Technical university of Liberec

Faculty of mechatronics, informatics
and interdisciplinary studies

FLOW123D - draft, scheme of new inputs

version 1.7.0

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

Liberec, 2011

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

Chapter 1

Overview

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

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

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

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

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

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

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

1.1 Basic usage

1.1.1 How to run the simulation.

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

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

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

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

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

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

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

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

You can use script `flow123d.sh` to start parallel jobs or limit resources used by the program. This script accepts the same parameters as the program itself and further some additional parameters.

-h
: Usage overview.

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

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

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

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

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

On the Windows system you have to use Cygwin libraries in other to emulate Linux API. The program with libraries can be downloaded form:

`http://dev.nti.tul.cz/~brezina/flow123d_packages/`

Then you can start a sequential run by command:

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

or a parallel run by command:

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

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

1.1.2 Structure of input

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

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

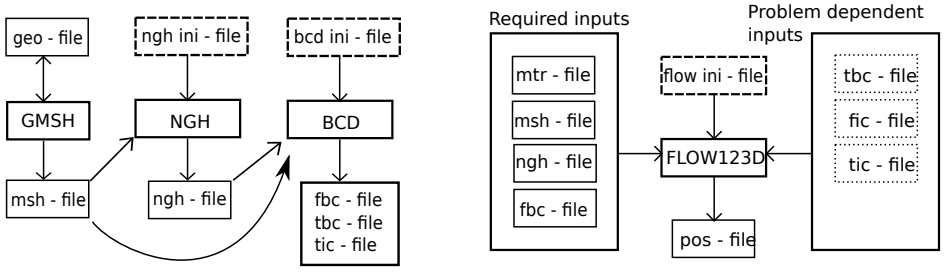

Figure 1.1: Preparation of input files.

For the preparation of input files we use several utilities (see Figure 1.1). We usually begin with a `*.geo` file as a description of the domain geometry. This come as an input for the GMSH mesh generator, which produce the mesh file. Then we run program `ngh` to produce adjacency file. Finally we can use program `bcd` for the preparation of files with boundary and initial conditions. The file with material properties has to be created manually, preferably by modifying some of the example problems. The programs `ngh` and `bcd` are distributed together with flow123d with their own limited documentation.

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

Chapter 2

Input files

The input consists of the root input file given as the parameter on the command line and possibly several other files with large input data. In this section we shall describe format of the input files. At first, we specify syntax of an extension to the JSON file format. Then we set rules for input of more specific data constructs. We continue by description of general scheme for input of boundary conditions and material time-space variable data. And finally, we describe setting of particular equations and their solvers.

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

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

[... something else?]

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

2.1 Root file format

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

1. You can use one line comments using hash `#`.
2. The quoting of the keys is optional if they do not contain spaces (holds for all Flow keys).

3. You can use equality sign `=` instead of colon `:` for separation of keys and values in JSON objects.

4. You can use any whitespace to separate tokens in JSON object or JSON array.

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

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

2.1.1 Humanized JSON

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

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

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

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

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

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

The second composed entity is the *array* which is sequence of (basic or composed) entities separated by comma (generic syntax) or whitespace sequence (extended syntax) and enclosed in the square brackets. The values stored in the array are accessed through the order. The

Flow reader offers either initialization of a container from JSON array or a sequential access. The latter one is the only possible access for the included arrays, which we discuss later.

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

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

2.1.2 Special keys

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

TYPE :

```
TYPE= <enum>
```

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

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

INCLUDE_RECORD :

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

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

INCLUDE_ARRAY :

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

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

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

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

[... better specify format string]

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

REFERENCE :

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

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

For example assume the file

```
mesh={
file_name="xyz"
}
array=[
{x=1, y=0}
{x=2, y=0}
{x=3, y=0}
]
outer_record={
output_file="x_out"
inner_record={
output_file={REF=["..","output_file"]}  # value "x_out"
}
x={REF="/array/2/x"} # value "3"
f_name={REF=["//","mesh","file_name"]} # value "xyz"
}
```

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

2.1.3 Semantic rules

Implicit creation of composed entities

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

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

Enum construct

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

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

String types

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

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

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

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

[Developers should provide default names to all files.]

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

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

text string Just text without particular meaning.

Record types

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

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

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

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

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

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

[? polymorphic record types]

2.2 Record types for input of data fields

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

2.2.1 Mesh type

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

record type: **Mesh**

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

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

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

The set of 0,1, or 2 dimensional boundary faces of the mesh should be partitioned into boundary segments in order to prescribe unique boundary condition on every boundary face. The segments numbers are assigned to boundary faces by iterating through the

array. Initially every boundary face has segment number 0. Every record in the array use "auxiliary" physical domains, elements or direct face specification to specify some set of boundary faces. The new segment number is assigned to each face in the set, possibly overwriting previous value.

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

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

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

---

record type: **Boundary segment**

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

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

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

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

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

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

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

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

2.2.2 Time-space field type

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

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

record type: **Steady field data**

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

Start time for the spatial filed data.

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

time interpolation enum cases:

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

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

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

record type: **Space function**

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

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

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

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

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

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

FE space enum cases:

`None=0` Use analytic function.

`P0=1` Zero order polynomial on element.

Currently we support only P0 base functions for data.

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

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

record type: **Element function**

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

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

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

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

double for scalar functions.

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

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

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

[How to allow parallel input of field data?]

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

Examples:

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

conductivity_tensor =
  [{ material = 1,
     rank = 2,
     analytic = [[1.0, 0.0, 0.0], [0.0, 1.0, 0.0], [0.0, 0.0, 1.0]]
   },
   { material = 2,
     rank = 2,
     analytic = [[10.0, 0.0, 0.0], [0.0, 10.0, 0.0], [0.0, 0.0, 10.0]]
   }
  ]
```

2.2.3 Boundary conditions

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

record type: **steady boundary data**

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

Time when the BC should be applied.

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

record type: **boundary condition**

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

Boundary segment where the boundary condition will be applied.

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

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

`dirichlet=0`

`neumann=1`

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

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

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

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

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

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

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

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

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

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

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

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

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

[More examples...]

2.3 Other record types recognized by Flow123d

2.3.1 Record types not related to equations

record type: **Root record**

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

Record with application setting.

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

Record with numerical problem to solve.

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

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

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

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

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

record type: **System**

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

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

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

Turns on/off more verbose mode.

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

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

vtk ascii stream:

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

GMSH ascii stream:

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

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

record type: **Output stream**

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

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

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

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

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

output format enum cases:

`vtk_ascii=0`

`gmsh_ascii=1`

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

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

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

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

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

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

2.3.2 Equation related record types

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

The *s*equential coupling problem has following keys:

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

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

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

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

The computational mesh common for both coupled equations.

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

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

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

Independent equation.

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

Equation with some data dependent on the primary equation.

---

record type: **Time governor**

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

Time when an equation starts its simulation.

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

Initial time step.

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

End time for an equation.

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

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

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

`TYPE` = <*enum*> DEFAULT:

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

`steady_MH=0`

`unsteady_LMH=1`

`unsteady_MH=2`

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

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

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

File with sources in old format. (OBSOLETE)

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

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

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

New scheme for setting boundary conditions.

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

File with boundary conditions in old format. (OBSOLETE)

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

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

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

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

This is just sub record to separate output setting.

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

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

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

File with initial condition in old format. (OBSOLETE)

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

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

---

record type: **Darcy flow output**

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

Time step between outputs.

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

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

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

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

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

2.3.3 Solver type

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

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

solver types enum cases:

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

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

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

Absolute residual tolerance.

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

Maximum number of outer iterations.

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

String with options for PETSc solvers.

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

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

2.3.4 Transport type

record type: **Transport type**

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

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

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

What kind of reactions is this? Age of water?

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

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

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

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

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

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

Vector valued initial condition for mobile phase of all species.

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

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

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

File with initial condition in old format. (OBSOLETE)

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

New scheme for setting boundary conditions.

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

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

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

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

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

record type: **Transport output**

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

Time step between outputs.

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

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

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

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

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

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

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

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

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

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

record type: **Reaction**

record type: **Decay chain**

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

Sequence of N ids of transported substances describing the order of isotopes in the decay chain.

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

This array of $N - 1$ half-lives of individual decays. If there are no bifurcation key specified, the decay chain is linear $1 \to 2 \to 3$. If there is the bifurcation key, the decay chain is branched $1 \to 2, 1 \to 3$.

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

Contains $N - 1$ probabilities for individual branches of the bifurcation decay. They should sum to one.

2.4 Flow123D ini file format (OBSOLETE)

In this section we briefly descripte every option you can use in global INI-file. Options marked (NOT IMPLEMENTED) are not implemented in current version, but probably will be reimplemented in near future. Options marked (NOT SUPPORTED) has not been tested since they are obsolete and will be replaced by similar functionality.

Section: [**Global**]

KEY	TYPE	DEFAULT	DESCRIPTION
`Problem_type`	**int**	NULL	Type of solved problem. Currently supported: 1 = steady saturated flow 2 = unsteady saturated flow using MH method 4 = unstedy saturated flow using Lumped MH method
`Description`	**string**	*undefined*	Short description of solved problem - any text.
`Stop_time`	**double**	1.0	Time interval of the whole problem.[time units]
`Time_step`	**double**	1.0	Time step for unstedy water flow solver. [time units]
`Save_step`	**double**	1.0	The output with transport is written every `Save_step`. [time units]

Section: [**Input**]

KEY	TYPE	DEFAULT	DESCRIPTION
`Mesh`	**string**	NULL	Name of file containig definition of the mesh for the problem.
`Material`	**string**	NULL	Name of file with hydraulical properties of the elements.
`Boundary`	**string**	NULL	Name of file with boundary condition data.
`Neighbouring`	**string**	NULL	Name of file describing topology of the mesh.
`Initial`	**string**	NULL	Name of file with initial condition for preasure head [length].
`Sources`	**string**	NULL	Name of file with definition of fluid sources. This is optional file, if this key is not defined, calculation goes on without sources.
`sources_formula`	**string**	NULL	Expression for sources as function of space coordinates x, y, z. See documentation of FParser library: `http://warp.povusers.org/FunctionParser/fparser.html#literals`

Section: [**Transport**]

KEY	TYPE	DEFAULT	DESCRIPTION
`Transport_on`	**YES/NO**	NO	If set "YES" program compute transport too.
`Sorption`	**YES/NO**	NO	If set "YES" program include sorption too.
`Dual_porosity`	**YES/NO**	NO	If set "YES" program include dual porosity too.
`Reactions`	**YES/NO**	NO	If set "YES" program include reactions too.
`Concentration`	**string**	NULL	Name of file with initial condition for concentrations of individual substances.
`Transport_BCD`	**string**	NULL	Name of file with boundary condition for transport.
`Transport_out`	**string**	NULL	Name of transport output file.
`Transport_out_im`	**string**	NULL	(NOT IMPLEMENTED IN 1.6.5) Name of transport immobile output file.
`Transport_out_sorp`	**string**	NULL	(NOT IMPLEMENTED IN 1.6.5) Name of transport sorbed output file.
`Transport_out_im_sorp`	**string**	NULL	(NOT IMPLEMENTED IN 1.6.5) Name of transport sorbed immobile output file.
`N_substances`	**int**	-1	Number of substances.
`Substances`	**string**	*undefined*	Names of the substances separated by commas.
`bc_times`	**list of doubles**	NULL	Times for changing boundary conditions. If you set this variable, you have to prepare a separate file with boundary condintions for every time in the list. Filenames for individual time level are formed from BC filename by appending underscore and three digits of time level number, e.g. `transport_bcd_000, transport_bcd_001, etc.`

Section: [**Run**]

KEY	TYPE	DEFAULT	DESCRIPTION
`Screen_verbosity`	**int**	0	Nonzero value turn on a more verbose mode (more messages on screen), however everything is in the log file.
`Pause_after_run`	**YES/NO**	NO	Wait for Enter after end of progam in order to keep output screen open.

Section: **[Solver]**

KEY	TYPE	DEFAULT	DESCRIPTION
`Use_last_solution`	**YES/NO**	NO	If set to "YES", uses last known solution for chosen solver.
`Solver_name`	**string**	petsc	Type of linear solver. Supported solvers are: `petsc`, `petsc_matis` (experimental)

`Solver_params`	**string**	NULL	PETSc options to override default choice of iterative solver and preconditioner (use with care). In particular to use UMFPACK sequantial direct solver set:
			`Solve_params = "-ksp preonly -pc_type lu -pc_factor_mat_solver_package umfpack"`
			To use parallel direct solver MUMPS use:
			`Solve_params = "-ksp preonly -pc_type lu -pc_factor_mat_solver_package mumps -mat_mumps_icntl_14 5"`

`Keep_solver_files`	**YES/NO**	NO	(NOT SUPPROTED IN 1.6.5) If set to "YES", files for solver are not deleted after the run of the solver.
`Manual_solver_run`	**YES/NO**	NO	(NOT SUPPROTED IN 1.6.5) If set to "YES", programm stops after writing input files for solver and lets user to run it.
`Use_control_file`	**YES/NO**	NO	(NOT SUPPROTED IN 1.6.5) If set to "YES", programm do not create control file for solver, it uses given file.
`Control_file`	**string**	NULL	(NOT SUPPROTED IN 1.6.5) Name of control file for situation, when `Use_control_file` ȲES.
`NSchurs`	**int**	2	Number of Schur complements to use. Valid values are 0,1,2. The last one should be the fastest.
`Solver_accuracy`	**double**	1e-6	When to stop solver run - value of residum of matrix. Useful values from 1e-4 to 1e-10. Bigger number = faster run, less accuracy.
`max_it`	**int**	200	Maximum number of iteration of linear solver.

Section: [**Output**]

KEY	TYPE	DEFAULT	DESCRIPTION
`Write_output_file`	**YES/NO**	NO	If set to "YES", writes output file.
`Output_file`	**string**	NULL	Name of the output file for water flow output.
`Output_file_2`	**string**	NULL	(NOT IMPLEMENTED IN 1.6.5) Name of the output file (type 2).
`Output_digits`	**int**	6	(NOT IMPLEMENTED IN 1.6.5) Number of digits used for floating point numbers in output file.
`Output_file_type`	**int**	1	(NOT IMPLEMENTED IN 1.6.5) Type of output file 1 - GMSH like format 2 - Flow data file 3 - both files (two separate names)
`Pos_format`	**string**	ASCII	Output file format. One can use: ASCII, BIN, or VTK_SERIAL_ASCII
`balance_output`	**string**	NULL	Name of file for output of water boundary fluxes and balnace of sources over material subdomains.

Description: Options controling output file of the programm

Section: [**Semchem_module**]

KEY	TYPE	DEFAULT	DESCRIPTION
`Compute_reactions`	**Yes/No**	"No"	NO = transport without chemical reactions YES = transport influenced by chemical reactions
`Output_precission`	**int**	1	Number of decimal places written to output file created by Semchem_module.
`Number_of_further_species`	**int**	0	Concentrations of these species are not computed, because they are ment to be unexghaustible.
`Temperature`	**double**	0.0	Temperature, one of state variables of the system.
`Temperature_Gf`	**double**	0.0	Temperature at which Free Gibbs Energy is specified.
`Param_Afi`	**double**	0.0	Parameter of the Debuy-Hückel equation for activity coeficients computation.
`Param_b`	**double**	0.0	Parameter of the Debuy-Hückel equation for activity coeficients computation.
`Epsilon`	**double**	0.0	Epsilon specifies relative norm of residuum estimate to stop numerical algorithms used by Semchem_module.
`Time_steps`	**int**	1	Number of transport step subdivisions for Semchem_module.
`Slow_kinetics_substeps`	**int**	0	Number of substeps performed by Runge-Kutta method used for slow kinetics simulation.
`Error_norm_type`	**string**	"Absolute"	Through wich kind of norm the error is measured.
`Scalling`	**boolean**	"No"	Type of the problem preconditioning for better convergence of numerical method.

Section: [**Aqueous_species**]

KEY	TYPE	DEFAULT	DESCRIPTION
`El_charge`	**int**	0	Electric charge of an Aqueous_specie particleunder consideration.
`dGf`	**double**	0.0	Free Gibbs Energy valid for TemperatureGf.
`dHf`	**double**	0.0	Enthalpy
`Molar_mass`	**double**	0.0	Molar mass of Aqueous_species.

Section: [**Further_species**]

KEY	TYPE	DEFAULT	DESCRIPTION
`Specie_name`	**string**	""	Name belonging to Further_specie under consideration.
`dGf`	**double**	0.0	Free Gibbs Energy valid for TemperatureGf.
`dHf`	**double**	0.0	Enthalpy
`Molar_mass`	**double**	0.0	Molar mass of Further_species.
`Activity`	**double**	0.0	Activity of Further_species.

Section: [**Reaction_i**]

KEY	TYPE	DEFAULT	DESCRIPTION
`Reaction_type`	**string**	"unknown"	Type of considered reaction (Equilibrium, Kinetics, Slow_kinetics).
`Stoichiometry`	**int**	0	Stoichiometric coeficients of species taking part in i-th reaction.
`Kinetic_constant`	**double**	0.0	Kinetic constant for determination of reaction rate.
`Order_of_reaction`	**int**	0	Order of kinetic reaction for participating species.
`Equilibrium_constant`	**double**	0.0	Equilibrium constant defining i-th reaction.

Section: [**Reaction_module**]

KEY	TYPE	DEFAULT	DESCRIPTION
`Compute_decay`	**Yes/No**	"No"	Enables to switch on simulation of radioactive decay.
`Nr_of_decay_chain`	**int**	0	How many steps is defined decay chain consisting of?

Section: [**Decay_i**]

KEY	TYPE	DEFAULT	DESCRIPTION
`Reaction_type`	**enumerate**	"decay"	In this time just decay type is possible.
`Nr_of_isotopes`	**int**	0	How many isotopes does the section work with.
`Substance_ids`	**array of int**	NULL	Sequence of ids describing the order of isotopes in decay chain.
`Half_lives`	**array of double**	NULL	Contain half-lives belonging to isotopes defined by ids.
`Bifurcation_on`	**Yes/No**	"No"	Make it possible to switch of bifurcation in current [Decay_i].
`Bifurcation`	**array of double**	NULL	Gives a percentage, which is the first isotope in current [Decay_i] decaying to products.

2.4.1 Mesh file format version 2.0

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

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

Comments concerning Flow123d:

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

  tag1 is number of geometry region in which the element lies.

  tag2 is number of material (reference to `.MTR` file) in the element.

  tag3 is partition number (CURRENTLY NOT USED).
- Currently, line (*type* = 1), triangle (*type* = 2) and tetrahedron (*type* = 4) are the only supported types of elements. Existence of an element of different type stops the calculation.
- Wherever possible, we use the file extension `.MSH`. It is not required, but highly recomended.

2.4.2 Material properties file format, version 1.0

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

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

where:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

optional.

Comments concerning `1-2-3-FLOW`:

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

2.4.3 Boundary conditions file format, version 1.0

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

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

where

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

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

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

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

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

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

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

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

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

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

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

Types of boundary conditions and their data

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

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

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

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

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

Ways of defining the place for the boundary condition

where = `1` — Condition on a node

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

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

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

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

Comments concerning `1-2-3-FLOW`:

- We assume homegemous Neumman's condition as the default one. Therefore we do not need to prescribe conditions on the whole boundary.
- If the condition is given on the inner edge, it is treated as an error and stops calculation.
- Any inconsistence in the file stops calculation. (Bad number of conditions, multiple definition of condition, reference to non-existing node, etc.)
- At least one of the conditions has to be of the Dirichlet's or Newton's type. This is well-known fact from the theory of the PDE's.
- Local numbers of sides for *where* = 2 must be lower than the number of sides of the particular element and greater then or equal to zero.
- The element specified for *where* = 3 must have only one external side, otherwise the program stops.

2.4.4 Neighbouring file format, version 1.0

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

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

where

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

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

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

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

type `int` — is type of the neighbouring.

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

Types of neighbouring and their specific data

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

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

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

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

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

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

Comments concerning `1-2-3-FLOW`:

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

2.4.5 Sources file format, version 1.0

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

`$SourceFormat`
`1.0` *file-type data-size*
`$EndSourceFormat`
`$Sources`
number-of-sources
eid density
`...`
`$EndSources`

where

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

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

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

eid `int` — is id-number of the element, where the source lies.

density `double` — is the density of the source, in volume of fluid per time unit. Possitive values are sources, negative are sinks.

Comments concerning `1-2-3-FLOW`:

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

ASCII post-processing file format version 1.2

File format of this file comes from the GMSH system. Following text is copied from the GMSH documentation.

`============== BEGIN OF INSERTED TEXT ==============`

The ASCII post-processing file is divided in several sections: one format section, enclosed between `$PostFormat-$EndPostFormat` tags, and one or more post-processing views, enclosed between `$View-$EndView` tags:

`$PostFormat`
`1.2` *file-type data-size*
`$EndPostFormat`
`$View`
view-name nb-time-steps
nb-scalar-points nb-vector-points nb-tensor-points
nb-scalar-lines nb-vector-lines nb-tensor-lines
nb-scalar-triangles nb-vector-triangles nb-tensor-triangles
nb-scalar-quadrangles nb-vector-quadrangles nb-tensor-quadrangles
nb-scalar-tetrahedra nb-vector-tetrahedra nb-tensor-tetrahedra
nb-scalar-hexahedra nb-vector-hexahedra nb-tensor-hexahedra
nb-scalar-prisms nb-vector-prisms nb-tensor-prisms
nb-scalar-pyramids nb-vector-pyramids nb-tensor-pyramids
nb-text2d nb-text2d-chars nb-text3d nb-text3d-chars
<*time-step-values*>
<*scalar-point-values*>
<*vector-point-values*>
<*tensor-point-values*>
<*scalar-line-values*>
<*vector-line-values*>
<*tensor-line-values*>
<*scalar-triangle-values*>
<*vector-triangle-values*>
<*tensor-triangle-values*>
<*scalar-quadrangle-values*>
<*vector-quadrangle-values*>
<*tensor-quadrangle-values*>
<*scalar-tetrahedron-values*>
<*vector-tetrahedron-values*>
<*tensor-tetrahedron-values*>
<*scalar-hexahedron-values*>
<*vector-hexahedron-values*>
<*tensor-hexahedron-values*>
<*scalar-prism-values*>
<*vector-prism-values*>
<*tensor-prism-values*>

<scalar-pyramid-values>
<vector-pyramid-values>
<tensor-pyramid-values>
<text2d> *<text2d-chars>*
<text3d> *<text3d-chars>*
`$EndView`

where:

file-type is an integer equal to 0 in the ASCII file format.

data-size is an integer equal to the size of the floating point numbers used in the file (usually, *data-size* = sizeof(double)).

view-name is a string containing the name of the view (max. 256 characters).

nb-time-steps is an integer giving the number of time steps in the view.

nb-scalar-points, *nb-vector-points*, ... are integers giving the number of scalar points, vector points,... in the view.

nb-text2d, *nb-text3d* are integers giving the number of 2D and 3D text strings in the view.

nb-text2d-chars, *nb-text3d-chars* are integers giving the total number of characters in the 2D and 3D strings.

time-step-values is a list of *nb-time-steps* double precision numbers giving the value of the time (or any other variable) for which an evolution was saved.

scalar-point-value, *vector-point-value*, ... are lists of double precision numbers giving the node coordinates and the values associated with the nodes of the *nb-scalar-points* scalar points, *nb-vector-points* vector points,..., for each of the *time-step-values*.

For example, *vector-triangle-value* is defined as:

coord1-node1 coord1-node2 coord1-node3
coord2-node1 coord2-node2 coord2-node3
coord3-node1 coord3-node2 coord3-node3
comp1-node1-time1 comp2-node1-time1 comp3-node1-time1
comp1-node2-time1 comp2-node2-time1 comp3-node2-time1
comp1-node3-time1 comp2-node3-time1 comp3-node3-time1
comp1-node1-time2 comp2-node1-time2 comp3-node1-time2
comp1-node2-time2 comp2-node2-time2 comp3-node2-time2
comp1-node3-time2 comp2-node3-time2 comp3-node3-time2
...

text2d is a list of 4 double precision numbers:

coord1 coord2 style index

where *coord1* and *coord2* give the coordinates of the leftmost element of the 2D string in screen coordinates, *index* gives the starting index of the string in *text2d-chars* and *style* is currently unused.

text2d-chars is a list of *nb-text2d-chars* characters. Substrings are separated with the '^' character (which is a forbidden character in regular strings).

text3d is a list of 5 double precision numbers

coord1 coord2 coord3 style index

where *coord1*, *coord2* and *coord3* give the coordinates of the leftmost element of the 3D string in model (real world) coordinates, *index* gives the starting index of the string in *text3d-chars* and *style* is currently unused.

text3d-chars is a list of *nb-text3d-chars* chars. Substrings are separated with the '^' character.

```
============== END OF INSERTED TEXT ==============
```

More information about GMSH can be found at its homepage:
`http://www.geuz.org/gmsh/`

Comments concerning `FFLOW20`:

- `FFLOW20` generates `.POS` file with four views: Elements' pressure, edges' pressure, interelement fluxes and complex view. First three views shows "raw data", results obtained by the solver without any interpolations, smoothing etc. The fourth view contains data processed in this way.

  Elements' pressure: Contains only *scalar-triangle-values*. Triangles are the same as the elements of the original mesh. We prescribe constant value of the pressure on the element, as it was calculated by the solver as the unknown p. Therefore, the three values on every triangle are the same.

  Edge pressure: Contains only *scalar-line-values*. The lines are the same as the edges of the elements of the original mesh. We prescribe constant value of the pressure on the edge, as it was calculated by the solver as the unknown λ. Therefore, the two values on every edge are the same.

  Interelement flux: Contains *vector-point-values* and *scalar-triangle-values*. The *scalar-triangle-values* carry no information, all values are set to 0, these are in the file only to define a shape of the elements. The points for the *vector-point-values* are midpoints of the sides of the elements. The vectors are calculated as $u\mathbf{n}$, where u is value of the flux calculated by the solver and $\mathbf{n}$ is normalized vector of outer normal of the element's side.

  Complex view: Contains *scalar-triangle-values* and *vector-point-values*. The *scalar-triangle-values* shows the shape of the pressure field. The triangles are the the same as the elements of the original mesh. Values of pressure in nodes are interpolated from ps and λs. The *vector-point-values* shows the velocity of the flow in the centres of the elements.

2.5 Output files

Output data

2.5.1 Output data fields of water flow modul

2.5.2 Output data fields of transport

2.5.3 GMSH viewer remarks

2.5.4 Paraview viewer remarks