous Neumann:

$$(\varrho_l c_l T\boldsymbol{q} - \delta\Lambda\nabla T)\cdot\boldsymbol{n} = 0 \text{ on } \Gamma_I^-,$$

- Neumann:

$$(\varrho_l c_l T\boldsymbol{q} - \delta\Lambda\nabla T)\cdot\boldsymbol{n} = f_N \text{ on } \Gamma_N,$$

- Robin (Newton):

$$(\varrho_l c_l T\boldsymbol{q} - \delta\Lambda\nabla T)\cdot\boldsymbol{n} = \sigma_R(T - T_D) \text{ on } \Gamma_R.$$

Communication between dimensions. Denoting T_{d+1}, T_d the temperature in Ω_{d+1} and Ω_d, respectively, the communication on the interface between Ω_{d+1} and Ω_d is described by the quantity

$$q^T_{d+1,d} = \sigma^T_{d+1,d} \frac{\delta^2_{d+1}}{\delta_d} 2\Lambda_d : \boldsymbol{n} \otimes \boldsymbol{n}(T_{d+1} - T_d) + \begin{cases} \varrho_l c_l q^l_{d+1,d} T_{d+1} & \text{if } q^l_{d+1,d} \geq 0, \\ \varrho_l c_l q^l_{d+1,d} \frac{\tilde{s}_d}{\tilde{s}_{d+1}} T_d & \text{if } q^l_{d+1,d} < 0, \end{cases} \tag{2.39}$$

where

- $q^T_{d+1,d}$ [Wm^{-2}] is the density of heat flux from Ω_{d+1} to Ω_d,
- $\sigma^T_{d+1,d}$ [$-$] is a transition parameter. Its value determines the exchange of energy between dimensions due to temperature difference. In general, it is recommended to leave the default value $\sigma^T = 1$ or to set $\sigma^T = 0$ (when exchange is due to water flux only).
- $q^l_{d+1,d} = \boldsymbol{q}_{d+1} \cdot \boldsymbol{n}$ is the water flux from Ω_{d+1} to Ω_d.

The communication between dimensions is incorporated as the total flux boundary condition for the problem on Ω_{d+1}:

$$(\varrho_l c_l T \boldsymbol{q} - \delta \Lambda \nabla T) \cdot \boldsymbol{n} = q^T \tag{2.40}$$

and a source term in Ω_d:

$$F^T_{C3} = 0, \quad F^T_{C2} = q^T_{32}, \quad F^T_{C1} = q^T_{21}. \tag{2.41}$$

Energy balance. The heat equation satisfies the balance of energy in the following form:

$$e(0) + \int_0^t s(\tau)\, d\tau - \int_0^t f(\tau)\, d\tau = e(t)$$

for any instant t in the computational time interval. Here

$$e(t) := \sum_{d=1}^{3} \int_{\Omega^d} (\delta \tilde{s} T)(t, \boldsymbol{x})\, d\boldsymbol{x},$$

$$s(t) := \sum_{d=1}^{3} \int_{\Omega^d} F^T_S(t, \boldsymbol{x})\, d\boldsymbol{x},$$

$$f(t) := \sum_{d=1}^{3} \int_{\partial\Omega^d} (\varrho_l c_l T \boldsymbol{q} - \delta \Lambda \nabla T)(t, \boldsymbol{x}) \cdot \boldsymbol{n}\, d\boldsymbol{x}$$

is the energy [J], the volume source [Js^{-1}] and the energy flux [Js^{-1}] at time t, respectively. The energy, flux and source on every geometrical region is calculated at each output time step and the values together with the control sums are written to the file `energy_balance.{dat|txt}`. If, in addition, cumulative is set to true then the time-integrated flux and source is written.

Chapter 3

Numerical Methods

3.1 Diagonalized Mixed-Hybrid Method

Model of flow described in section 2.3 is solved by the mixed-hybrid formulation (MH) of the finite element method. As in the previous chapter, let τ be the time step and $\mathcal{T}_d$ a regular simplicial partition of Ω_d, $d = 1, 2, 3$. Denote by $\boldsymbol{W}_d(T_d) \subset \boldsymbol{H}(div, T_d)$ the space of Raviart-Thomas functions of order zero (RT_0) on an element $T_d \in \mathcal{T}_d$. We introduce the following spaces:

$$\boldsymbol{W} = \boldsymbol{W}_1 \times \boldsymbol{W}_2 \times \boldsymbol{W}_3, \quad \boldsymbol{W}_d = \prod_{T_d \in \mathcal{T}_d} \boldsymbol{W}_d(T_d),$$

$$Q = Q_1 \times Q_2 \times Q_3, \quad Q_d = L^2(\Omega_d). \tag{3.1}$$

For every $T_d \in \mathcal{T}_d$ we define the auxiliary space of values on interior sides of T_d:

$$\mathring{Q}(T_d) = \left\{ \mathring{q} \in L^2(\partial T_d \setminus \partial\Omega_d^D) : \mathring{q} = \boldsymbol{w} \cdot \boldsymbol{n}|_{\partial T_d}, \boldsymbol{w} \in \boldsymbol{W}_d \right\}. \tag{3.2}$$

Further we introduce the space of functions defined on interior sides that do not coincide with elements of the lower dimension:

$$\mathring{Q}_d = \Big\{ \mathring{q} \in \prod_{T \in \mathcal{T}_d} \mathring{Q}(T);\ \mathring{q}|_{\partial T} = \mathring{q}|_{\partial \tilde{T}} \quad \text{on the side } F = \partial T \cap \partial \tilde{T} \quad \text{if } F \cap \Omega_{d-1} = \emptyset \Big\}. \tag{3.3}$$

Finally we set $\mathring{Q} = \mathring{Q}_1 \times \mathring{Q}_2 \times \mathring{Q}_3$.

The *mixed-hybrid method* for the unsteady Darcy flow reads as follows. We are looking for a trio $(\mathbf{u}, h, \mathring{h}) \in \boldsymbol{W} \times Q \times \mathring{Q}$ which satisfies the saddle-point problem:

$$a(\boldsymbol{u}, \boldsymbol{v}) + b(\boldsymbol{v}, p) + \mathring{b}(\boldsymbol{v}, \mathring{p}) = \langle g, \boldsymbol{v} \rangle, \qquad \forall \boldsymbol{v} \in \boldsymbol{W}, \tag{3.4}$$

$$b(\boldsymbol{u}, q) + \mathring{b}(\boldsymbol{u}, \mathring{q}) - c(p, \mathring{p}, q, \mathring{q}) = \langle f, (q, \mathring{q}) \rangle, \qquad \forall q \in Q,\ \mathring{q} \in \mathring{Q}, \tag{3.5}$$

where

$$a(\boldsymbol{u},\boldsymbol{v}) = \sum_{d=1}^{3}\sum_{T\in\mathcal{T}_d}\int_T \frac{1}{\delta_d}\mathbb{K}_d^{-1}\boldsymbol{u}_d\cdot\boldsymbol{v}_d\,dx, \tag{3.6}$$

$$b(\boldsymbol{u},q) = -\sum_{d=1}^{3}\sum_{T\in\mathcal{T}_d}\int_T q_d\ \mathrm{div}\,\boldsymbol{u}_d\,dx, \tag{3.7}$$

$$\mathring{b}(\boldsymbol{u},\mathring{q}) = \sum_{d=1}^{3}\sum_{T\in\mathcal{T}_d}\int_{\partial T\setminus\partial\Omega_d}\mathring{q}|_{\partial T}(\boldsymbol{u}_d\cdot\boldsymbol{n})\,ds, \tag{3.8}$$

$$c(h,\mathring{h},q,\mathring{q}) = c_f(h,\mathring{h},q,\mathring{q}) + c_t(h,\mathring{h},q,\mathring{q}) + c_R(\mathring{h},\mathring{q}) \tag{3.9}$$

$$c_f(h,\mathring{h},q,\mathring{q}) = \sum_{d=2,3}\sum_{T\in\mathcal{T}_d}\int_{\partial T\cap\Omega_{d-1}}\sigma_d(p_{d-1}-\mathring{p}_d)(q_{d-1}-\mathring{q}_d)\,ds \tag{3.10}$$

$$c_t(h,\mathring{h},q,\mathring{q}) = \sum_{d=1}^{3}\sum_{T\in\mathcal{T}_d}\int_T \frac{\delta_d S_d}{\tau}h_d q_d\,dx, \tag{3.11}$$

$$c_R(\mathring{h},\mathring{q}) = \int_{\partial T\setminus\partial\Omega_d}\sigma_d^R\,h_d\mathring{q}_d\,ds, \tag{3.12}$$

$$\langle g,\boldsymbol{v}\rangle = -\sum_{d=1}^{3}\sum_{T\in\mathcal{T}_d}\int_{\partial T\cap\partial\Omega_N} p_d^D\,(\boldsymbol{v}\cdot\boldsymbol{n})\,ds, \tag{3.13}$$

$$\langle f,q\rangle = -\sum_{d=1}^{3}\int_{\Omega_d}\delta_d\,f_d\,q_d\,dx, \tag{3.14}$$

$$+\sum_{d=1}^{3}\sum_{T\in\mathcal{T}_d}\int_{\partial T\cap\partial\Omega_N} q_d^N\mathring{q}_d - \sigma_d^R\,h_d^R\mathring{q}_d\,ds \tag{3.15}$$

$$-\,c_t(\tilde{h},\mathring{\tilde{h}},q,\mathring{q}). \tag{3.16}$$

All quantities are meant in time t, only $\tilde{h}$ is the pressure head in time $t-\tau$.

The advantage of the mixed-hybrid method is that the set of equations (3.4) – (3.5) can be reduced by eliminating the unknowns $\boldsymbol{u}$ and q to a sparse positive definite system for $\mathring{q}$. This equation can then be efficiently solved using a preconditioned conjugate gradient method. Unfortunately, it appears that the resulting system does not satisfy the discrete maximum principle which in particular for short time steps τ can lead to unphysical oscillations. One possible solution is the diagonalization of the method (lumped mixed-hybrid method, LMH) proposed in [10]. This method was implemented in Flow123d as well. It consists in replacing the form c_t by

$$c_t(h,\mathring{h},q,\mathring{q}) = \sum_{d=1}^{3}\sum_{T\in\mathcal{T}_d}\sum_{i=1}^{d+1}\alpha_{T,i}|T|\frac{\delta_d S_d}{\tau}\left(\mathring{h}|_{S_{T,i}}\,\mathring{q}|_{S_{T,i}}\right),$$

and the source term $\sum_{d=1}^{3}\int_{\Omega_d}\delta_d\,f_d\,q_d\,dx$ by

$$\sum_{d=1}^{3}\sum_{T\in\mathcal{T}_d}\sum_{i=1}^{d+1}\alpha_{T,i}|T|\delta_d f_d\,\mathring{q}|_{S_{T,i}},$$

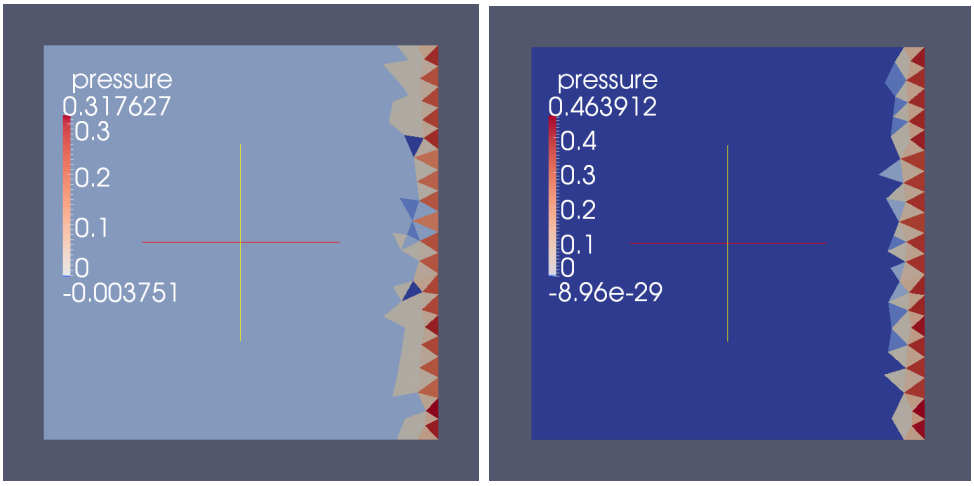


Figure 3.1: Comparison of MH (left) and LMH scheme (right), $\tau = 10^{-4}$.

where $|T|$ is the size of an element T, $S_{T,i}$ is the i-th side of T, and $\mathring{h}|_{S_{T,i}}$ is the degree of freedom on the side $S_{T,i}$. Weights $\alpha_{T,i}$ can be chosen to be $1/(d+1)$. After solving the set of equations it is necessary to modify the velocity field $\boldsymbol{u}$ by adding the time term. This modified system already satisfies the discrete maximum principle and does not produce oscillations. Figure 3.1 shows a comparison of the results using conventional MH scheme and LMH scheme. At the MH scheme one can observe oscillations in the wavefront where the minimum value is significantly less than zero.

3.2 Discontinuous Galerkin Method

Models for solute transport and heat transfer described in sections 2.4 and 2.6 are collectively formulated as a system of abstract advection-diffusion equations on domains Ω_d, $d = 1, 2, 3$, connected by communication terms. Consider for $d = 1, 2, 3$ the equation

$$\partial_t u_d + \operatorname{div}(\boldsymbol{b} u_d) - \operatorname{div}(\mathbb{A}\nabla u_d) = f^0 + f^1(u^S - u_d) + q(u_{d+1}, u_d) \text{ in } (0,T)\times\Omega_d \quad (3.17a)$$

with initial and boundary conditions

$$u_d(0,\cdot) = u^0 \qquad \text{in } \Omega_d, \quad (3.17b)$$

$$u_d = u^D \qquad \text{on } (0,T)\times\Gamma_d^D, \quad (3.17c)$$

$$(\boldsymbol{b}u_d - \mathbb{A}\nabla u_d)\cdot\boldsymbol{n} = f^N + \sigma^R(u_d - u^D) \qquad \text{on } (0,T)\times\Gamma_d^N, \quad (3.17d)$$

$$(\boldsymbol{b}u_d - \mathbb{A}\nabla u_d)\cdot\boldsymbol{n} = q(u_d, u_{d-1}) \qquad \text{on } (0,T)\times\Gamma_d^C, \quad (3.17e)$$

where

$$\Gamma_d^C := \overline{\Omega}_d \cap \overline{\Omega}_{d-1}.$$

The communication term $q(u_{d+1}, u_d)$ has the form

$$q(u_{d+1}, u_d) = \begin{cases} \alpha u_{d+1} + \beta u_d & \text{in } \Gamma_{d+1}^C,\ d = 1,2, \\ 0 & \text{on } \Omega_d \setminus \Gamma_{d+1}^C,\ d = 1,2, \text{ and for } d = 0,3. \end{cases} \quad (3.17f)$$

System (3.17) is spatially discretized by the discontinuous Galerkin method with weighted averages, which was derived for the case of one domain in [5] (for a posteriori estimate see [6]). For time discretization we use the implicit Euler method.

Let τ denote the time step. For a regular splitting $\mathcal{T}_d$ of Ω^d, $d = 1, 2, 3$, into simplices we define the following sets of element sides:

$\mathcal{E}_d$	sides of all elements in $\mathcal{T}_d$ (i.e. triangles for $d = 3$, lines for $d = 2$ and nodes for $d = 1$),
$\mathcal{E}_{d,I}$	interior sides (shared by 2 or more d-dimensional elements),
$\mathcal{E}_{d,B}$	outer sides (belonging to only one element),
$\mathcal{E}_{d,D}(t)$	sides, where the Dirichlet condition (3.17c) is given,
$\mathcal{E}_{d,N}(t)$	sides, where the Neumann or Robin condition (3.20c) is given,
$\mathcal{E}_{d,C}$	sides coinciding with Γ_d^C.

For an interior side E we denote by $\mathcal{N}_d(E)$ the set of elements that share E (notice that 1D and 0D sides can be shared by more than 2 elements). For an element $T \in \mathcal{N}_d(E)$ we denote $q_T := (\boldsymbol{b} \cdot \boldsymbol{n})_{|T}$ the outflow from T, and define $\mathcal{N}_d^-(E) := \{T \in \mathcal{N}_d(E) \,|\, q_T \le 0\}$, $\mathcal{N}_d^+(E) := \{T \in \mathcal{N}_d(E) \,|\, q_T > 0\}$ the sets of all outflow and inflow elements, respectively. For every pair $(T^+, T^-) \in \mathcal{N}_d^+(E) \times \mathcal{N}_d^-(E)$ we define the flux from T^+ to T^- as

$$q_{T^+ \to T^-} := \frac{q_{T^+} q_{T^-}}{\sum_{T \in \mathcal{N}_d^-(E)} q_T}.$$

We select arbitrary element $T_E \in \mathcal{N}_d(E)$ and define $\boldsymbol{n}_E$ as the the unit outward normal vector to ∂T_E at E. The jump in values of a function f between two adjacent elements $T_1, T_2 \in \mathcal{N}_d(E)$ is defined by $[f]_{T_1,T_2} = f_{|T_1|E} - f_{|T_2|E}$, similarly we introduce the average $\{f\}_{T_1,T_2} = \frac{1}{2}(f_{|T_1|E} + f_{|T_2|E})$ and a weighted average $\{f\}^{\omega}_{T_1,T_2} = \omega f_{|T_1|E} + (1-\omega) f_{|T_2|E}$. The weight ω is selected in a specific way (see [5]) taking into account the possible inhomogeneity of the tensor $\mathbb{A}$.

For every time step $t_k = k\tau$ we look for the discrete solution $u^k = (u_1^k, u_2^k, u_3^k) \in V$, where

$$V = \prod_{d=1}^{3} V_d \quad \text{and} \quad V_d = \{v : \overline{\Omega^d} \to \mathbb{R} \,|\, v_{|T} \in P_p(T)\ \forall T \in \mathcal{T}_d\}$$

are the spaces of piecewise polynomial functions of degree at most p on elements $\mathcal{T}_d$, generally discontinuous on interfaces of elements. The initial condition for u_d^0 is defined as the L^2-projection of u^0 to V_d. For $k = 1, 2, \ldots$, u^k is given as the solution of the problem

$$\frac{1}{\tau}\left(u^k - u^{k-1}, v\right)_V + a^k(u^k, v) = b^k(v) \quad \forall v \in V.$$

Here $(f, g)_V = \sum_{d=1}^{d} (f, g)_{\Omega^d}$, $(f, g)_{\Omega^d} = \int_{\Omega^d} fg$, and forms a^k, b^k are defined as follows:

$$\begin{aligned} &a^k((u_1, u_2, u_3), (v_1, v_2, v_3)) \\ &\quad = \sum_{d=1}^{3} \left(a_d^k(u_d, v_d) - (q(u_{d+1}, u_d), v_d)_{\Omega^d} - \sum_{E \in \mathcal{E}_{d,C}^d(t_k)} (q(u_d, u_{d-1}), v_d)_E \right), \end{aligned} \tag{3.18}$$

$$b^k((v_1, v_2, v_3)) = \sum_{d=1}^{3} b_d^k(v_d), \tag{3.19}$$

$$
\begin{aligned}
a_d^k(u,v) = {} & (\mathbb{A}\nabla u, \nabla v)_{\Omega^d} - (\boldsymbol{b}u, \nabla v)_{\Omega^d} + \left(f^1 u, v\right)_{\Omega^d} \\
& - \sum_{E\in\mathcal{E}_{d,I}^d} \sum_{\substack{T_1,T_2\in\mathcal{N}_d(E)\\ T_1\neq T_2}} \Bigg(\left(\{\mathbb{A}\nabla u\}_{T_1,T_2}^{\omega}\cdot\boldsymbol{n}_E, [v]_{T_1,T_2}\right)_E + \Theta\left(\{\mathbb{A}\nabla v\}_{T_1,T_2}^{\omega}\cdot\boldsymbol{n}_E, [u]_{T_1,T_2}\right)_E \\
& - \gamma_E\left([u]_{T_1,T_2}, [v]_{T_1,T_2}\right)_E \Bigg) - \sum_{E\in\mathcal{E}_{d,I}^d} \sum_{\substack{T^+\in\mathcal{N}_d^+(E)\\ T^-\in\mathcal{N}_d^-(E)}} \left(q_{T^+\to T^-}\{u\}_{T^+,T^-}, [v]_{T^+,T^-}\right)_E \\
& + \sum_{E\in\mathcal{E}_{d,D}^d(t_k)} \left(\gamma_E\left(u,v\right)_E + \left(\boldsymbol{b}\cdot\boldsymbol{n}u, v\right)_E - \left(\mathbb{A}\nabla u\cdot\boldsymbol{n}, v\right)_E - \Theta\left(\mathbb{A}\nabla v\cdot\boldsymbol{n}, u\right)_E\right) \\
& + \sum_{E\in\mathcal{E}_{d,N}^d(t_k)} \left(\sigma^R u, v\right)_E, \\
b_d^k(v) = {} & \left(f^0 + f^1 u^S, v\right)_{\Omega^d} + \sum_{E\in\mathcal{E}_{d,D}^d(t_k)} \left(\gamma_E\left(u^D, v\right)_E - \Theta\left(u^D, \mathbb{A}\nabla v\cdot\boldsymbol{n}\right)_E\right) \\
& + \sum_{E\in\mathcal{E}_{d,N}^d(t_k)} \left(\sigma^R u^D - f^N, v\right)_E.
\end{aligned}
$$

The Dirichlet condition is here enforced by a penalty with an arbitrary parameter $\gamma_E > 0$; its value influences the level of solution's discontinuity. For $\gamma_E \to +\infty$ we obtain asymptotically (at least formally) the finite element method. The constant Θ can take the values -1, 0 or 1, where -1 corresponds to the nonsymetric, 0 to the incomplete and 1 to the symetric variant of the discontinuous Galerkin method.

3.3 Finite Volume Method for Convective Transport

In the case of the purely convective solute transport ($\mathbb{D} = 0$), problem (3.17) is replaced by:

$$
\begin{aligned}
\partial_t u_d + \operatorname{div}(\boldsymbol{b}u_d) &= f^0 + f^1(u^S - u_d) + q(u_{d+1}, u_d) && \text{in } (0,T)\times\Omega_d, && \text{(3.20a)}\\
u_d(0,\cdot) &= u^0 && \text{in } \Omega_d, && \text{(3.20b)}\\
(\boldsymbol{b}\cdot\boldsymbol{n})u_d &= (\boldsymbol{b}\cdot\boldsymbol{n})u^D && \text{on } \Gamma_d^I, && \text{(3.20c)}
\end{aligned}
$$

where

$$
\Gamma_d^I := \{(t,\boldsymbol{x}) \in (0,T)\times\partial\Omega_d \,|\, \boldsymbol{b}(t,\boldsymbol{x})\cdot\boldsymbol{n}(\boldsymbol{x}) < 0\}.
$$

The communication term $q(u_{d+1}, u_d)$ has the same structure as in (3.17f).

The system is discretized by the cell-centered finite volume method combined with the explicit Euler time discretization. Using the notation of Section 3.2, we consider the space V of piecewise constants on elements and define the discrete problem:

$$
\frac{1}{\tau}\left(u^k - u^{k-1}, v\right)_V + a^{k-1}(u^{k-1}, v) = b^{k-1}(v) \quad \forall v \in V,
$$

where the forms a^k and b^k are defined in (3.18)-(3.19) and a_d^k, b_d^k now have simplified form:

$$a_d^k(u,v) = -\sum_{T_i\in\mathcal{T}_d}\left(((\boldsymbol{b}\cdot\boldsymbol{n})^+u,v)_{\partial T_i} + \sum_{T_j\in\mathcal{T}_d}(q_{T_j\to T_i}u,v)_{\partial T_i\cap\partial T_j}\right),$$
$$b_d^k(v) = (f^0 + f^1(u^S - u_d^{k-1})^+, v)_{\Omega^d} + \sum_{T_i\in\mathcal{T}_d}((\boldsymbol{b}\cdot\boldsymbol{n})^- u^D, v)_{\partial T_i\cap\partial\Omega_d}.$$

The above formulation corresponds to the upwind scheme, ideal mixing in case of multiple elements sharing one side, and explicit treatment of linear source term.

3.4 Solution Issues for Reaction Term

3.4.1 Dual Porosity

The analytic solution of the system of differential equations (2.25) at the time t with initial conditions $c_m(0)$ and $c_i(0)$ is

$$c_m(t) = (c_m(0) - c_a(0))\exp\left(-D_{dp}\left(\frac{1}{\vartheta_m}+\frac{1}{\vartheta_i}\right)t\right) + c_a(0), \tag{3.21}$$

$$c_i(t) = (c_i(0) - c_a(0))\exp\left(-D_{dp}\left(\frac{1}{\vartheta_m}+\frac{1}{\vartheta_i}\right)t\right) + c_a(0), \tag{3.22}$$

where c_a is the weighted average

$$c_a = \frac{\vartheta_m c_m + \vartheta_i c_i}{\vartheta_m + \vartheta_i}.$$

If the time step is large, we use the analytic solution to compute new values of concentrations. Otherwise, we replace the time derivatives in (2.25a) and (2.25b) by first order forward differences and we get the classical Euler scheme

$$c_m(t^+) = \frac{D_{dp}\Delta t}{\vartheta_m}(c_i(t) - c_m(t)) + c_m(t), \tag{3.23}$$

$$c_i(t^+) = \frac{D_{dp}\Delta t}{\vartheta_i}(c_m(t) - c_i(t)) + c_i(t), \tag{3.24}$$

$$\tag{3.25}$$

where $\Delta t = t^+ - t$ is the time step.

The condition on the size of the time step is derived from the Taylor expansion of (3.21) or (3.22), respectively. We neglect the higher order terms and we want the second order term to be smaller than the given scheme tolerance tol, relatively to c_a,

$$(c_m(0) - c_a(0))\frac{D_{dp}^2(\Delta t)^2\left(\frac{\vartheta_m+\vartheta_i}{\vartheta_m\vartheta_i}\right)^2}{2}\frac{1}{c_a} \le tol. \tag{3.26}$$

We then transform the above inequation into the following condition which is tested in the program

$$\max(|c_m(0) - c_a(0)|, |c_i(0) - c_a(0)|) \le 2c_a\left(\frac{\vartheta_m\vartheta_i}{D_{dp}\Delta t(\vartheta_m+\vartheta_i)}\right)^2 tol. \tag{3.27}$$

If the inequation (3.27) is not satisfied, then the analytic solution is used.

3.4.2 Equilibrial Sorption

Let us now describe the actual computation of the sorption model. To solve (2.28) iteratively, it is very important to define the interval where to look for the solution (unknown c_l), see Figure 3.2. The lower bound is 0 (concentration can not reach negative values). The upper bound is derived using a simple mapping. Let us suppose limited solubility of the selected transported substance and let us denote the limit $\bar{c}_l$. We keep the maximal "total mass" $\bar{c}_T = \mu_l \cdot \bar{c}_l + \mu_s \cdot f(\bar{c}_l)$, but we dissolve all the mass to get maximal $c_l^{max} > \bar{c}_l$. That means $c_s = 0$ at this moment. We can slightly enlarge the interval by setting the upper bound equal to $c_l^{max} + const_{small}$.

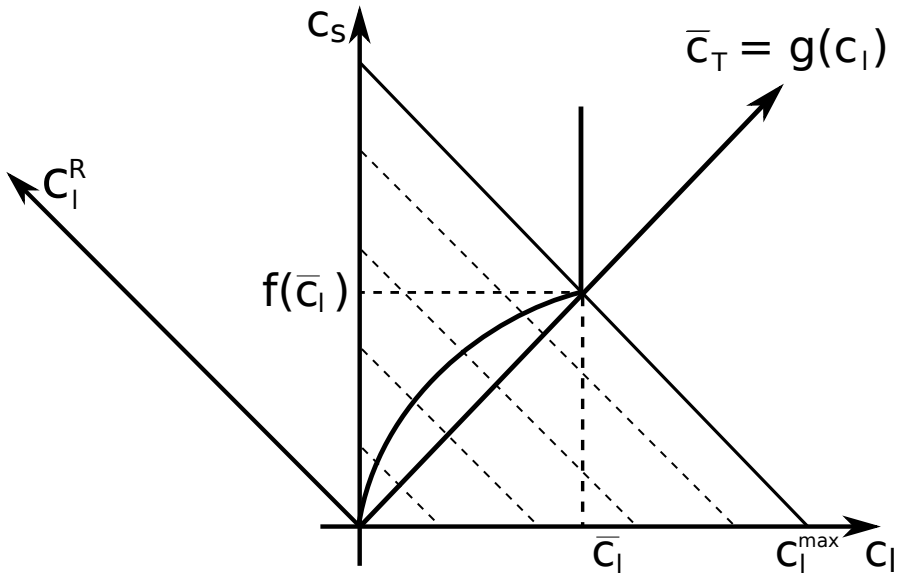


Figure 3.2: Sorption in combination with limited solubility.

To approximate the equation (2.28) using interpolation, we need to prepare the set of values which represents $[c_l, f(c_l)]$, with c_l equidistantly distributed in transformed (rotated and rescaled) coordination system at first. The construction process of the interpolation table follows.

1. Maximal "total mass" $\bar{c}_T = \mu_l \cdot \bar{c}_l + \mu_s \cdot f(\bar{c}_l)$ is computed.
2. Total mass step is derived $mass\_step = \bar{c}_T / n\_steps$. n_steps is the number of substeps.
3. Appropriate $c_T^j = (mass\_step \cdot j)/\mu_l,\ j \in \{0, \ldots, n\_steps\}$ are computed.
4. The equations $\mu_l \cdot c_T^j = \mu_l \cdot c_l^j + \mu_s \cdot f(c_l^j)\ j \in \{0, \ldots, n\_steps\}$ are solved for c_l^j as unknowns. The solution is the set of ordered couples (points) $[c_l^j, f(c_l^j)],\ j \in \{0, \ldots, n\_steps\}$.

After the computation of $\{[c_l^j, f(c_l^j)]\}$, we transform these coordinates to the system where the total mass is an independent variable. This is done by multiplication of

precomputed points using the transformation matrix $\mathbf{A}$:

$$\begin{aligned} \vec{c}^{\,R} &= \mathbf{A} \cdot \vec{c} \\ \begin{bmatrix} c_l^{R,j} \\ c_s^{R,j} \end{bmatrix} &= \begin{bmatrix} \vartheta \cdot \rho_w & M_s(1-\vartheta)\rho_R \\ -M_s(1-\vartheta)\rho_R & \vartheta \cdot \rho_w \end{bmatrix} \cdot \begin{bmatrix} c_l^j \\ c_s^j \end{bmatrix} \\ j &\in \{0, \ldots, n\_steps\} \end{aligned} \tag{3.28}$$

The values $c_l^{R,j}$ are equidistantly distributed and there is no reason to save them, but the values $c_s^{R,j}$ are stored in onedimensional interpolation table.

Once we have the interpolation table, we can use it for projecting the transport results $[c_l, c_s]$ on the isotherm under consideration. Following steps must be taken.

1. Achieved concentrations are transformed to the coordinate system through multiplication with the matrix $\mathbf{A}$, see (3.28).
2. Transformed values are interpolated.
3. The result of interpolation is transformed back. The backward transformation consists of multiplication with $\mathbf{A}^T$ which is followed by rescaling the result. Rescaling the result is necessary because $\mathbf{A}$ is not orthonormal as it is shown bellow.

$$\mathbf{A}^T \cdot \mathbf{A} = ((\vartheta - 1)^2 \cdot M_s^2 \cdot \rho_R^2 + \vartheta^2 \cdot \rho_w^2) \cdot \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

Limited solubility. When $\mu_l \cdot c_l + \mu_s \cdot f(c_l) > \mu_l \cdot \bar{c}_l + \mu_s \cdot f(\bar{c}_l)$, neither iterative solver nor interpolation table is used. The aqueous concentration is set to be $\bar{c}_l$ and sorbed concentration is computed $c_s = (\mu_l \cdot c_l + \mu_s \cdot f(c_l) - \mu_l \cdot \bar{c}_l)/\mu_s$.

3.4.3 System of Linear Ordinary Differential Equations

A system of linear ordinary differential equations (ODE) appears in several places in the model. We provide several solvers which we shall brielfy describe in this section. Let us denote the ODE system

$$\partial_t \boldsymbol{c}(t) = \mathbf{A}(t)\boldsymbol{c}(t) + \boldsymbol{b}(t).$$

Semi-analytic solution. A semi-analytic solution can be obtained in special cases due to the physical nature of the problem. The problem can be then solved only by a matrix multiplication $\boldsymbol{c}(t + \Delta t) = \mathbf{R}\boldsymbol{c}(t)$. This is used in case of radioactive decays and first order kinetic reactions.

The right hand side $\boldsymbol{b}$ is zero and $\mathbf{A}$ is constant. The assumption is made that the equations are independent during one time step. Each quantity c_i (concentration in this case) is decreased by $e^{a_{ii}\Delta t}$ (supposing negative diagonal) during time step Δt. The decrement $\left(1 - e^{a_{ii}\Delta t}\right)$ is then distributed among other quantities according to the given fraction.

In case of radioactive decays and first order reactions, the elements of the solution matrix $\mathbf{R}$ are

$$\begin{aligned} r_{ii} &= e^{-k_i \Delta t}, \\ r_{ji} &= \left(1 - e^{-k_i \Delta t}\right) b_{ji} \frac{M_j}{M_i}, \end{aligned}$$

where b_{ji} is the branching ratio of i-th reactant (or radionuclide) and $\frac{M_j}{M_i}$ is the fraction of molar masses. The expressions $b_{ji}\frac{M_j}{M_i}$ are then obtained from the system matrix by dividing $-\frac{a_{ji}}{a_{ii}}$. See the system matrix entries in (2.38).

The assumption (equations independence) is adequate when a very small time step is applied. This will then lead to huge amount of evaluations of the exponential functions which can be expensive, so other numerical methods might be more appropriate. When the time step is large then the assumption is inadequate.

On the other hand, if the time step is constant (for significantly large number of time steps), we get the solution cheaply just by matrix multiplication, because the matrix $\mathbf{R}$ is constant.

Padé approximant. For homogenous systems with constant matrix $\mathbf{A}$, we can use Padé approximation to find the solution. This method finds a rational function whose power series agrees with a power series expansion of a given function to the highest possible order (e.g. in [9]). Let

$$f(t) = \sum_{j=0}^{\infty} c_j t^j = \sum_{j=0}^{\infty} \frac{1}{n!} f^{(j)}(t_0)$$

be the function being approximated and its power series given by Taylor expansion about t_0. Then the rational function

$$R_{mn}(t) = \frac{P_m(t)}{Q_n(t)} = \frac{\sum_{j=0}^{m} p_j t^j}{\sum_{j=0}^{n} q_j t^j}, \tag{3.29}$$

which satisfies

$$f(t) \approx \sum_{j=0}^{m+n} c_j t^j = R_{mn}(t), \tag{3.30}$$

is called Padé approximant. From (3.30), we obtain $m+n+2$ equations for coefficients of the nominator P_m (polynomial of degree m) and the denominator Q_n (polynomial of degree n). We also see that the error of the approximation is $O(t^{m+n+1})$. By convention, the denominator is normalized such that $q_0 = 1$.

Now, we consider the solution of our ODE system in a form $\boldsymbol{c}(t) = e^{\mathbf{A}t}\boldsymbol{c}(0)$. We shall approximate the matrix exponential function using a matrix form of (3.29). For exponential functions, there are known coeffficients of the nominator and denominator:

$$\mathbf{P}_m(\mathbf{A}t) = \sum_{j=0}^{m} \frac{(m+n-j)!m!}{(m+n)!j!(m-j)!} (\mathbf{A}t)^j, \tag{3.31}$$

$$\mathbf{Q}_n(\mathbf{A}t) = \sum_{j=0}^{n} (-1)^j \frac{(m+n-j)!n!}{(m+n)!j!(n-j)!} (\mathbf{A}t)^j. \tag{3.32}$$

Finally, we can write the solution at time $t + \Delta t$

$$\boldsymbol{c}(t + \Delta t) = \frac{\mathbf{P}_m(\mathbf{A}\Delta t)}{\mathbf{Q}_n(\mathbf{A}\Delta t)}\boldsymbol{c}(t) = \mathbf{R}_{mn}(\mathbf{A}\Delta t)\boldsymbol{c}(t). \tag{3.33}$$

If the time step Δt is constant, we do not need to compute the matrix $\mathbf{R}_{mn}$ repeatedly and we get the solution cheaply just by matrix multiplication. In the oposite case, we avoid evaluating the exponential function and still get the solution quite fast (comparing to computing semi-analytic solution).

Chapter 4

File Formats

4.1 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 4.1. 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.

4.1.1 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.

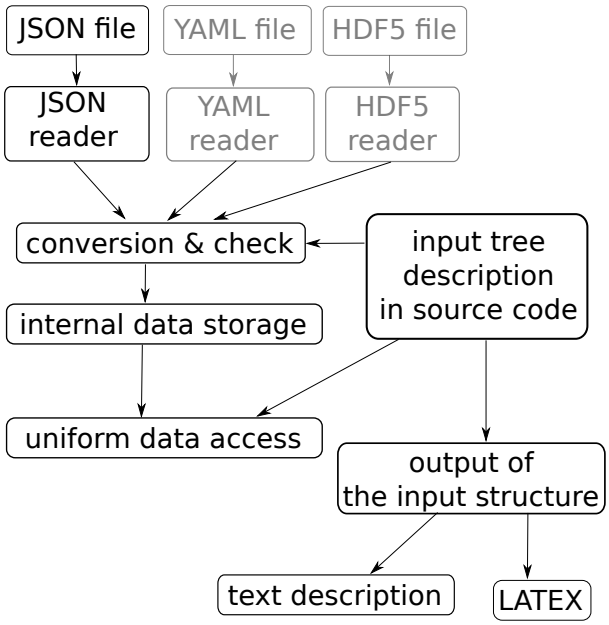


Figure 4.1: Sturucture of the input subsystem. Grey boxes are not implemented yet.

- 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/`.

4.1.2 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 5. 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.

4.1.3 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"
}
```

4.1.4 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.

4.2 Important Record Types of Flow123d Input

4.2.1 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".

4.2.2 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"]
}
```

4.2.3 Field Data for Equations

Every equation record has key `input_fields`, intended to set both the bulk and boundary fields. These keys contain an array of region-time initialization records like the `Data` record of the DarcyFlow equation. Every such record specifies 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 overwrite the previous ones. Times should form a non-decreasing sequence.

Example:

```
input_fields = [
    { // 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"
        }
    }
]
```

4.3 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. Elements of one region should have common dimension. Flow123d interprets regions with labels starting with period as the boundary elements that are not used for calculations.

`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. Integer code `<element shape>` represents the shape of element, 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. This header is folowed by field data on individual elements. Flow123d assumes that elements are sorted by `element ID`, but doesn't need to form a continuos sequence.

4.4 Output Files

Flow123d supports output of scalar, vector and tensor 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 software packages. However, our primal goal is to support data transfer into the Paraview visualization software. See key `format`.

Input record of every equation (flow, transport, reactions, heat) contains the keys `output_stream` and `output_fields`. In `output_stream`, the name and type of the output file is specified. Further, in `output_fields`, one determines the list of fields intended for output. The available output fields include input data as well as the simulation results.

Below we mention the most important output fields of all equations and link to the complete lists.

Darcy flow	
`pressure_p0`	Pressure head [m], piecewise constant on every element. This field is directly produced by the MH method and thus contains no postprocessing error.
`pressure_p1`	Same pressure head field, but interpolated into $P1$ continuous scalar field. Namely you lost dicontinuities on fractures.
`velocity_p0`	Vector field of water flux [m^3s^{-1}]. For every element we evaluate discrete flux field in barycenter.
`piezo_head_p0`	Piezometric head [m], 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`).
complete list	See Darcy flow output fields.
Convection transport	
`conc`	Concentration [kgm^{-3}], piecewise constant on every element.
complete list	See Convection transport output fields.
Transport with dispersion	
`conc`	Concentration [kgm^{-3}], piecewise linear on every element. Even if higher order polynomial approximation is used in simulation, the results are saved only in element corners.
complete list	See Transport with dispersion output fields.
Dual porosity	
`conc_immobile`	Concentration [kgm^{-3}] in immobile zone, piecewise linear on every element.
complete list	See Dual porosity output fields.
Sorption, Mobile sorption, Immobile sorption	
`conc_solid`	Concentration [$mol\,kg^{-1}$] of sorbed substance, piecewise linear on every element.
complete list	See Sorption output fields, Mobile sorption output fields, Immobile sorption output fields.
Heat transfer	
`temperature`	Temperature [K], piecewise linear on every element. Even if higher order polynomial approximation is used in simulation, the results are saved only in element corners.
complete list	See Heat transfer output fields.

4.4.1 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.

Balance of Conservative Quantities

Primary and secondary equations can produce additional information on fluxes, sources and state of conservative quantities (for flow it is the volume of water, for transport the mass of a substance, for heat transfer the energy). The computation of balance is governed by the key `balance`. The balance file (default `water_balance.txt`, `mass_balance.txt`, `energy_balance.txt`) contains the following information:

- time and region
- name and unit of the quantity
- mass (current state), flux through boundary and volume source at given time and region
- incoming and outgoing flux and source
- flux and source increment since the last balance output time
- cumulative flux and source
- error: current mass should equal to initial mass + cumulative sources - cumulative fluxes

Raw Water Flow Data File

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

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

where

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

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

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

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

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

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

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

Chapter 5

Main Input File Reference

---

record: **Root**

Root record of JSON input for Flow123d.

`problem` = ⟨*abstract type: Problem* ⟩

Default: ⟨*obligatory* ⟩

Simulation problem to be solved.

`pause_after_run` = ⟨*Bool* ⟩

Default: false

If true, the program will wait for key press before it terminates.

---

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)

`partitioning` = ⟨*record:* *Partition* ⟩

Default: any_neighboring

Parameters of mesh partitioning algorithms.

---

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: **Partition** constructible from key: graph_type

Setting for various types of mesh partitioning.

`tool` = ⟨*selection: PartTool* ⟩

Default: METIS

Software package used for partitioning. See corresponding selection.

`graph_type` = ⟨*selection: GraphType* ⟩

Default: any_neighboring

Algorithm for generating graph and its weights from a multidimensional mesh.

---

selection type: **PartTool**

Select the partitioning tool to use.

Possible values:

`PETSc` : Use PETSc interface to various partitioning tools.

`METIS` : Use direct interface to Metis.

---

selection type: **GraphType**

Different algorithms to make the sparse graph with weighted edges from the multidimensional mesh. Main difference is dealing with neighborings of elements of different dimension.

Possible values:

`any_neighboring` : Add edge for any pair of neighboring elements.

`any_wight_lower_dim_cuts` : Same as before and assign higher weight to cuts of lower dimension in order to make them stick to one face.

`same_dimension_neghboring` : Add edge for any pair of neighboring elements of same dimension (bad for matrix multiply).

---

record: **TimeGovernor** constructible from key: max_dt

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

`start_time` = ⟨*Double* ⟩

Default: 0.0

Start time of the simulation.

`end_time` = ⟨*Double* ⟩

Default: "Infinite end time."

End time of the simulation.

`init_dt` = ⟨*Double [0,]* ⟩

Default: 0.0

Initial guess for the time step.
Only useful for equations that use adaptive time stepping.If set to 0.0, the time step is determined in fully autonomous way if the equation supports it.

`min_dt` = ⟨*Double [0,]* ⟩

Default: "Machine precision."

Soft lower limit for the time step. Equation using adaptive time stepping can notsuggest smaller time step, but actual time step could be smaller in order to match prescribed input or output times.

`max_dt` = ⟨*Double [0,]* ⟩

Default: "Whole time of the simulation if specified, infinity else."

Hard upper limit for the time step. Actual length of the time step is also limitedby input and output times.

---

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: LinSys* ⟩

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.

`balance` = ⟨*record: Balance* ⟩

Default: ⟨*obligatory* ⟩

Settings for computing mass balance.

`input_fields` = ⟨*Array of record: DarcyFlowMH_Data* ⟩

Default: ⟨*obligatory* ⟩

---

abstract type: **LinSys**

Descendants:

Linear solver setting.

Petsc

Bddc

---

record: **Petsc** implements abstract type: LinSys

Solver setting.

`TYPE` = ⟨*selection: LinSys_TYPE_selection* ⟩

Default: Petsc

Sub-record selection.

`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.

`a_tol` = ⟨*Double [0,]* ⟩

Default: 1.0e-9

Absolute residual tolerance.

`options` = ⟨*String (generic)* ⟩

Default:

Options passed to PETSC before creating KSP instead of default setting.

---

record: **Bddc** implements abstract type: LinSys

Solver setting.

`TYPE` = ⟨*selection: LinSys_TYPE_selection* ⟩

Default: Bddc

Sub-record selection.

`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.

`max_nondecr_it` = ⟨*Integer [0,]* ⟩

Default: 30

Maximum number of iterations of the linear solver with non-decreasing residual.

`number_of_levels` = ⟨*Integer [0,]* ⟩

Default: 2

Number of levels in the multilevel method (=2 for the standard BDDC).

`use_adaptive_bddc` = ⟨*Bool* ⟩

Default: false

Use adaptive selection of constraints in BDDCML.

`bddcml_verbosity_level` = ⟨*Integer [0, 2]* ⟩

Default: 0

Level of verbosity of the BDDCML library: 0 - no output, 1 - mild output, 2 - detailed output.

---

record: **DarcyMHOutput**

Parameters of MH output.

`output_stream` = ⟨*record: OutputStream* ⟩

Default: ⟨*obligatory* ⟩

Parameters of output stream.

`output_fields` = ⟨*Array of selection: DarcyMHOutput_Selection* ⟩

Default: ⟨*obligatory* ⟩

List of fields to write to output file.

`compute_errors` = ⟨*Bool* ⟩

Default: false

SPECIAL PURPOSE. Computing errors pro non-compatible coupling.

`raw_flow_output` = ⟨*output file name* ⟩

Default: ⟨*optional* ⟩

Output file with raw data form MH module.

---

record: **OutputStream**

Parameters of output.

`file` = ⟨*output file name* ⟩

Default: ⟨*obligatory* ⟩

File path to the connected output file.

`format` = ⟨*abstract type: OutputTime* ⟩

Default: ⟨*optional* ⟩

Format of output stream and possible parameters.

`time_step` = ⟨*Double [0,]* ⟩

Default: ⟨*optional* ⟩

Time interval between outputs.
Regular grid of output time points starts at the initial time of the equation and ends at the end time which must be specified.
The start time and the end time are always added.

`time_list` = ⟨*Array of Double [0,]* ⟩

Default: "List containing the initial time of the equation. You can prescribe an empty list to override this behavior."

Explicit array of output time points (can be combined with 'time_step'.

`add_input_times` = ⟨*Bool* ⟩

Default: false

Add all input time points of the equation, mentioned in the 'input_fields' list, also as the output points.

---

abstract type: **OutputTime**

Descendants:

Format of output stream and possible parameters.

vtk

gmsh

---

record: **vtk** implements abstract type: OutputTime

Parameters of vtk output format.

`TYPE` = ⟨*selection: OutputTime_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: OutputTime

Parameters of gmsh output format.

`TYPE` = ⟨*selection: OutputTime_TYPE_selection* ⟩

Default: gmsh

Sub-record selection.

---

selection type: **DarcyMHOutput_Selection**

Selection of fields available for output.

Possible values:

`anisotropy` : Output of the field anisotropy [−] (Anisotropy of the conductivity tensor.).

`cross_section` : Output of the field cross_section $[m^{3-d}]$ (Complement dimension parameter (cross section for 1D, thickness for 2D).).

`conductivity` : Output of the field conductivity $[ms^{-1}]$ (Isotropic conductivity scalar.).

`sigma` : Output of the field sigma [−] (Transition coefficient between dimensions.).

`water_source_density` : Output of the field water_source_density $[s^{-1}]$ (Water source density.).

`init_pressure` : Output of the field init_pressure $[m]$ (Initial condition as pressure).

`storativity` : Output of the field storativity $[m^{-1}]$ (Storativity.).

`pressure_p0` : Output of the field pressure_p0 $[m]$.

`pressure_p1` : Output of the field pressure_p1 $[m]$.

`piezo_head_p0` : Output of the field piezo_head_p0 $[m]$.

`velocity_p0` : Output of the field velocity_p0 $[ms^{-1}]$.

`subdomain` : Output of the field subdomain [−].

`region_id` : Output of the field region_id [−].

`pressure_diff` : Output of the field pressure_diff [m].

`velocity_diff` : Output of the field velocity_diff [ms^{-1}].

`div_diff` : Output of the field div_diff [s^{-1}].

---

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: **Balance** constructible from key: balance_on

Balance of a conservative quantity, boundary fluxes and sources.

`balance_on` = ⟨*Bool* ⟩

Default: true

Balance is computed if the value is true.

`format` = ⟨*selection: Balance_output_format* ⟩

Default: txt

Format of output file.

`cumulative` = ⟨*Bool* ⟩

Default: false

Compute cumulative balance over time. If true, then balance is calculated at each computational time step, which can slow down the program.

`file` = ⟨*output file name* ⟩

Default: "FileName balance.*"

File name for output of balance.

---

selection type: **Balance_output_format**

Format of output file for balance.

Possible values:

`legacy` : Legacy format used by previous program versions.

`txt` : Excel format with tab delimiter.

`gnuplot` : Format compatible with GnuPlot datafile with fixed column width.

Record to set fields of the equation. 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_Data 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 have 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). $[m^{3-d}]$

`conductivity` = ⟨*abstract type: Field:R3 → Real* ⟩

Default: ⟨*optional* ⟩

Isotropic conductivity scalar. $[ms^{-1}]$

`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. $[s^{-1}]$

`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. $[m]$

`bc_flux` = ⟨*abstract type: Field:R3 → Real* ⟩

Default: ⟨*optional* ⟩

Flux in Neumman or Robin boundary condition. $[m^{4-d}s^{-1}]$

`bc_robin_sigma` = ⟨*abstract type: Field:R3 → Real* ⟩

Default: ⟨*optional* ⟩

Conductivity coefficient in Robin boundary condition. $[m^{3-d}s^{-1}]$

`init_pressure` = ⟨*abstract type: Field:R3 → Real* ⟩

Default: ⟨*optional* ⟩

Initial condition as pressure $[m]$

`storativity` = ⟨*abstract type: Field:R3 → Real* ⟩

Default: ⟨*optional* ⟩

Storativity. $[m^{-1}]$

`bc_piezo_head` = ⟨*abstract type: Field:R3 → Real* ⟩

Default: ⟨*optional* ⟩

Boundary condition for pressure as piezometric head.

`init_piezo_head` = ⟨*abstract type: Field:R3 → Real* ⟩

Default: ⟨*optional* ⟩

Initial condition for pressure as piezometric head.

`flow_old_bcd_file` = ⟨*input file name* ⟩

Default: ⟨*optional* ⟩

File with mesh dependent boundary conditions (obsolete).

---

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]

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

`TYPE` = ⟨*selection: Field:R3 → Real[3,3]_TYPE_selection* ⟩

Default: FieldInterpolatedP0

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

---

R3 → Real Field constant in space.

`TYPE` = ⟨*selection: Field:R3 → Real_TYPE_selection* ⟩

Default: FieldInterpolatedP0

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 → Enum** default descendant: FieldConstant

---

Descendants:

Abstract record for all time-space functions.

`FieldConstant`

`FieldFormula`

`FieldPython`

`FieldInterpolatedP0`

`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: DarcyFlow_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: **DarcyFlow_BC_Type**

Possible values:

`none` : Homogeneous Neumann boundary condition. Zero flux

`dirichlet` : Dirichlet boundary condition. Specify the pressure head through the 'bc_pressure' field or the piezometric head through the 'bc_piezo_head' field.

`neumann` : Neumann boundary condition. Prescribe water outflow by the 'bc_flux' field.

`robin` : Robin boundary condition. Water outflow equal to $sigma(h - h^R)$. Specify the transition coefficient by 'bc_sigma' and the reference pressure head or pieaozmetric head through 'bc_pressure' and 'bc_piezo_head' respectively.

---

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: **FieldInterpolatedP0** implements abstract type: Field:R3 → Enum

---

R3 → Enum Field constant in space.

`TYPE` = ⟨*selection: Field:R3 → Enum_TYPE_selection* ⟩

Default: FieldInterpolatedP0

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: **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.

---

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: LinSys* ⟩

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.

`balance` = ⟨*record: Balance* ⟩

Default: ⟨*obligatory* ⟩

Settings for computing mass balance.

`input_fields` = ⟨*Array of record: DarcyFlowMH_Data* ⟩

Default: ⟨*obligatory* ⟩

`time` = ⟨*record: TimeGovernor* ⟩

Default: ⟨*obligatory* ⟩

Time governor setting for the unsteady Darcy flow model.

---

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: LinSys* ⟩

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.

`balance` = ⟨*record:* *Balance* ⟩

Default: ⟨*obligatory* ⟩

Settings for computing mass balance.

`input_fields` = ⟨*Array of record:* *DarcyFlowMH_Data* ⟩

Default: ⟨*obligatory* ⟩

`time` = ⟨*record:* *TimeGovernor* ⟩

Default: ⟨*obligatory* ⟩

Time governor setting for the unsteady Darcy flow model.

---

abstract type: **Transport**

Descendants:

Secondary equation for transport of substances.

`TransportOperatorSplitting`

`SoluteTransport_DG`

`HeatTransfer_DG`

---

record: **TransportOperatorSplitting** implements abstract type: Transport

---

Explicit FVM transport (no diffusion) coupled with reaction and adsorption 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 secondary equation.

`balance` = ⟨*record:* *Balance* ⟩

Default: ⟨*obligatory* ⟩

Settings for computing balance.

`output_stream` = ⟨*record:* *OutputStream* ⟩

Default: ⟨*obligatory* ⟩

Parameters of output stream.

`substances` = ⟨*Array of record:* *Substance* ⟩

Default: ⟨*obligatory* ⟩

Specification of transported substances.

`reaction_term` = ⟨*abstract type: ReactionTerm* ⟩

Default: ⟨*optional* ⟩

Reaction model involved in transport.

`input_fields` = ⟨*Array of record: TransportOperatorSplitting_Data* ⟩

Default: ⟨*obligatory* ⟩

`output_fields` = ⟨*Array of selection: ConvectionTransport_Output* ⟩

Default: conc

List of fields to write to output file.

---

record: **Substance** constructible from key: name

Chemical substance.

`name` = ⟨*String (generic)* ⟩

Default: ⟨*obligatory* ⟩

Name of the substance.

`molar_mass` = ⟨*Double [0,]* ⟩

Default: 1

Molar mass of the substance [kg/mol].

---

abstract type: **ReactionTerm**

Descendants:

Equation for reading information about simple chemical reactions.

`FirstOrderReaction`

`RadioactiveDecay`

`Sorption`

`SorptionMobile`

`SorptionImmobile`

`DualPorosity`

`Semchem`

---

record: **FirstOrderReaction** implements abstract type: ReactionTerm

---

A model of first order chemical reactions (decompositions of a reactant into products).

`TYPE` = ⟨*selection: ReactionTerm_TYPE_selection* ⟩

Default: FirstOrderReaction

Sub-record selection.

`reactions` = ⟨*Array of record: Reaction* ⟩

Default: ⟨*obligatory* ⟩

An array of first order chemical reactions.

`ode_solver` = ⟨*abstract type: LinearODESolver* ⟩

Default: ⟨*optional* ⟩

Numerical solver for the system of first order ordinary differential equations coming from the model.

---

record: **Reaction**

Describes a single first order chemical reaction.

`reactants` = ⟨*Array [1,] of record: FirstOrderReactionReactant* ⟩

Default: ⟨*obligatory* ⟩

An array of reactants. Do not use array, reactions with only one reactant (decays) are implemented at the moment!

`reaction_rate` = ⟨*Double [0,]* ⟩

Default: ⟨*obligatory* ⟩

The reaction rate coefficient of the first order reaction.

`products` = ⟨*Array [1,] of record: FirstOrderReactionProduct* ⟩

Default: ⟨*obligatory* ⟩

An array of products.

---

record: **FirstOrderReactionReactant** constructible from key: name

---

A record describing a reactant of a reaction.

`name` = ⟨*String (generic)* ⟩

Default: ⟨*obligatory* ⟩

The name of the reactant.

---

record: **FirstOrderReactionProduct** constructible from key: name

---

A record describing a product of a reaction.

`name` = ⟨*String (generic)* ⟩

Default: ⟨*obligatory* ⟩

The name of the product.

`branching_ratio` = ⟨*Double [0,]* ⟩

Default: 1.0

The branching ratio of the product when there are more products.
The value must be positive. Further, the branching ratios of all products are

normalized in order to sum to one.
The default value 1.0, should only be used in the case of single product.

---

abstract type: **LinearODESolver**

Descendants:

Solver of a linear system of ODEs.

PadeApproximant

LinearODEAnalytic

---

record: **PadeApproximant** implements abstract type: LinearODESolver

---

Record with an information about pade approximant parameters.

TYPE = ⟨*selection: LinearODESolver_TYPE_selection* ⟩

Default: PadeApproximant

Sub-record selection.

nominator_degree = ⟨*Integer [1,]* ⟩

Default: 2

Polynomial degree of the nominator of Pade approximant.

denominator_degree = ⟨*Integer [1,]* ⟩

Default: 2

Polynomial degree of the nominator of Pade approximant

---

record: **LinearODEAnalytic** implements abstract type: LinearODESolver

---

Evaluate analytic solution of the system of ODEs.

TYPE = ⟨*selection: LinearODESolver_TYPE_selection* ⟩

Default: LinearODEAnalytic

Sub-record selection.

---

record: **RadioactiveDecay** implements abstract type: ReactionTerm

---

A model of a radioactive decay and possibly of a decay chain.

TYPE = ⟨*selection: ReactionTerm_TYPE_selection* ⟩

Default: RadioactiveDecay

Sub-record selection.

decays = ⟨*Array [1,] of record: Decay* ⟩

Default: ⟨*obligatory* ⟩

An array of radioactive decays.

`ode_solver` = ⟨*abstract type: LinearODESolver* ⟩

Default: ⟨*optional* ⟩

Numerical solver for the system of first order ordinary differential equations coming from the model.

---

record: **Decay**

A model of a radioactive decay.

`radionuclide` = ⟨*String (generic)* ⟩

Default: ⟨*obligatory* ⟩

The name of the parent radionuclide.

`half_life` = ⟨*Double [0,]* ⟩

Default: ⟨*obligatory* ⟩

The half life of the parent radionuclide in seconds.

`products` = ⟨*Array [1,] of record: RadioactiveDecayProduct* ⟩

Default: ⟨*obligatory* ⟩

An array of the decay products (daughters).

---

record: **RadioactiveDecayProduct** constructible from key: name

---

A record describing a product of a radioactive decay.

`name` = ⟨*String (generic)* ⟩

Default: ⟨*obligatory* ⟩

The name of the product.

`energy` = ⟨*Double [0,]* ⟩

Default: 0.0

Not used at the moment! The released energy in MeV from the decay of the radionuclide into the product.

`branching_ratio` = ⟨*Double [0,]* ⟩

Default: 1.0

The branching ratio of the product when there is more than one.Considering only one product, the default ratio 1.0 is used.Its value must be positive. Further, the branching ratios of all products are normalizedby their sum, so the sum then gives 1.0 (this also resolves possible rounding errors).

---

record: **Sorption** implements abstract type: ReactionTerm

Sorption model in the reaction term of transport.

`TYPE` = ⟨*selection: ReactionTerm_TYPE_selection* ⟩

Default: Sorption

Sub-record selection.

`substances` = ⟨*Array [1,] of String (generic)* ⟩

Default: ⟨*obligatory* ⟩

Names of the substances that take part in the sorption model.

`solvent_density` = ⟨*Double [0,]* ⟩

Default: 1.0

Density of the solvent.

`substeps` = ⟨*Integer [1,]* ⟩

Default: 1000

Number of equidistant substeps, molar mass and isotherm intersections

`solubility` = ⟨*Array of Double [0,]* ⟩

Default: ⟨*optional* ⟩

Specifies solubility limits of all the sorbing species.

`table_limits` = ⟨*Array of Double [0,]* ⟩

Default: ⟨*optional* ⟩

Specifies highest aqueous concentration in interpolation table.

`input_fields` = ⟨*Array of record: Sorption_Data* ⟩

Default: ⟨*obligatory* ⟩

Containes region specific data necessary to construct isotherms.

`reaction_liquid` = ⟨*abstract type: ReactionTerm* ⟩

Default: ⟨*optional* ⟩

Reaction model following the sorption in the liquid.

`reaction_solid` = ⟨*abstract type: ReactionTerm* ⟩

Default: ⟨*optional* ⟩

Reaction model following the sorption in the solid.

`output_fields` = ⟨*Array of selection: Sorption_Output* ⟩

Default: conc_solid

List of fields to write to output stream.

---

record: **Sorption_Data**

Record to set fields of the equation. 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_Data 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 have to form an increasing sequence.

`rock_density` = ⟨*abstract type: Field:R3 → Real* ⟩

Default: ⟨*optional* ⟩

Rock matrix density. $[m^{-3}kg]$

`sorption_type` = ⟨*abstract type: Field:R3 → Enum[n]* ⟩

Default: ⟨*optional* ⟩

Considered sorption is described by selected isotherm. If porosity on an element is equal or even higher than 1.0 (meaning no sorbing surface), then type 'none' will be selected automatically. [−]

`isotherm_mult` = ⟨*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. $[kg^{-1}mol]$

`isotherm_other` = ⟨*abstract type: Field:R3 → Real[n]* ⟩

Default: ⟨*optional* ⟩

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

`init_conc_solid` = ⟨*abstract type: Field:R3 → Real[n]* ⟩

Default: ⟨*optional* ⟩

Initial solid concentration of substances. Vector, one value for every substance. $[kg^{-1}mol]$

---

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

---

Descendants:

Abstract record for all time-space functions.

FieldConstant

FieldFormula

FieldPython

FieldInterpolatedP0

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 runs the concentration exchange between liquid and solid.

`langmuir` : Langmuir isotherm runs the concentration exchange between liquid and solid.

`freundlich` : Freundlich isotherm runs the concentration exchange between liquid and solid.

---

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: **FieldInterpolatedP0** implements abstract type: Field:R3 → Enum[n]

R3 → Enum[n] Field constant in space.

`TYPE` = ⟨*selection: Field:R3 → Enum[n]_TYPE_selection* ⟩

Default: FieldInterpolatedP0

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: **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]

R3 → Real[n] Field constant in space.

`TYPE` = ⟨*selection: Field:R3 → Real[n]_TYPE_selection* ⟩

Default: FieldInterpolatedP0

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.

selection type: **Sorption_Output**

Possible values:

`rock_density` : Output of the field rock_density $[m^{-3}kg]$ (Rock matrix density.).

`sorption_type` : Output of the field sorption_type [−] (Considered sorption is described by selected isotherm. If porosity on an element is equal or even higher than 1.0 (meaning no sorbing surface), then type 'none' will be selected automatically.).

`isotherm_mult` : Output of the field isotherm_mult $[kg^{-1}mol]$ (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.).

`isotherm_other` : Output of the field isotherm_other [−] (Second parameters (alpha, ...) defining isotherm c_s = omega * (alpha*c_a)/(1- alpha*c_a).).

`init_conc_solid` : Output of the field init_conc_solid $[kg^{-1}mol]$ (Initial solid concentration of substances. Vector, one value for every substance.).

`conc_solid` : Output of the field conc_solid $[m^{-3}kg]$.

---

record: **SorptionMobile** implements abstract type: ReactionTerm

---

Sorption model in the mobile zone, following the dual porosity model.

`TYPE` = ⟨*selection: ReactionTerm_TYPE_selection* ⟩

Default: SorptionMobile

Sub-record selection.

`substances` = ⟨*Array [1,] of String (generic)* ⟩

Default: ⟨*obligatory* ⟩

Names of the substances that take part in the sorption model.

`solvent_density` = ⟨*Double [0,]* ⟩

Default: 1.0

Density of the solvent.

`substeps` = ⟨*Integer [1,]* ⟩

Default: 1000

Number of equidistant substeps, molar mass and isotherm intersections

`solubility` = ⟨*Array of Double [0,]* ⟩

Default: ⟨*optional* ⟩

Specifies solubility limits of all the sorbing species.

`table_limits` = ⟨*Array of Double [0,]* ⟩

Default: ⟨*optional* ⟩

Specifies highest aqueous concentration in interpolation table.

`input_fields` = ⟨*Array of record: Sorption_Data* ⟩

Default: ⟨*obligatory* ⟩

Containes region specific data necessary to construct isotherms.

`reaction_liquid` = ⟨*abstract type: ReactionTerm* ⟩

Default: ⟨*optional* ⟩

Reaction model following the sorption in the liquid.

`reaction_solid` = ⟨*abstract type: ReactionTerm* ⟩

Default: ⟨*optional* ⟩

Reaction model following the sorption in the solid.

`output_fields` = ⟨*Array of selection: SorptionMobile_Output* ⟩

Default: conc_solid

List of fields to write to output stream.

---

selection type: **SorptionMobile_Output**

Possible values:

`rock_density` : Output of the field rock_density $[m^{-3}kg]$ (Rock matrix density.).

`sorption_type` : Output of the field sorption_type $[-]$ (Considered sorption is described by selected isotherm. If porosity on an element is equal or even higher than 1.0 (meaning no sorbing surface), then type 'none' will be selected automatically.).

`isotherm_mult` : Output of the field isotherm_mult $[kg^{-1}mol]$ (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.).

`isotherm_other` : Output of the field isotherm_other $[-]$ (Second parameters (alpha, ...) defining isotherm c_s = omega * (alpha*c_a)/(1- alpha*c_a).).

`init_conc_solid` : Output of the field init_conc_solid $[kg^{-1}mol]$ (Initial solid concentration of substances. Vector, one value for every substance.).

`conc_solid` : Output of the field conc_solid $[m^{-3}kg]$.

---

record: **SorptionImmobile** implements abstract type: ReactionTerm

---

Sorption model in the immobile zone, following the dual porosity model.

`TYPE` = ⟨*selection: ReactionTerm_TYPE_selection* ⟩

Default: SorptionImmobile

Sub-record selection.

`substances` = ⟨*Array [1,] of String (generic)* ⟩

Default: ⟨*obligatory* ⟩

Names of the substances that take part in the sorption model.

`solvent_density` = ⟨*Double [0,]* ⟩

Default: 1.0

Density of the solvent.

`substeps` = ⟨*Integer [1,]* ⟩

Default: 1000

Number of equidistant substeps, molar mass and isotherm intersections

`solubility` = ⟨*Array of Double [0,]* ⟩

Default: ⟨*optional* ⟩

Specifies solubility limits of all the sorbing species.

`table_limits` = ⟨*Array of Double [0,]* ⟩

Default: ⟨*optional* ⟩

Specifies highest aqueous concentration in interpolation table.

`input_fields` = ⟨*Array of record: Sorption_Data* ⟩

Default: ⟨*obligatory* ⟩

Containes region specific data necessary to construct isotherms.

`reaction_liquid` = ⟨*abstract type: ReactionTerm* ⟩

Default: ⟨*optional* ⟩

Reaction model following the sorption in the liquid.

`reaction_solid` = ⟨*abstract type: ReactionTerm* ⟩

Default: ⟨*optional* ⟩

Reaction model following the sorption in the solid.

`output_fields` = ⟨*Array of selection: SorptionImmobile_Output* ⟩

Default: conc_immobile_solid

List of fields to write to output stream.

---

selection type: **SorptionImmobile_Output**

Possible values:

`rock_density` : Output of the field rock_density $[m^{-3}kg]$ (Rock matrix density.).

`sorption_type` : Output of the field sorption_type [−] (Considered sorption is described by selected isotherm. If porosity on an element is equal or even higher than 1.0 (meaning no sorbing surface), then type 'none' will be selected automatically.).

`isotherm_mult` : Output of the field isotherm_mult $[kg^{-1}mol]$ (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.).

`isotherm_other` : Output of the field isotherm_other [−] (Second parameters (alpha, ...) defining isotherm c_s = omega * (alpha*c_a)/(1- alpha*c_a).).

`init_conc_solid` : Output of the field init_conc_solid $[kg^{-1}mol]$ (Initial solid concentration of substances. Vector, one value for every substance.).

`conc_immobile_solid` : Output of the field conc_immobile_solid $[m^{-3}kg]$.

---

record: **DualPorosity** implements abstract type: ReactionTerm

---

Dual porosity model in transport problems. Provides computing the concentration of substances in mobile and immobile zone.

`TYPE` = ⟨*selection: ReactionTerm_TYPE_selection* ⟩

Default: DualPorosity

Sub-record selection.

`input_fields` = ⟨*Array of record: DualPorosity_Data* ⟩

Default: ⟨*obligatory* ⟩

Containes region specific data necessary to construct dual porosity model.

`scheme_tolerance` = ⟨*Double [0,]* ⟩

Default: 1e-3

Tolerance according to which the explicit Euler scheme is used or not.Set 0.0 to use analytic formula only (can be slower).

`reaction_mobile` = ⟨*abstract type: ReactionTerm* ⟩

Default: ⟨*optional* ⟩

Reaction model in mobile zone.

`reaction_immobile` = ⟨*abstract type: ReactionTerm* ⟩

Default: ⟨*optional* ⟩

Reaction model in immobile zone.

`output_fields` = ⟨*Array of selection: DualPorosity_Output* ⟩

Default: conc_immobile

List of fields to write to output stream.

---

record: **DualPorosity_Data**

Record to set fields of the equation. 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 DualPorosity_Data 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 have to form an increasing sequence.

`diffusion_rate_immobile` = ⟨*abstract type: Field:R3 → Real[n]* ⟩

Default: ⟨*optional* ⟩

Diffusion coefficient of non-equilibrium linear exchange between mobile and immobile zone. $[s^{-1}]$

`porosity_immobile` = ⟨*abstract type: Field:R3 → Real* ⟩

Default: ⟨*optional* ⟩

Porosity of the immobile zone. [−]

`init_conc_immobile` = ⟨*abstract type: Field:R3 → Real[n]* ⟩

Default: ⟨*optional* ⟩

Initial concentration of substances in the immobile zone. $[m^{-3}kg]$

---

selection type: **DualPorosity_Output**

Possible values:

`diffusion_rate_immobile` : Output of the field diffusion_rate_immobile $[s^{-1}]$ (Diffusion coefficient of non-equilibrium linear exchange between mobile and immobile zone.).

`porosity_immobile` : Output of the field porosity_immobile [−] (Porosity of the immobile zone.).

`init_conc_immobile` : Output of the field init_conc_immobile $[m^{-3}kg]$ (Initial concentration of substances in the immobile zone.).

`conc_immobile` : Output of the field conc_immobile $[m^{-3}kg]$.

---

record: **Semchem** implements abstract type: ReactionTerm

Declares infos valid for all reactions. NOT SUPPORTED!!!.

`TYPE` = ⟨*selection: ReactionTerm_TYPE_selection* ⟩

Default: Semchem

Sub-record selection.

`precision` = ⟨*Integer* ⟩

Default: ⟨*obligatory* ⟩

How accurate should the simulation be, decimal places(?).

`temperature` = ⟨*Double* ⟩

Default: ⟨*obligatory* ⟩

Isothermal reaction, thermodynamic temperature.

`temp_gf` = ⟨*Double* ⟩

Default: ⟨*obligatory* ⟩

Thermodynamic parameter.

`param_afi` = ⟨*Double* ⟩

Default: ⟨*obligatory* ⟩

Thermodynamic parameter.

`param_b` = ⟨*Double* ⟩

Default: ⟨*obligatory* ⟩

Thermodynamic parameter.

`epsilon` = ⟨*Double* ⟩

Default: ⟨*obligatory* ⟩

Thermodynamic parameter.

`time_steps` = ⟨*Integer* ⟩

Default: ⟨*obligatory* ⟩

Simulation parameter.

`slow_kinetic_steps` = ⟨*Integer* ⟩

Default: ⟨*obligatory* ⟩

Simulation parameter.

---

record: **TransportOperatorSplitting_Data**

Record to set fields of the equation. 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_Data 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 have to form an increasing sequence.

`porosity` = ⟨*abstract type: Field:R3 → Real* ⟩

Default: ⟨*optional* ⟩

Mobile porosity [−]

`sources_density` = ⟨*abstract type: Field:R3 → Real[n]* ⟩

Default: ⟨*optional* ⟩

Density of concentration sources. $[m^{-3}kgs^{-1}]$

`sources_sigma` = ⟨*abstract type: Field:R3 → Real[n]* ⟩

Default: ⟨*optional* ⟩

Concentration flux. $[s^{-1}]$

`sources_conc` = ⟨*abstract type: Field:R3 → Real[n]* ⟩

Default: ⟨*optional* ⟩

Concentration sources threshold. $[m^{-3}kg]$

`bc_conc` = ⟨*abstract type: Field:R3 → Real[n]* ⟩

Default: ⟨*optional* ⟩

Boundary conditions for concentrations. $[m^{-3}kg]$

`init_conc` = ⟨*abstract type: Field:R3 → Real[n]* ⟩

Default: ⟨*optional* ⟩

Initial concentrations. $[m^{-3}kg]$

`transport_old_bcd_file` = ⟨*input file name* ⟩

Default: ⟨*optional* ⟩

File with mesh dependent boundary conditions (obsolete).

---

selection type: **ConvectionTransport_Output**

Possible values:

`porosity` : Output of the field porosity [−] (Mobile porosity).

`sources_density` : Output of the field sources_density $[m^{-3}kgs^{-1}]$ (Density of concentration sources.).

`sources_sigma` : Output of the field sources_sigma $[s^{-1}]$ (Concentration flux.).

`sources_conc` : Output of the field sources_conc $[m^{-3}kg]$ (Concentration sources threshold.).

`init_conc` : Output of the field init_conc $[m^{-3}kg]$ (Initial concentrations.).

`conc` : Output of the field conc $[m^{-3}kg]$.

`region_id` : Output of the field region_id [−].

---

record: **SoluteTransport_DG** implements abstract type: Transport

---

DG solver for solute transport.

`TYPE` = ⟨*selection: Transport_TYPE_selection* ⟩

Default: SoluteTransport_DG

Sub-record selection.

`time` = ⟨*record:* *TimeGovernor* ⟩

Default: ⟨*obligatory* ⟩

Time governor setting for the secondary equation.

`balance` = ⟨*record:* *Balance* ⟩

Default: ⟨*obligatory* ⟩

Settings for computing balance.

`output_stream` = ⟨*record:* *OutputStream* ⟩

Default: ⟨*obligatory* ⟩

Parameters of output stream.

`substances` = ⟨*Array of String (generic)* ⟩

Default: ⟨*obligatory* ⟩

Names of transported substances.

`solver` = ⟨*record:* *Petsc* ⟩

Default: ⟨*obligatory* ⟩

Linear solver for MH problem.

`input_fields` = ⟨*Array of record:* *SoluteTransport_DG_Data* ⟩

Default: ⟨*obligatory* ⟩

`dg_variant` = ⟨*selection:* *DG_variant* ⟩

Default: non-symmetric

Variant of interior penalty discontinuous Galerkin method.

`dg_order` = ⟨*Integer [0, 3]* ⟩

Default: 1

Polynomial order for finite element in DG method (order 0 is suitable if there is no diffusion/dispersion).

`output_fields` = ⟨*Array of selection:* *SoluteTransport_DG_Output* ⟩

Default: conc

List of fields to write to output file.

---

record: **SoluteTransport_DG_Data**

Record to set fields of the equation. 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 SoluteTransport_DG_Data 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 have to form an increasing sequence.

`porosity` = ⟨*abstract type: Field:R3 → Real* ⟩

Default: ⟨*optional* ⟩

Mobile porosity $[-]$

`sources_density` = ⟨*abstract type: Field:R3 → Real[n]* ⟩

Default: ⟨*optional* ⟩

Density of concentration sources. $[m^{-3}kgs^{-1}]$

`sources_sigma` = ⟨*abstract type: Field:R3 → Real[n]* ⟩

Default: ⟨*optional* ⟩

Concentration flux. $[s^{-1}]$

`sources_conc` = ⟨*abstract type: Field:R3 → Real[n]* ⟩

Default: ⟨*optional* ⟩

Concentration sources threshold. $[m^{-3}kg]$

`bc_conc` = ⟨*abstract type: Field:R3 → Real[n]* ⟩

Default: ⟨*optional* ⟩

Dirichlet boundary condition (for each substance). $[m^{-3}kg]$

`init_conc` = ⟨*abstract type: Field:R3 → Real[n]* ⟩

Default: ⟨*optional* ⟩

Initial concentrations. $[m^{-3}kg]$

`disp_l` = ⟨*abstract type: Field:R3 → Real[n]* ⟩

Default: ⟨*optional* ⟩

Longitudal dispersivity (for each substance). $[m]$

`disp_t` = ⟨*abstract type: Field:R3 → Real[n]* ⟩

Default: ⟨*optional* ⟩

Transversal dispersivity (for each substance). $[m]$

`diff_m` = ⟨*abstract type: Field:R3 → Real[n]* ⟩

Default: ⟨*optional* ⟩

Molecular diffusivity (for each substance). $[m^2 s^{-1}]$

`fracture_sigma` = ⟨*abstract type: Field:R3 → Real[n]* ⟩

Default: ⟨*optional* ⟩

Coefficient of diffusive transfer through fractures (for each substance). [−]

`dg_penalty` = ⟨*abstract type: Field:R3 → Real[n]* ⟩

Default: ⟨*optional* ⟩

Penalty parameter influencing the discontinuity of the solution (for each substance). Its default value 1 is sufficient in most cases. Higher value diminishes the inter-element jumps. [−]

`bc_type` = ⟨*abstract type: Field:R3 → Enum[n]* ⟩

Default: ⟨*optional* ⟩

Boundary condition type, possible values: inflow, dirichlet, neumann, robin. [−]

`bc_flux` = ⟨*abstract type: Field:R3 → Real[n]* ⟩

Default: ⟨*optional* ⟩

Flux in Neumann boundary condition. $[m^{1-d} kg s^{-1}]$

`bc_robin_sigma` = ⟨*abstract type: Field:R3 → Real[n]* ⟩

Default: ⟨*optional* ⟩

Conductivity coefficient in Robin boundary condition. $[m^{4-d} s^{-1}]$

---

selection type: **DG_variant**

Type of penalty term.

Possible values:

`non-symmetric` : non-symmetric weighted interior penalty DG method

`incomplete` : incomplete weighted interior penalty DG method

`symmetric` : symmetric weighted interior penalty DG method

---

selection type: **SoluteTransport_DG_Output**

Output record for DG solver for solute transport.

Possible values:

`porosity` : Output of the field porosity [−] (Mobile porosity).

`sources_density` : Output of the field sources_density $[m^{-3} kg s^{-1}]$ (Density of concentration sources.).

`sources_sigma` : Output of the field sources_sigma $[s^{-1}]$ (Concentration flux.).

`sources_conc` : Output of the field sources_conc $[m^{-3} kg]$ (Concentration sources threshold.).

`init_conc` : Output of the field init_conc $[m^{-3}kg]$ (Initial concentrations.).

`disp_l` : Output of the field disp_l $[m]$ (Longitudal dispersivity (for each substance).).

`disp_t` : Output of the field disp_t $[m]$ (Transversal dispersivity (for each substance).).

`diff_m` : Output of the field diff_m $[m^2s^{-1}]$ (Molecular diffusivity (for each substance).).

`conc` : Output of the field conc $[m^{-3}kg]$.

`fracture_sigma` : Output of the field fracture_sigma $[-]$ (Coefficient of diffusive transfer through fractures (for each substance).).

`dg_penalty` : Output of the field dg_penalty $[-]$ (Penalty parameter influencing the discontinuity of the solution (for each substance). Its default value 1 is sufficient in most cases. Higher value diminishes the inter-element jumps.).

`region_id` : Output of the field region_id $[-]$.

---

record: **HeatTransfer_DG** implements abstract type: Transport

---

DG solver for heat transfer.

`TYPE` = ⟨*selection: Transport_TYPE_selection* ⟩

Default: HeatTransfer_DG

Sub-record selection.

`time` = ⟨*record: TimeGovernor* ⟩

Default: ⟨*obligatory* ⟩

Time governor setting for the secondary equation.

`balance` = ⟨*record: Balance* ⟩

Default: ⟨*obligatory* ⟩

Settings for computing balance.

`output_stream` = ⟨*record: OutputStream* ⟩

Default: ⟨*obligatory* ⟩

Parameters of output stream.

`solver` = ⟨*record: Petsc* ⟩

Default: ⟨*obligatory* ⟩

Linear solver for MH problem.

`input_fields` = ⟨*Array of record: HeatTransfer_DG_Data* ⟩

Default: ⟨*obligatory* ⟩

`dg_variant` = ⟨*selection: DG_variant* ⟩

Default: non-symmetric

Variant of interior penalty discontinuous Galerkin method.

`dg_order` = ⟨*Integer [0, 3]* ⟩

Default: 1

Polynomial order for finite element in DG method (order 0 is suitable if there is no diffusion/dispersion).

`output_fields` = ⟨*Array of selection:* *HeatTransfer_DG_Output* ⟩

Default: temperature

List of fields to write to output file.

---

record: **HeatTransfer_DG_Data**

Record to set fields of the equation. 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 HeatTransfer_DG_Data 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 have to form an increasing sequence.

`bc_temperature` = ⟨*abstract type:* *Field:R3 → Real* ⟩

Default: ⟨*optional* ⟩

Boundary value of temperature. $[K]$

`init_temperature` = ⟨*abstract type:* *Field:R3 → Real* ⟩

Default: ⟨*optional* ⟩

Initial temperature. $[K]$

`porosity` = ⟨*abstract type:* *Field:R3 → Real* ⟩

Default: ⟨*optional* ⟩

Porosity. $[-]$

`fluid_density` = ⟨*abstract type:* *Field:R3 → Real* ⟩

Default: ⟨*optional* ⟩

Density of fluid. $[m^{-3}kg]$

`fluid_heat_capacity` = ⟨*abstract type: Field:R3 → Real* ⟩

Default: ⟨*optional* ⟩

Heat capacity of fluid. $[m^2 s^{-2} K^{-1}]$

`fluid_heat_conductivity` = ⟨*abstract type: Field:R3 → Real* ⟩

Default: ⟨*optional* ⟩

Heat conductivity of fluid. $[mkgs^{-3} K^{-1}]$

`solid_density` = ⟨*abstract type: Field:R3 → Real* ⟩

Default: ⟨*optional* ⟩

Density of solid (rock). $[m^{-3} kg]$

`solid_heat_capacity` = ⟨*abstract type: Field:R3 → Real* ⟩

Default: ⟨*optional* ⟩

Heat capacity of solid (rock). $[m^2 s^{-2} K^{-1}]$

`solid_heat_conductivity` = ⟨*abstract type: Field:R3 → Real* ⟩

Default: ⟨*optional* ⟩

Heat conductivity of solid (rock). $[mkgs^{-3} K^{-1}]$

`disp_l` = ⟨*abstract type: Field:R3 → Real* ⟩

Default: ⟨*optional* ⟩

Longitudal heat dispersivity in fluid. $[m]$

`disp_t` = ⟨*abstract type: Field:R3 → Real* ⟩

Default: ⟨*optional* ⟩

Transversal heat dispersivity in fluid. $[m]$

`fluid_thermal_source` = ⟨*abstract type: Field:R3 → Real* ⟩

Default: ⟨*optional* ⟩

Thermal source density in fluid. $[m^{-1} kgs^{-3}]$

`solid_thermal_source` = ⟨*abstract type: Field:R3 → Real* ⟩

Default: ⟨*optional* ⟩

Thermal source density in solid. $[m^{-1} kgs^{-3}]$

`fluid_heat_exchange_rate` = ⟨*abstract type: Field:R3 → Real* ⟩

Default: ⟨*optional* ⟩

Heat exchange rate in fluid. $[s^{-1}]$

`solid_heat_exchange_rate` = ⟨*abstract type: Field:R3 → Real* ⟩

Default: ⟨*optional* ⟩

Heat exchange rate of source in solid. $[s^{-1}]$

`fluid_ref_temperature` = ⟨*abstract type: Field:R3 → Real* ⟩

Default: ⟨*optional* ⟩

Reference temperature of source in fluid. $[K]$

`solid_ref_temperature` = ⟨*abstract type: Field:R3 → Real* ⟩

Default: ⟨*optional* ⟩

Reference temperature in solid. $[K]$

`fracture_sigma` = ⟨*abstract type: Field:R3 → Real[n]* ⟩

Default: ⟨*optional* ⟩

Coefficient of diffusive transfer through fractures (for each substance). $[-]$

`dg_penalty` = ⟨*abstract type: Field:R3 → Real[n]* ⟩

Default: ⟨*optional* ⟩

Penalty parameter influencing the discontinuity of the solution (for each substance). Its default value 1 is sufficient in most cases. Higher value diminishes the inter-element jumps. $[-]$

`bc_type` = ⟨*abstract type: Field:R3 → Enum[n]* ⟩

Default: ⟨*optional* ⟩

Boundary condition type, possible values: inflow, dirichlet, neumann, robin. $[-]$

`bc_flux` = ⟨*abstract type: Field:R3 → Real[n]* ⟩

Default: ⟨*optional* ⟩

Flux in Neumann boundary condition. $[m^{1-d}kgs^{-1}]$

`bc_robin_sigma` = ⟨*abstract type: Field:R3 → Real[n]* ⟩

Default: ⟨*optional* ⟩

Conductivity coefficient in Robin boundary condition. $[m^{4-d}s^{-1}]$

---

selection type: **HeatTransfer_DG_Output**

Selection for output fields of DG solver for heat transfer.

Possible values:

`init_temperature` : Output of the field init_temperature $[K]$ (Initial temperature.).

`porosity` : Output of the field porosity $[-]$ (Porosity.).

`fluid_density` : Output of the field fluid_density $[m^{-3}kg]$ (Density of fluid.).

`fluid_heat_capacity` : Output of the field fluid_heat_capacity $[m^2s^{-2}K^{-1}]$ (Heat capacity of fluid.).

`fluid_heat_conductivity` : Output of the field fluid_heat_conductivity $[mkgs^{-3}K^{-1}]$ (Heat conductivity of fluid.).

`solid_density` : Output of the field solid_density $[m^{-3}kg]$ (Density of solid (rock).).

`solid_heat_capacity` : Output of the field solid_heat_capacity $[m^2s^{-2}K^{-1}]$ (Heat capacity of solid (rock).).

`solid_heat_conductivity` : Output of the field solid_heat_conductivity $[mkgs^{-3}K^{-1}]$

(Heat conductivity of solid (rock).).

`disp_l` : Output of the field disp_l $[m]$ (Longitudal heat dispersivity in fluid.).

`disp_t` : Output of the field disp_t $[m]$ (Transversal heat dispersivity in fluid.).

`fluid_thermal_source` : Output of the field fluid_thermal_source $[m^{-1}kgs^{-3}]$ (Thermal source density in fluid.).

`solid_thermal_source` : Output of the field solid_thermal_source $[m^{-1}kgs^{-3}]$ (Thermal source density in solid.).

`fluid_heat_exchange_rate` : Output of the field fluid_heat_exchange_rate $[s^{-1}]$ (Heat exchange rate in fluid.).

`solid_heat_exchange_rate` : Output of the field solid_heat_exchange_rate $[s^{-1}]$ (Heat exchange rate of source in solid.).

`fluid_ref_temperature` : Output of the field fluid_ref_temperature $[K]$ (Reference temperature of source in fluid.).

`solid_ref_temperature` : Output of the field solid_ref_temperature $[K]$ (Reference temperature in solid.).

`temperature` : Output of the field temperature $[K]$.

`fracture_sigma` : Output of the field fracture_sigma $[-]$ (Coefficient of diffusive transfer through fractures (for each substance).).

`dg_penalty` : Output of the field dg_penalty $[-]$ (Penalty parameter influencing the discontinuity of the solution (for each substance). Its default value 1 is sufficient in most cases. Higher value diminishes the inter-element jumps.).

`region_id` : Output of the field region_id $[-]$.

Bibliography

[1] B. T. Bowman. Conversion of freundlich adsorption k values to the mole fraction format and the use of SY values to express relative adsorption of pesticides1. 46(4):740. ISSN 0361-5995. doi: 10.2136/sssaj1982.03615995004600040014x. URL https://www.soils.org/publications/sssaj/abstracts/46/4/SS0460040740?access=0&view=pdf.

[2] M. Císlerová and T. Vogel. *Transportní procesy*. ČVUT, 1998.

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

[4] P. A. Domenico and F. W. Schwartz. *Physical and chemical hydrogeology*, volume 824. Wiley New York, 1990.

[5] A. Ern, A. F. Stephansen, and P. Zunino. A discontinuous Galerkin method with weighted averages for advection–diffusion equations with locally small and anisotropic diffusivity. *IMA Journal of Numerical Analysis*, 29(2):235–256, 2009.

[6] A. Ern, A. F. Stephansen, and M. Vohralík. Guaranteed and robust discontinuous galerkin a posteriori error estimates for convection–diffusion–reaction problems. *Journal of computational and applied mathematics*, 234(1):114–130, 2010.

[7] V. Martin, J. Jaffré, and J. E. Roberts. Modeling fractures and barriers as interfaces for flow in porous media. *SIAM Journal on Scientific Computing*, 26(5):1667, 2005. ISSN 10648275. doi: 10.1137/S1064827503429363. URL http://link.aip.org/link/SJOCE3/v26/i5/p1667/s1&Agg=doi.

[8] R. Millington and J. Quirk. Permeability of porous solids. *Transactions of the Faraday Society*, 57:1200–1207, 1961.

[9] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling. *Numerical Recipes in C: The Art of Scientific Computing, Second Edition*. Cambridge University Press, 2 edition edition. ISBN 9780521431088. URL http://www.nrbook.com/a/bookcpdf.php.

[10] A. Younes, P. Ackerer, and F. Lehmann. A new mass lumping scheme for the mixed hybrid finite element method. *Int. J. Numer. Meth. Engng*, 67:89–107, 2006.