Delphin 6 Output File Specification

Technical Report

Stefan Vogelsang, Andreas Nicolai

Institut für Bauklimatik
Technische Universität Dresden
D-01062 Dresden, Germany
stefan.vogelsang@tu-dresden.de

Abstract

This paper describes the file formats of the output data and geometry files generated by the *Delphin* program, a simulation model for hygrothermal transport in porous media. The output data format is suitable for any kind of simulation output generated by transient transport simulation models. Implementing support for the *Delphin* output format enables use of the advanced post-processing functionality provided by the *Delphin* post-processing tool and its dedicated physical analysis functionality. The article also discusses the application programming interface of the *DataIO* library that can be used to read/write *Delphin* output data and geometry files conveniently and efficiently.

Contents

1 Introduction

Transient simulation models for transport processes typically solve partial differential equations, hereby producing output data depending on the numerical grid employed. An example for such simulation models is the *Delphin* simulation program that solves combined mass and energy transport problems in porous media [1]. The monitored quantities are either fields of variables, obtained for grids cells, nodes, and edges, or scalar quantities. In order to analyze simulation data in post-processing tools, information about the geometry and the actual simulation output

is needed. Since geometry data is common to many simulation output files, geometry data and actual output data is stored in separate files.

For each monitored quantity, meta information is needed to simplify/streamline post-processing tasks. Such meta information include, for example, the description and unit of the physical quantity in the data file, and the element, node, or edge references.

For complex simulation models, simulation data files can become large. To reduce disk space and most importantly increase data reading speed, output data files are typically written in binary format. Furthermore, the binary format allows on-demand access of sections of the data, which greatly reduces memory consumption during post-processing. A plain text format is also specified, which is meaningful for small/moderate grid sizes and when direct text-editing and copying of data sections into spreadsheet applications is required.

Both, binary and ASCII files can be written incrementally, meaning that new data fields are written when simulation output becomes available. Post-processing can already commence when output is still being generated. The *DataIO* library, discussed in §5, facilitates efficient reading and updating of data sets.

In the following sections, the file formats for data and geometry files are defined.

2 Magic Headers

The output files begin with the same header structure stored in the first 16 bytes of each file. The first 4 bytes encode the identity of a file. This information is used to automatically select the correct parser for each output file format. Table 1 lists the magic header numbers, that uniquely identify file type and format (binary/ASCII).

	ASCII format	binary format
geometry file	`0x41473644`	`0x42473644`
data file	`0x414F3644`	`0x424F3644`

Table 1: Magic header numbers

The following 4 bytes must be equal to the number `0x4C5A2100`. This allows disambiguation with other magic file headers and thus guarantees a unique identification in the UNIX world.

The next 8 bytes encode the file format version number consisting of a major and minor number. All minor version changes are downwards compatible. No content sections in the files will be removed or altered when a new minor version is released. Consequently, applications supporting older version of file formats can still read newer files formats. For example, minor versions may add meta information to the file headers that can simply be ignored.

Major version changes indicate significant changes in the file format and result in incompatible files. A major revision number change might require updates for older distributions of the *Delphin Post Processing* software.

The maximum major and minor version number is limited to 255, corresponding to an 8 bit number. This limit is introduced to ensure that the ASCII representation of a version number has at maximum 3 digits. Since this document specifies already version 6.0 of the file format, $249 * 256 = 63744$ version numbers are left within this format number definition.

2.1 Version Number Encoding in Binary Files

The binary version is encoded in two 32 bit unsigned int values (64 bit), to ensure the same length of the magic header for binary and ASCII files. The actual version numbers are encoded in the first 32 bit unsigned int, which is followed by a dummy 32 bit unsigned int, as shown in the following example (hex).

```
0x060F0000 0x00000000
```

Here, the first byte encodes the major version number (here 6), and the second byte encodes the minor version number (here 15, as 0F in hexadecimal).

The remaining 6 bytes of the 8 byte version block are reserved and must be zero.

2.2 Version Number Encoding in ASCII Files

The ASCII Version number must have exactly 8 characters (64 bit), as in the example below.

```
␣006.015
```

The version string begins with a single space character, followed by the version numbers. Major and minor version numbers are separated by a . (dot) character. Both version numbers must have exactly 3 digits, using the 0 as fill character.

3 Output Data File Format

Each output data file holds output values for a single quantity. This also applies to flux quantities. For example, x- and y- velocity components are written in two files. The post-processing software reads both data files and reconstructs the flow field from both data sets.

3.1 File Name Specification

While not part of the file format specification itself, a naming convention of file names is meaningful to organize multiple output data and geometry files, generated from different simulation runs.

Each output data file must be named after the following pattern:

```
<quantityName>_<projectHashCode>.<fileExtension>
```

`fileExtension` is defined as ".d6o" for plain text format and ".d6b" for binary format,

`quantityName` corresponds to the name of the quantity within in the file, or is a user-defined name and can be used to identify output data files by file name

`projectHashCode` is a result of a mathematical function calculated on the content of the simulation project file. As "simulation project file content" this document defines: A *string* containing all relevant data sections influencing geometry, element/node/edge indices and other data resulting in differences of the geometry file (see §4). Adding the hash code

1. allows automatic generation of the data file name itself,
2. allows unique mapping of a given data file to the correct geometry file, and
3. indicates, if different from other data files, that the project file content relevant to output data was altered.

For example, in a variation study a part of a simulation project file related to the geometry may be altered by script (e.g. in a grid sensitivity study) with the aim to compare generated results. Each set of simulation results then requires an own geometry file. Results from the last simulation run, and especially the geometry file should not be overwritten, since the interpretation of data files requires the corresponding geometry file. A common file name suffix (i.e. the `projectHashCode`) uniquely identifies groups of data files and their geometry file.

3.2 General File Layout

All *Delphin 6* output files are composed of three sections:

```
MAGIC HEADER SECTION
HEADER/META INFORMATION SECTION
DATA SECTION
```

The magic header section was already described above.

3.3 ASCII Format

The data file is written line-based and strings are encoded in *UTF8*.

3.3.1 Header Section

Two different types of outputs are created by the *Delphin 6* simulation tool: Field and Flux outputs. Field outputs are created when a quantity within an element is monitored, such as temperature or moisture content. Flux outputs are created when a flux over an element/cell boundary is monitored. The file type can be distinguished by the type information in the header.

A typical field output file header looks like:

```
TYPE          = FIELD
PROJECT_FILE  = D:\examples\hamstad_benchmark_4.dpj
CREATED       = Fri Nov 11 15:55:53 2005
QUANTITY      = Total moisture mass
QUANTITY_KW   = MoistureMassDensity
GEO_FILE      = hamstad_benchmark_4_12345678.g6o
GEO_FILE_HASH = 0x3FA08374
SPACE_TYPE    = SINGLE
TIME_TYPE     = NONE
VALUE_UNIT    = kg/m3
TIME_UNIT     = h
START_YEAR    = 2000
INDICES       = 1 2 3 4 10 11 12
```

A typical flux output file header looks like:

```
TYPE          = FLUX
PROJECT_FILE  = D:\examples\hamstad_benchmark_4.dpj
CREATED       = Fri Nov 11 15:55:53 2005
QUANTITY      = Heat flux
QUANTITY_KW   = FluxHeatConduction
GEO_FILE      = hamstad_benchmark_4_12345678.g6o
GEO_FILE_HASH = 0x3FA08374
SPACE_TYPE    = SINGLE
TIME_TYPE     = NONE
VALUE_UNIT    = W/m2
TIME_UNIT     = h
START_YEAR    = 2000
INDICES       = 0 1 2 50 51 101 102 103
```

Each line of the header is stored in the format:

```
<keyword> = <value>
```

The last line of the header is marked by the keyword `INDICES`, which must appear in this line *without any leading white-spaces*. Apart from that, the order of the keywords is arbitrary. Table 2 lists the keywords and value formats.

Keyword	Description/Value Format	Since	Removed
TYPE	Defines type of output, either FIELD, FLUX, or REFERENCE. For type REFERENCE, only scalar outputs are allowed	5.0	
PROJECT_FILE	Full path to project file	5.0	
CREATED	The date and time the output file was written	5.0	
QUANTITY	A description of the quantity (to be used as axis label in plots)	5.0	
QUANTITY_KW	A keyword that uniquely identifies the physical quantity that is written, needed for special physical computations (list of known quantities is defined within *Delphin/Delphin Post Processing*).	6.0	
GEO_FILE_HASH	Optional keyword, that stores the hash value calculated over a written geometry file. Its intention is to proof the integrity of a geometry file.	6.0	
GEO_FILE	File name of the corresponding geometry file (relative file name only, must not contain path separators).	6.0	
SPACE_TYPE	Spatial averaging/integration method used, possible values are: • SINGLE - one value for each element/side, • MEAN - the values of all selected elements/sides are averaged (using weighted average), • INTEGRAL - the values of all selected elements/sides are integrated in space (by multiplying the values with the volumes/areas)	5.0	
TIME_TYPE	Time integration/averaging method used, possible values are: • NONE - the values are written as they are obtained at the output time point, • MEAN - the values calculated between output time points are averaged in time, • INTEGRAL - the values calculated between output time points are integrated in time	5.0	
VALUE_UNIT	The unit of the values stored in this file (see §3.3.2).	5.0	
TIME_UNIT	The unit of the time points stored in this file (see §3.3.2)	5.0	
START_YEAR	The year that the time points are relative to (see §3.3.2)	6.0	
INDICES	White-space separated list of element/side indices. The numbers refer to the elements/sides that the values in the data section belong to. In case of a SPACE_TYPE of MEAN or INTEGRAL, the numbers are interpreted as indices to elements/sides that the mean/integral value was computed from. For file TYPE of REFERENCE the index is to be interpreted as unique ID of the referenced quantity.	6.0	
ELEMENTS	List of element indices. Same meaning as INDICES. *Deprecated and invalid* in version 6.0 and newer file versions.	5.0	6.0
SIDES	List of side indices. Same meaning as INDICES. *Deprecated and invalid* in version 6.0 and newer file versions.	5.0	6.0

Table 2: Keywords in Data File Headers

3.3.2 Data Section

After the header section the actual output data follows. Since transient simulation results are generated, the values are written grouped by time point. The data is stored in one line per time point, in so-called data blocks, with the following format of an output line:

```
<time point> <first value> <second value> ... <nth value>
```

The number of values depends on both the SPACE_TYPE and INDICES properties of the file headers according to the following specification:

1. When SPACE_TYPE is set to SINGLE, the number of values in the line corresponds to the number of element/side indices of the INDICES property,

2. otherwise, for `SPACE_TYPE` set to `MEAN` or `INTEGRAL`, only a single value is written for each output time point, regardless of the number of elements/sides referenced in the `INDICES` property.

The physical units used for the time point and values are defined in the header properties `VALUE_UNIT` and `TIME_UNIT`. The time points must increase monotonically. The time difference between output time points is arbitrary.

The time points are relative to midnight January 1 of the start year. The start (or reference) year is defined through the `START_YEAR` property. In the interpretation of output data leap days are typically ignored but can be used if needed (e.g. to plot simulation data versus measured data obtained in leap years).

3.4 Binary File Format

3.4.1 Principle Data Types

The file is written using standard data type encoding, listed in Table 3. Strings and arrays are stored in variable-length format. Empty strings and arrays are encoded by a single 32bit unsigned integer value of `0x00000000`. Note, that strings *do not* contain a termination character.

Count	Data Type	Description
	Strings	
1	32bit unsigned int	Number of characters in string -> n
n	8bit char	Characters in string
	Array of Doubles	
1	32bit unsigned int	Number of values in array -> n
n	64bit double	Values in array
	Array of Integers	
1	32bit unsigned int	Number of values in array -> n
n	32bit unsigned int	Values in array

Table 3: Binary Encoding for Standard Data Types

3.4.2 Header Section

The header section begins at file offset 16 bytes, directly after the magic header. It contains the same information as the ASCII version, in the encoding listed in Table 4.

Data Type	Description
32bit unsigned int	File offset that the data section starts
32bit unsigned int	Number of data values in each data block
32bit unsigned int	Type: 0 - FIELD, 1 - FLUX, 2 - REFERENCE
String	Project file name
String	Geometry file name
32bit unsigned int	Geometry file hash
64bit int	Creating time, encoded as time_t (64-bit version)
String	Quantity description
String	Quantity keyword
32bit unsigned int	Space integration type: 0 - SINGLE, 1 - MEAN, 2 - INTEGRAL
32bit unsigned int	Time integration type: 0 - NONE, 1 - MEAN, 2 - INTEGRAL
String	Value unit
String	Time unit
32bit integer	Start/reference year
Array of Integers	The array with indices of elements/sides.

Table 4: Encoding of Header in Data Files

The first two integer values in the header can be used to quickly access the output data content without parsing the header. This is primarily useful when developing automatic data extracting tools that do not require header

information. In such cases, development of header parsing functionality is not necessary and with the file offset stored after the magic header, it is possible to seek directory to the data section in the file.

3.4.3 Data Section

The data section follows the header section. Data is written in blocks per time point. The number of values `n` per block is obtained from the `SPACE_TYPE` and `INDICES` properties, as described in §3.3.2. This number is also encoded in the second integer value of the header section. Each block has the size `8 bytes + n*8 bytes` and is encoded as follows:

Count	Data Type	Description
1	64bit double	Time point
n	64bit double	Values

The constant size of blocks allows direct reading of a data block for a given time index, which enables selective reading of data sections with little memory consumption and high efficiency.

4 Geometry File

4.1 General

The geometry file contains the information that links element/side numbers used in the `INDICES` property of data files to the respective geometrical representation. The *Delphin 6* simulation program currently supports orthogonal, non-rectangular meshes, e.g. L-shaped geometries such as wall corners. The information needed to reconstruct such meshes and display simulation output data is defined through the geometry file. In future versions, the format of the geometry file may be extended to support unstructured grids. Note, that such an extension does not require changes in the file format of the data files.

4.2 File Name Specification

Each geometry file must be named after the following pattern:

`<projectFileName>_<projectHashCode>.<fileExtension>`

`fileExtension` is defined as ".g6a" for plain text and ".g6b" for binary output formats.

`projectFileName` corresponds to the name of the *Delphin 6* project it was generated from. A user-defined name is also possible.

`projectHashCode` is a result of a mathematical function calculated on the content of the simulation project file. As "simulation project file content" this document defines: A *string* containing all relevant data sections influencing geometry, element/node/edge indices and other data resulting in differences of the geometry file. It ensures a unique identification of a geometry file and

1. allows automatic generation of the geometry file name itself,
2. allows unique mapping of a given data file to the correct geometry file, and
3. indicates, if different from other geometry files, that the project file content relevant to output data was altered.

The geometry file begins with a 16 byte magic header and contains a number of data tables describing:

- Materials,
- Grid,
- Elements, and
- Sides.

4.3 ASCII Format

In ASCII format, each table starts with a table definition

```
TABLE <table keyword>
```

and ends with an empty line. An empty line is a line with only white-space characters (preferably no characters at all). In ASCII format the individual tables are identified by the keyword. Therefore, the order of tables is arbitrary. However, it is recommended to write the tables in the order they are described below.

4.3.1 Materials

The first table is the materials table, listing the materials used in the structure. The keyword for the material table is `MATERIALS`. The table contains as many lines as materials. Each line has the format:

```
<material ID> <color ID> <material name>
```

The `materialID` is the unique identifier for the material. The `colorID` is an integer value corresponding to an RGB encoded color (`0xRRGGBB`) that can be used to colorize the construction sketch. An example for such a material table is:

```
TABLE MATERIALS
1 1231665 "Ceramic brick"
2 3312756 "Mineral wool"
3 4322234 "Gypsum board"
```

4.3.2 Grid

The next table is the grid table, containing the widths, heights and thicknesses of the columns, rows, and layers of the grid. The keyword for this table is GRID. The table contains 3 lines. The first line contains the widths of the columns of the grid (x-dimensions). The second line contains the heights of the rows (y-dimensions). The last line contains the thicknesses (z-dimension) or in the case of 3D rotation symmetric geometry a single 0. The dimensions are all in m (meter). For a simple 1D structure, the table could look as follows:

```
TABLE GRID
0.1 0.1 0.1 0.02 0.02 0.01 0.01
1
1
```

Rotation-symmetric geometries are 2D geometries, with an implicit definition of the rotation axis at x=0.

4.3.3 Element Geometry

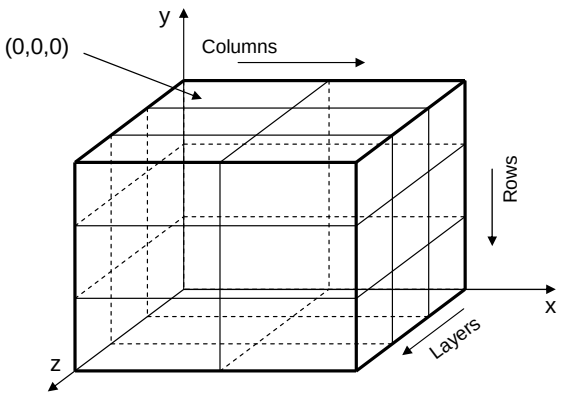
The next table in the geometry file is the element geometry table, with the keyword `ELEMENT_GEOMETRY`. It defines for each element its index number, its coordinates within the global coordinate system and the column, row, and layer indices.

```
<element number> <x-coord> <y-coord> <z-coord> <column> <row> <layer> <material ID>
```

The x, y and z coordinates are hereby the coordinates of the element center points and given in m. The axes are defined as right-hand-side Cartesian coordinate system, with y-axis pointing upward, and z-axis towards the viewer. The column and layer indices start with value 0, growing in the direction of the corresponding axis. The row index grows in negative y-coordinate direction, since element (0, 0, 0) of a given construction is defined as upper left corner in the lowest layer (farthest away from the users view point) in the coordinate system (see illustration below).

The following example shows an element geometry table for a 1D structure.

```
TABLE ELEMENT_GEOMETRY
1 0.05   0.5 0.5 0 0 0 1
2 0.15   0.5 0.5 1 0 0 1
3 0.25   0.5 0.5 2 0 0 1
4 0.31   0.5 0.5 3 0 0 2
5 0.33   0.5 0.5 4 0 0 2
6 0.345 0.5 0.5 5 0 0 3
7 0.355 0.5 0.5 6 0 0 3
```



4.3.4 Sides Geometry

A table containing geometry information for the sides is also included in the geometry file. The keyword for this table is `SIDES_GEOMETRY`. Each line in this table has the following format:

```
<side number> <x-coord> <y-coord> <z-coord> <column> <row> <layer> <direction>
```

The x, y and z coordinates are hereby the coordinates of the middle point of the side given in m. The last value is one of the set [0=`X_DIRECTION`, 1=`Y_DIRECTION`, 2=`Z_DIRECTION`], corresponding to direction (X, Y, or Z) parallel to the orientation of the normal vector of a side. `X_DIRECTION` means that the fluxes are calculated from left to right in X direction. An example sides geometry table for a 1D structure (all vertical sides) is given below:

```
TABLE SIDES_GEOMETRY
1 0.0 0.5 0.5 0 0 0 0
2 0.1 0.5 0.5 1 0 0 0
3 0.2 0.5 0.5 2 0 0 0
4 0.3 0.5 0.5 3 0 0 0
5 0.4 0.5 0.5 4 0 0 0
6 0.5 0.5 0.5 5 0 0 0
```

To save hard disk space the z-information in `ELEMENT_GEOMETRY` and `SIDES_GEOMETRY` are omitted in 2D cases. 2D grids are characterized by a single value of the thicknesses array (z-direction layer count = 1) in the `GRID` table.

4.4 Binary Format

Geometry files in binary format contain the tables described for the ASCII format in the defined order:

1. Materials
2. Grid
3. Element Geometry
4. Sides Geometry

For a description of the content in the tables please refer to the ASCII format section.

4.4.1 Materials

The materials table begins at file offset 16 bytes, directly after the magic header. The material data is encoded as in the following description

Count	Data Type	Description
1	32bit unsigned int	Number of materials -> n
n	MaterialRecordType	Materials in table

with `MaterialRecordType` defined as

Count	Data Type	Description
1	32bit unsigned int	Material ID
1	32bit unsigned int	Color encoded as RGB (0xRRGGBB)
1	String (see Table 3)	Material name

4.4.2 Grid

The grid table is encoded as follows:

Count	Data Type	Description
1	Array of Doubles	Width of columns (Δx values)
1	Array of Doubles	Height of rows (Δy values)
1	Array of Doubles	Thicknesses of layers (Δz values)

The encoding of a variable-length array of doubles is defined in Table 3. In case of a rotation-symmetric geometry, the last vector has length 0.

4.4.3 Element Geometry Table

The encoding of the element geometry data is similar to the material table:

Count	Data Type	Description
1	32bit unsigned int	Number of element definitions -> n
n	ElementRecordType	Element data

with `ElementRecordType` defined as

Count	Data Type	Description
1	32bit unsigned int	Element number
1	64bit double	x-coordinate of element's center point
1	64bit double	y-coordinate of element's center point
1	64bit double	z-coordinate of element's center point
1	32bit unsigned int	Column index
1	32bit unsigned int	Row index
1	32bit unsigned int	Layer index
1	32bit unsigned int	Material ID

In binary encoding, the full geometry information including z-direction information is always written, even if the construction is only 1D or 2D.

4.4.4 Sides Geometry Table

The last table in the geometry file is the sides geometry table, with the following layout:

Count	Data Type	Description
1	32bit unsigned int	Number of side definitions -> n
n	SideRecordType	Side data

with `SideRecordType` defined as

Count	Data Type	Description
1	32bit unsigned int	Side number
1	64bit double	x-coordinate of side's center point
1	64bit double	y-coordinate of side's center point
1	64bit double	z-coordinate of side's center point
1	32bit unsigned int	Column index
1	32bit unsigned int	Row index
1	32bit unsigned int	Layer index
1	32bit unsigned int	Orientation

The orientation is a value [0,1,2] as defined in §4.3.4.

5 Overview of the Programming Interface (API)

The C++ *DataIO* library provides the functionality to read and write output data and geometry files efficiently. This is particularly important, for large data files in ASCII format. The *DataIO* library uses lazy-evaluation techniques for reading of data files. The main API read and write access functions are covered by two classes:

- The class `DataIO` stores all output data from a *Delphin* output file.
- The class `GeoFile` encapsulates the geometry data, simulation results are related to.

This two-parted interface was designed to enable splitted and continuous reading of geometry information and actual simulation results. Since all access operation can be executed on both data types the interface defines the same class routine names for equivalent operations. This results in a small programming interface that can be easily accessed in own software codes.

5.1 Reading Data Files with the DataIO Library

The following C++ source code extract illustrates the use of the library for reading data files:

```
// include main header file for DataIO class
#include <DATAIO_DataIO.h>
...

int main() {
    DATAIO::DataIO dataFile;
    string filename = "..."; // set filename

    // read file
    if (!dataFile.read(filename, errmsg)) {
        // print error message
    }

    // you can now access the header data
    cout << "File type               = " << DATAIO::DataIO::type2string(dataFile.m_type) << endl;
    cout << "Quantity                = " << dataFile.m_quantity << endl;
    // and request a list of stored time points
    cout << "Number of time points   = " << dataFile.m_timepoints.size() << endl;
    cout << "Number of values per tp = " << dataFile.nValues() << endl;

    // access data at time point with time index 10
    cout << "Retrieving data for time index 10, time = "
         << dataFile.m_timepoints[10] << " " << dataFile.m_timeUnit << endl;

    // first argument to data() is the time index, function returns NULL if reading failed
    const double * valdata_t10 = dataFile.data(10, errmsg);
    if (valdata_t10 != NULL) {
        // for example, copy the data into some other memory block, for example std::vector
        vector<double> values(valdata_t10, valdata_t10 + dataFile.nValues());
        // ...
    } else {
        // write error message cerr << errmsg << endl;
    }

    // to access the last data set, use
    const double * lastData = dataFile.data(dataFile.m_timepoints.size() - 1, errmsg);

    ...
}
```

`DATAIO` is the dedicated namespace holding all classes and functions related to reading, writing, and converting simulation output data. The file header `DATAIO_DataIO.h` holds the declaration of the data container class `DataIO`, which encapsulates the data of an output data file.

For binary data files, the file header is read, and afterwards *only* the time points of all data blocks are read and stored in the `DataIO` container. Thus, the memory consumption after the call to `read()` is minimal. Each call to `data()` will result in one data block to be read and stored in memory, unless it had been read already. This operation is very fast. Accessing all data sets of a data file will gradually increase memory consumption until the `DataIO` container uses about the same memory as the data file on hard disk.

Output files stored in ASCII format are read *completely* using efficient binary read functions. The header is parsed and all time points of the data blocks are extracted. However, the actual data values will be stored in strings as read from the ASCII file. Only upon access of a data block via the `data()` function, the corresponding values for each block are parsed. By delaying the time-consuming string parsing to the time of access, the time needed to read even large ASCII files is very short. The `DataIO` container will use about the same memory as the ASCII data file on hard disk after the call to `read()`. When accessing data, the values are parsed from the strings holding the data block, stored in arrays of double values and the original strings are emptied. Thus, each call to `data()` will reduce the memory consumption. Once all data blocks have been accessed, the memory consumption of the `DataIO` container will correspond to the memory requirements of a binary file representation. For typical ASCII value formats (10 digits plus separation white-space characters) the ASCII files will have approximately twice the size of the corresponding binary formats.

In the case that output data files are still being written by running simulation models, repeated calls to the `read()` function will simply update the data container, while preserving already read/parsed data. In the case of binary files only those blocks are accessed, that were added since the last call to `read()`, resulting in very fast update operations.

5.2 Writing Data Files with the DataIO Library

Similar to the `read()` function, the library provides writing functionality. With respect to the interval-based computation of output data, the writing of output data files is split into two functions. The first writes only the file header, whereas the second appends a single data block for a certain time point. This allows continuous writing of files when output data becomes available. Also, simulations that are stopped and continued can simply append new output data to the files.

The details of the programming interface are beyond the scope of this article and can be found in the formal API documentation (see Developers Documentation at [1]).

6 Summary

The formats for output data and geometry files created by the *Delphin* simulation program and used by the *Delphin Post-Processing* software are defined and described. The output data and geometry file format are recommended to be used by transport simulation programs, in order to establish a standard for input files used in advanced analysis and post-processing software.

References

[1] A. Nicolai. Delphin, Numerical simulation tool for the coupled heat, air, moisture and salt transport, Webpage, *http://www.bauklimatik-dresden.de/delphin*, 2011.