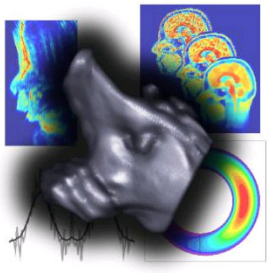


Introduction to HDF5 and F5

Bachelor Thesis

Marcel Ritter, 9817001

March 2007

Supervisor: Dr. Frank Lenzen

Institute of Computer Science

Leopold-Franzens-Universität Innsbruck

Abstract

The thesis describes the scientific data file format F5. Its foundation, the Hierarchical Data Format HDF5, is described and the concepts of F5 are presented. Examples for working with F5 in practice are presented and the compilation and installation processes needed to run them are explained. Finally, data structures and functions for operating on basic tensor field types are implemented in C and C++ and a medical data set is read using F5. The text also should serve as an introductory guide to HDF5 and F5 for a new user.

Table Of Contents

Introduction

Carrying out research has become nearly impossible without intensive use of computers, their computational power today is of great help to many different fields. Two very important fields are data processing and data visualisation, which have many applications, for example in engineering, the medical and life sciences, physics, astronomy and meteorology.

Theoretical models of processes are created while studying the world around us. These mathematical models often become that complicated that analytical investigations are impossible, leaving only numerical methods to study the equations. This limits the kind of data one can work on to numerical data.

Also measuring physical properties with digital instruments and recording the data digitally results in numerical data.

Artificially created or measured data can be processed further, for example with algorithms that, among many other operations, filter, convert, segment, transform, compress or resample the data to highlight certain properties, like using a magnifying glass.

Visualisation of the data then is the subsequent result as it serves as a very powerful method to understand or read the data, particularly when huge numerical data sets have to be analysed. An image created from this data is another representation of this data, but with the great advantage that the data immediately becomes 'readable' for us at a glance. Just imagine the difference between a x-ray film and a spelled out list of grey value levels. Visualisation is also indispensable for communicating the findings or for presenting complicated contexts to an audience.

The software applications that are used by the scientific community share one common need: They need to access the data they should operate on. At present time the applied software often uses non standardised data formats for the external representation. This use of proprietary data formats greatly limits the possibility to interchange data between packages and, therefore, also between research groups or projects. Groups wishing to work together first have to overcome the problem of differing data formats to be able to work on the same data. This is time consuming and sometimes even nearly impossible if the format is not documented, as it is the case for many proprietary formats.

The data has to be represented internally in an application and externally on a physical storage device. Achieving optimal computational performance is the main goal for the internal memory representation. Whereas the external representation should allow easy interchangeability and be as self-explanatory as possible but also allow compact storage and fast read and write access.

From a user's or researcher's point of view the lack of a standard leads to numerous duplication and huge time investments in the case data has to be shared. As working together more and more becomes important, the wish and need for a standardised way to exchange data between software packages and groups becomes very important.

Attempts have been made in the recent time to overcome this problem and to offer a standardised way of storing and accessing scientific data. This thesis presents the Hierarchical Data Format (HDF) library developed by the National Center for Supercomputing Applications [HDF06].

> "HDF5 is a general purpose library and file format for storing scientific data." [An Introduction to HDF5, http://www.hdfgroup.org/HDF5/doc/H5.intro.html, 21.03.2007]

Scientific data, as opposed to information data, is data that carries intrinsic geometric information, is defined in an n-dimensional base space and is given on a specified grid structure. Among the most basic data fields for example, is a scalar field that gives one numeric value for every three dimensional space point, like a temperature field. It could be given on a uniform grid, which means the space points are equally spaced in every direction. Vector fields are another example. They associate a vector to every space point. A velocity field describing flowing fluids in three spacial dimensions is a vector field. A more complex structure is the tensor field, which assigns a multilinear mapping to every point in a grid.

To be able to create the technical implementation one needs further information about properties like the needed numerical resolution, data hierarchy, as well as precise knowledge of the hardware platforms, because numbers are represented differently on different systems. Being able to store meta data is of interest, too. These topics have be addressed by creators of general data formats.

As a user of a format one should not need to know of the technical low level implementation. Ideally, the user only operates on the higher level structures

without having to invest time to address low level implementation details. In the scenario where, for example, data sets should be exchanged it is sufficient if both parties make use of one general, widely available format to store and access their data. This eases and speeds up the application development, helps in sharing data between the groups and leaves the user with more time to concentrate on the scientific work.

To provide such a framework is exactly the purpose of the HDF5 library. “HDF5 can store two primary objects: datasets and groups. A dataset is essentially a multidimensional array of data elements, and a group is a structure for organizing objects in an HDF5 file. Using these two basic objects, one can create and store almost any kind of scientific data structure, such as images, arrays of vectors, and structured and unstructured grids. You can also mix and match them in HDF5 files according to your needs.)” [An Introduction to HDF5, http://www.hdfgroup.org/HDF5/doc/H5.intro.html, 21.03.2007]

The first chapter describes and explains the data format HDF concepts and details. The NCSA implementation of this format, the HDF5 C-library, is then described and some simple and more complex C examples are given.

The second chapter introduces the F5 library which is built on top of HDF5. Fiber bundles are introduced, which are the mathematical basis of the F5 data concept. The data concept is presented and, finally, the F5 library, including installation and use, is described, which includes documentation of important functions and data structures.

The third chapter demonstrates the use of F5 in an example data structure that supports operations on a basic tensor field type. Functions for reading and writing to file and getting meta data or finding minimum and maximum values are implemented. Two versions of this code are presented, one in C, the other in C++, using the generic programming approach.

The thesis ends with a short conclusion.

Hierarchical Data Format (HDF)

The **H**ierarchical **D**ata **F**ormat (Version 5, HDF5) was first released in 2002 by the National Centre of Supercomputing of Illinois.

HDF was designed with respect to application in science and engineering. It should be able to handle as many differently structured data as possible for data storage. This could, for example, be data collected from numerical computations given on a multi-grid, as well as data coming from measurements containing additional meta data.

Scientists use many different programming languages and platforms to develop and run their tools. The developers of HDF5 aimed for maximal compatibility of their library to as many programming languages, platforms and development tools as possible. Only then collaboration and data exchange among the scientific community can be theoretically made possible.

Besides data exchange, data archiving is also an important issue. Especially reading old data, created and stored by somebody who is not available any more is a big problem. If no sufficient documentation is available, this is a time consuming process. After having figured out how the data technically is stored also the interpretation of its meaning has to be figured out, which is either difficult or impossible. A standardised format that allows all necessary meta data to be added inside the data file itself would be a great advance.

While supplying such additional functionality and structure, this must not have major effects on the performance of read and write operations to the data. If using raw binary formats for large data sets is a lot faster, nobody would consider replacing his data operations with the library supplied methods.

The HDF5 group with their development of the HDF5 library tried to address all these problems and supply the community with a ready to use free software implementation of the HDF data format. Easy to install with hooks to many commonly used languages and compilers the library should serve the developers of scientific software tools.

The library comes with a set of command line tools. A tool for examining the content of a HDF5 file and printing the out the content in ASCII for debugging purposes is included. Also tools for getting general information about

the content of a HDF5 file or comparing two files are supplied.

The general development aim of the HDF5 developer group was

"To develop, promote, deploy, and support open and free technologies that facilitate scientific data exchange, access, analysis archiving and discovery." [taken from the slides of a introductory talk by Mike Folk, manager of the HDF Group]

Key Features of the HDF5 library

Data files can be of "**unlimited" size**.

The size is only limited by the size of the available physical storage. Limitations, like a maximum file size limits stemming from the use of 32 bit file pointers, are dealt with inside the HDF5 library. In this case, for example, the HDF5 library can distribute the data between numerous files or disks. Despite of the change of the underlying properties like memory architecture or bus widths the access method to the data never changes. This also holds true if HDF5 files are transported between platforms.

HDF5 data files can have an "**unlimited" size of objects** stored in them.

It was diligently paid attention to **portability and extensibility.** This also ensures that the library can and will be updated and adapted to future systems.

The HDF5 library is available for **several platforms and different languages**. There are distributions for C, C++, Java and Fortran90 and the platforms AIX, UNICOS, FreeBSD, HP-UX, SGI Altix, SGI IRIX, Linux, Mac OS X, OSF1, Solaris and Windows® MVS6.0 and 2003.net are officially supported. This covers to a great extend the platforms that are used by the scientific community.

The **data model** of HDF5 is **simple** and **flexible**. Many other already existing file formats can be mapped to a HDF5 file. An example for this is the HDF5 equivalent to the NetCFD[1] file, developed by the HDF5 team.

HDF5 supports **many data types.** Starting from types like DEC-Alpha-Integer to the IEEE64 bit big endian float type. It is, in addition, also possible to

1 NetCFD is a file format created by NCAR to store atmospheric research and modelling data.

create user defined data types, see *section Concept of Data Organisation, Data-types*.

Meta information like endianness, size and architecture is always stored for a data type, so all information for reading the data is self contained.

The data types together with a grouping and linking mechanism to organise data allows to create an unlimited variety of **complex data types**.

The design of HDF5 includes an I/O layer, called virtual file layer (VFL). Users can write own I/O drivers to access data directly via network, for example.

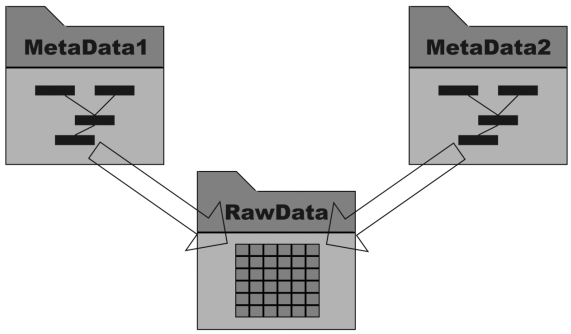


Figure 1: Separation of raw and meta data

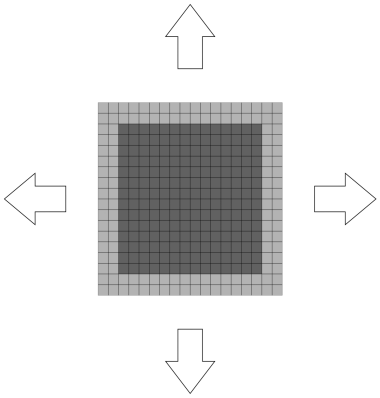


Figure 2: Extensibility of arrays

HDF5 allows for **separation of meta data and raw data.** A HDF5 file can be split to in two files one containing only meta data and the other only raw data. An application still sees these separated files as one logical HDF5 file. The meta data file and the raw file may also be located on different files systems or on different computers. The meta data on one system might be optimized for many small I/O accesses and the raw data on different system could optimized for sparse I/O accesses but transporting much data, as a tape based storage device would do. Raw data files can also be shared between different meta data files to reduce disc space usage, see *figure 1* for an example.

HDF5 is able to **compress data** for storage. It uses the zlib library [ZLIB05] for this purpose, but the user can also provide his own compression or filtering methods. These transformations can be inserted into the I/O operations chain.

Data types can be **converted** during HDF5 I/O operations. Data types are organised in several classes and can be converted inside the according class during I/O. For example, types of the class float can be converted to different

float types, like the conversion of a 4 byte little endian float to a 6 byte big-endian float.

The data array type can be **extended,** if necessary, in all possible directions along all dimensions, see *figure 2*. The size of a data set must not be known at creation time and the data has not to be written at once but can be written in smaller blocks.

HDF5 supports **sub selection** and **spatial data transformations** during I/O operations. This is important when working with large data sets. Complex sub selections of data sets can be achieved by union operations.

The transferred selection can be transformed to a different shape. *Figure 3* shows an example, where a sub selection of a 2-dim array is transformed into a 1-dim one during I/O operation. The number of involved elements must not change, of course.

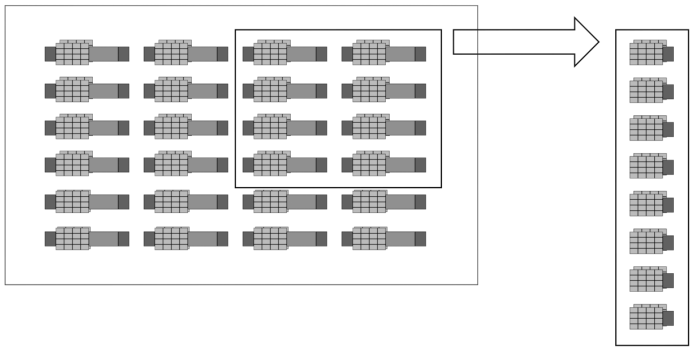


Figure 3: Sub selection and spatial data transformation

Additionally, an element of a data set may consist of several other data objects. Such an element is then called a compound. *Figure 4a* illustrates a compound consisting of two floats, an array of floats and a second float . This compound is an element of the left array in *figure 3*.

Compound components can also be transformed to different compounds during I/O. *Figure 4b* shows a compound after a possible I/O transformation.

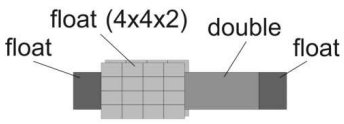


Figure 4a: Source Compound

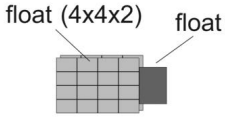


Figure 4b: Target Compound

Concept of Data Organisation

A HDF5 data model is a directed graph structure with one root node. It is created from the following components: g*roups, data-sets* and *links*. *Groups* and *data-sets* are nodes in the graph. *Links* and "contained in" relations are edges. *Files*, *attributes*, *data-types* and *data spaces* are additional components that are not related to the graph structure.

- ***File:***

  A file serves as the container for all other hdf5-objects that build up the complete data structure. The file holds meta data information and the root node, a group object, named the root group "/".

  The physical file on disk typically has the extension ".h5".

- ***Group:***

  A group is a hdf5-named-object. It consists of a group header containing the group name and a list of attributes and a list of other hdf5-named-objects, the elements of this group, in a symbol table.

  A group is similar to a directory entry in a UNIX or a folder in a Windows® file system, but may contain cycles. A graph as shown in *figure 5* is allowed in HDF5 (with A, B, C being groups) but not in a file system (with A, B, C being directories respectively).

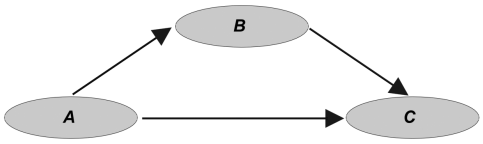


Figure 5: Graph cycle

- ***Data Set:***

  A data-set is also a hdf5-named-object. It consists of a header and a data array. The header contains the information necessary to read the array and meta data. This includes the name of the data in alphanumeric ASCII characters, a list of attributes, the data-type of all the data array elements, the data-space of the array (dimensions and sizes) and the layout the data is stored in.

The storage layout describes the ordering of the array elements in memory or in the file.

A contiguous layout, for example, arranges one element besides another, such that one continuous block of data is created.

Other possible layouts comprise the compact and chunked layout, see *chapter "II Using HDF5-The Specifics" section "Datasets" in* [H5U06].

The data array itself is a rectangular array of simple or compound data-types up to a dimensionality (rank) of 32. This maximum dimensionality is defined in the HDF5 library. The HDF5 file format theoretically supports a rank up to the maximum integer value.

- ***Links***

Links are used to share hdf5-named-objects between hdf5-named-objects. There are hard and soft links.

A hard link is an element in a group and must point to an existing hdf5-named-object. "X is contained in a group" is equivalent to "The group contains a hard link to X". Group membership is also implemented via hard links. A hdf5-named-object remembers the number of hard links pointing to it in a reference counter.

A soft link is an element in a group that holds a pathname to a hdf5-named-object, which may or may not exist. No reference counter is stored for a soft link.

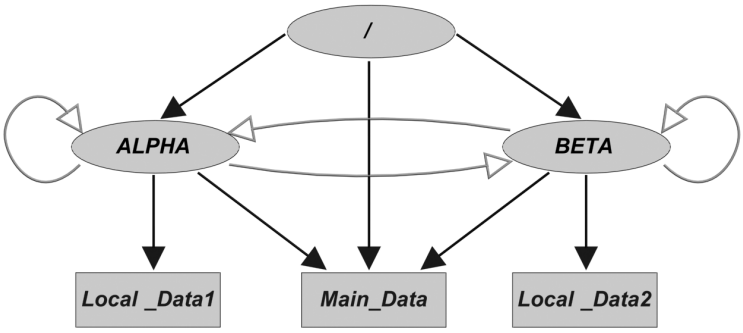


Figure 6: HDF5 structure example.
The ellipsoids illustrate groups, the rectangles illustrate data-sets and the arrows links. A black arrow represents a hard link and a grey arrow a soft link.

Figure 6 shows a possible structure in a HDF5 file. The root group "/" holds the two groups "/ALPHA" and "/BETA" and a data-set

"/Main_Data". "/Main_Data" is shared between the two groups using hard links. Each group also contains its own data-set "/Local_Data" and soft links to itself and the other group.

- ***Data Type:***

A data-type is a description of a data element, which forms the building blocks of arrays. A data-type can be an atomic data-type, a native data-type, a compound data-type or a named data-type.

Atomic data types are data-types that cannot be split further. HDF5 supports seven different classes of atomic types: *integer*, *float*, *string*, *bitfield*, *time*, *opaque* and *reference*. For each of these classes a different set of properties is stored.

For example, an atomic type of class integer has the properties *size* in bytes, *precision* in bits, *offset* in bits, *pad*, *byte order* and s*igned/unsigned*. These properties can be read or modified by functions of the data-type interface section.

The list of properties for a float type is *size* in bytes, *precision* in bits, *offset* in bits, *pad*, *byte order*, *sign position*, *exponent position*, *exponent size* in bits, *exponent sign*, *exponent bias*, *mantissa position*, *mantissa size* in bits, *mantissa sign*, *mantissa normalization* and *internal padding*.

Properties of other atomic data-types can be found in *chapter "II. Using HDF5-The Specifics" section "Datatypes" table 1 in* [H5U06].

Native data-types are atomic data-types, which are platform independent. These native types should be used by the programmer, as HDF5 automatically chooses the best matching atomic data-type for file storage according to architecture and platform.

For example, a C *double* value should be represented by the native data-type *H5T_NATIVE_DOUBLE* in the application. When this type is written to a file on IA32[1] platform the HDF5 file contains the data-type description *H5T_IEEE_F64LE*, which stands for IEEE float 64 bit little endian. When the same native type is written to a file on a SPARC platform the file would contain a *H5T_IEEE_F64BE*, as this platform uses big endian floats.

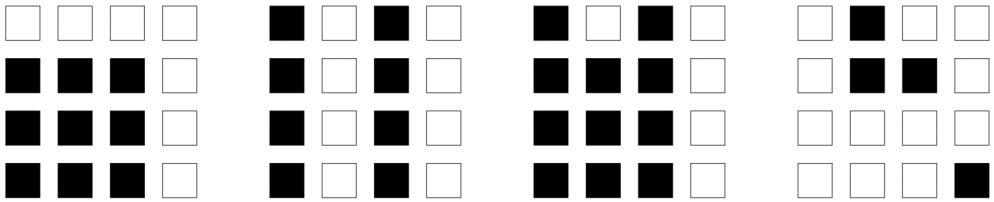
Compound data-types are collections of data-types. An element of a compound data-type can be an atomic data-type or compound data-type. A compound data-type is similar to a *struct* in the C programming language.

1 Intel Architecture, 32-Bit

Named data-types are hdf5-named-objects. They contain data-type information independent of data-sets. They stand on their own and can then be referred to by data-sets and attributes. A named data-type can be shared among different data-sets or attributes.

- ***Data Space***

One data-space object is required in a data-set or in a attribute. In a data-set the data-space contains its rank, which is the number of dimensions of the data array, the number of elements in each dimension and the maximum number of elements in each dimension. The number of dimensions can be fixed or unlimited. A fixed data-set cannot be extended later on.



(a) ***(b)*** ***(c)*** ***(d)***

Figure 7: *Data space sub selection examples*

Besides defining the data-space of a data-set or attribute data-spaces are also used during I/O operations, where they specify the elements involved during the operation. Data-spaces can specify sub selections on data-sets. Selections can be *(a)* contiguous n-dimensional, *(b)* non contiguous equally spaced, *(c)* unions of selections and *(d)* a list of data-set elements, see *figure 7*.

For example, a transforming I/O operation takes two data-spaces as arguments, the source and the sink data-space for the involved data-sets.

- ***Attribute***

Attributes can be associated to groups, data-set and named data-types. Attributes provide a way to store additional user defined meta data to an object. They consist of a name, a data-space and a data-type. Attributes should be used to store small data to the associated objects. No sharing, compression, chunking or sub selection is possible with attributes. They aredirectly written into the header of the associated object.

The HDF5 Library

The HDF5 library predefines basic data types and needed functions and comes with binary tools for debugging and manipulation of hdf5 files. It s available for C, C++ and FORTAN90 and JAVA.

The predefined functions are organised in the 12 sections:

Library Functions,	**A**ttribute Interface,
Dataset Interface,	**E**rror Interface,
File Interface,	**G**roup Interface,
Identifier Interface,	**P**roperty List Interface[1],
Reference Interface,	Data-**S**pace Interface,
Data-**T**ype Interface,	Filters and Compression Interfaces

Function naming convention is: "H5" + a letter of the section the function belongs (printed in bold above)[2].

More than 100 different basic data-types for different platforms, memory layouts, bit depths, etc. are predefined. See *section Concept of Data Organisation, Data-Types* for the properties of a data-type. A complete list of all predefined data-types can be found in the HDF5 reference, see [H5R06].

New types also can be created using the functions of the Data-Type Interface, which allow to set and get properties of a data-type.

Working with HDF5

The library can be obtained from the HDF group homepage [HDF06] in the download section. There is a source distribution as well as multiple binary distributions. The library has two dependencies to external libraries when installed with all features. It uses SZIP [SZIP07] and the ZLIB [ZLIB05] libraries for data compression.

1 Property lists are used to pass additional parameters to functions.
2 Exceptions are "H5" for library functions and "H5Z" for filter and compression functions.

Installation HDF5 for cygwin and Linux

This describes the installation procedure of the HDF5 source library inside the MS-Windows® Cygwin [CYG07] environment. Cygwin is a portability library and frame work that adds full POSIX standard compliance to MS-Windows®. The HDF5 source installation is done analogously in Linux.

The described process follows mainly the instructions given in *http://www.hdfgroup.org/windows/INSTALL_Cygwin.txt*.

The source distribution of hdf5 can be downloaded at the homepage http://www.hdfgroup.com/HDF5/release/obtain5.html. For example the newest version 1.8.0. The file hdf5-1.8.0-alpha5.tar.gz can be uncompressed and unpacked using the command, for example into the /tmp directory.

```
tar xfz hdf5-1.8.0.tar.gz
```

The LZIP library should have been installed by the installer of cygwin for the standard installation. If the lzip library is missing the the setup of cygwin should be used to add it, or it can be obtained in the hdf5 download section.

Next step is to get and compile the szip library [SZIP07]. A binary distribution (szip_cygwin_encoder.zip) for cygwin can be found at [http:// www. .hdfgroup.com/HDF5/release//obtain5.html, 2007].

If the hdf5 library should be installed for a local user, place all libraries in sub directories of the users's home directory. If it should be installed globally then the libraries should be put under the /opt directory.

Now the installation of hdf5 can be configured, for example by the following command executed in /tmp/hdf5.

```
./configure --with-szlib=~/hdf5/szip --prefix=~/hdf5/hdf5 –enable-cxx
```

This installs the library into the user's home directory specified by the flag *--prefix*. The flag *-enable-cxx* enables support for c++. By default c is supported only.

After the installation was configured the library can be compiled by executing *make* in the /tmp/hdf5 directory and the compilation can be tested.

```
./make
./make check
```

If the self test procedures were successful one can proceed to the final installation process.

```
./make install
```

This copies the libraries, binaries, header files and documentation to the directory given under the –prefix option. The directory containing the hdf5 binary files should be added to your *PATH* environment. In case of a local user installation the following line should be added to the *.bashrc* file in the home directory.

```
export PATH=$PATH:~/hdf5/hdf5/bin
```

For a global installation add the according line at the end of your system wide */etc/profile*.

To test the library compile one of the example files using the compile command script of hdf5, see *section Using the Binary Tools*.

```
h5cc -O3 -o h5_write /tmp/hdf5/hdf5-1.8.0-alpha5/examples/h5_write.c
```

After execution of *h5_write* the hdf5 file *SDS.h5* should have been created in the local directory. Its content can then be examined by *h5ls*, see *section H5ls*.

```
h5ls -rv SHS.h5
```

If this works successfully then the hdf5 library is ready to use.

Install hdf5 for MS Visual Studio®

This describes how to install the HDF5 library for Microsoft's Software Development platform Microsoft Visual Studio® 2005 [MVS05]

The hdf5 binary distribution for the windows® compilers MSVC++ 6.0 and .NET 2003 can be found at *ftp://ftp.hdfgroup.org/HDF5/current/bin/windows/*, it is also compatible with the newer Visual Studio® 2005 [MVS05].

Get the binary distributions of szip [SZIP] and zlib [ZLIB05] for windows following the instructions here:
http://www.hdfgroup.com/HDF5/release/obtain5.html.

The archives contain header files, static and shared libraries. The archives

can be extracted, for example, using the tool winrar [RAR07] and should be placed into folders like *c:\hdf5\5-165-win-net*, *c:\hdf5\szip20-win-xp-enc* and *c:\hdf5\zlib122-windows*. The folder *c:\hdf5\5-165-win-net\bi*n can then be added to the *PATH* environment variable, by executing the sequence:

My Computer > Properties > Advanced > Environment Variables

The delimiter ";" separates between several paths. Commands like *h5ls* can then be executed from any directory in the windows command prompt.

A new project in Visual Studio® can be created and configured. The search paths that contain the header files have to be set. In our example the three directories *c:\hdf5\5-165-win-net\include*, *c:\hdf5\szip20-win-xp-enc\include* and *c:\hdf5\zlib122-windows\include* have to be added to the include path list of Visual Studio®, see *figure 8*.

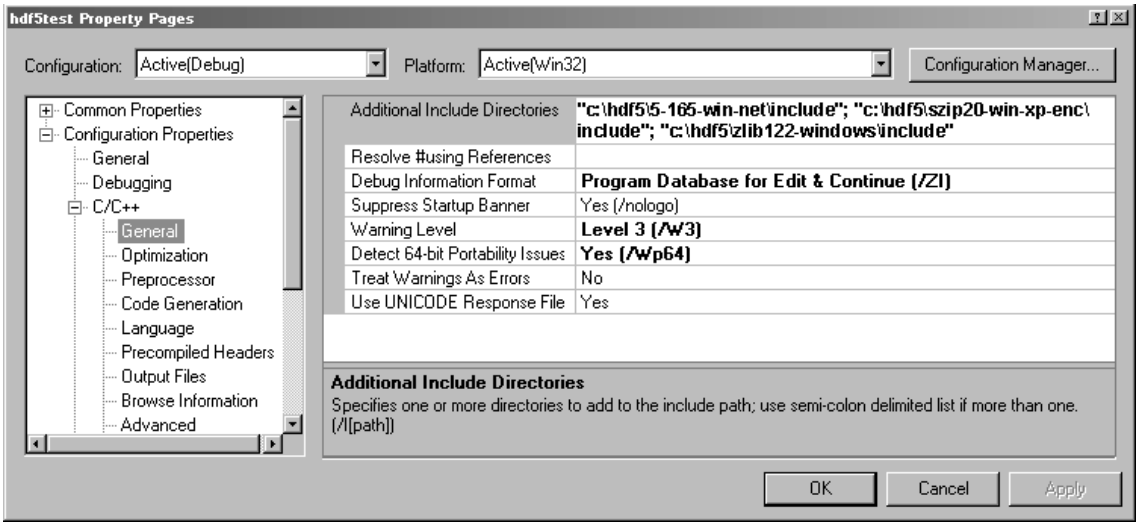


Figure 8: *Configure include paths*

The same has to be done for shared library search paths. The library paths *c:\hdf5\5-165-win-net\lib*, *c:\hdf5\szip20-win-xp-enc\lib* and c*:\hdf5\zlib122-windows\lib* must be added to the "Additional Library Directories", according to *figure 9*.

The libraries must then be specification as additional dependencies as illustrated in *figure 10*.

If the binary was dynamically linked, the shared libraries must be available to the application at runtime. When executing your application windows® will typically search for a dll in the following directories:

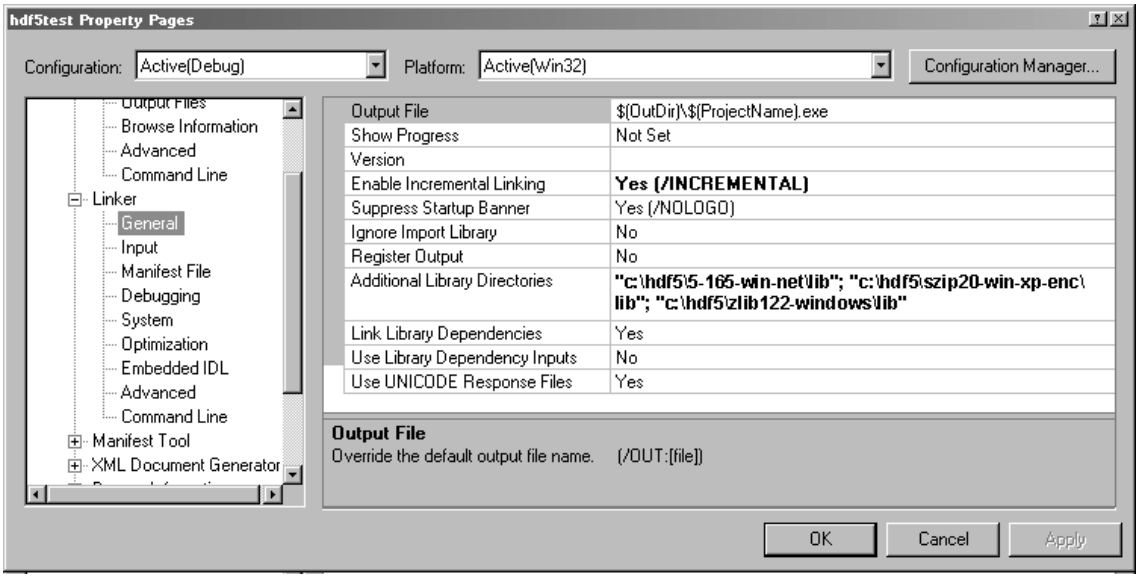


Figure 9: *Configure library paths*

“

1. The directory from which the application loaded.
2. The system directory. Use the GetSystemDirectory function to get the path of this directory.
3. The 16-bit system directory. There is no function that obtains the path of this directory, but it is searched.
4. The Windows directory. Use the GetWindowsDirectory function to get the path of this directory.
5. The current directory.
6. The directories that are listed in the PATH environment variable. Note that this does not include the per-application path specified by the App Paths registry key.

“ [http://msdn2.microsoft.com/en-us/library/ms682586.aspx]

According to point 6 the folders containing the dlls should be added to your PATH environment variable. They then are available to all applications in your system, whether it is a stand alone console application or a shared library dll by itself.

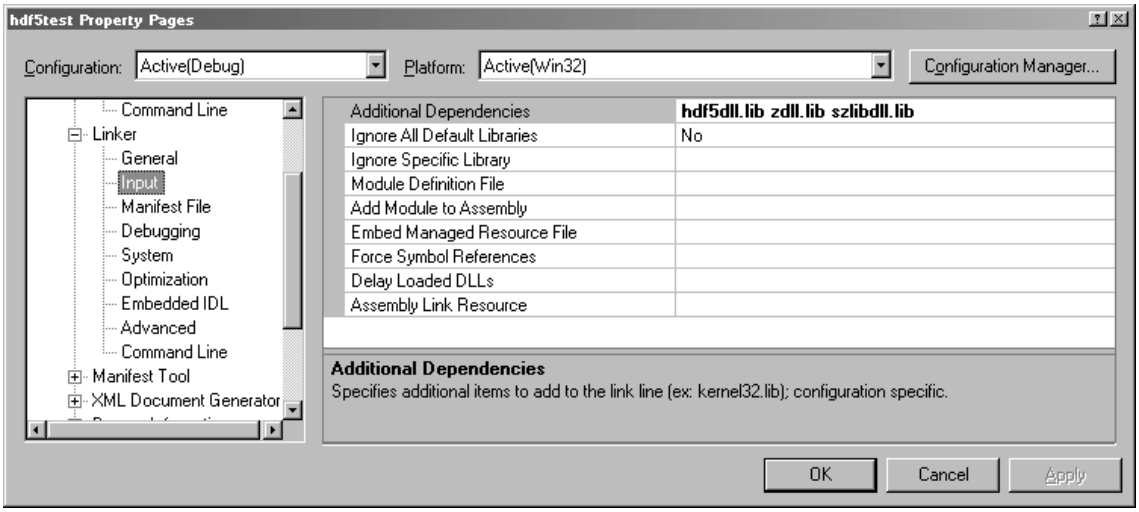


Figure 10: *Configure library paths*

Using the Binary Tools

HDF5 provides many command line tools for viewing, analysing, manipulating and testing existing HDF5 files. These commands are placed in the hdf5/bin directory. A complete documentation of the tools can be found at [H5R06].

Very useful tools for developing command line programs in Linux or within cygwin/windows are *h5cc* and *h5*c++. If these command are used for compilation and call the compiler specified during hdf5 installation. They add all hdf5 relevant compiler options and library and include paths.

The two commands *h5ls* and *h5dump* produce human readable output and are a valuable debugging tool during development.

The command H5ls

H5ls lists the content of an HDF5 file in different levels of detail. The command without additional parameters prints all hdf5 objects inside the root group. To see all objects and their types invoke *h5ls -r* (-r, for a recursive), which results in an ASCII output of the following format:

```
/GroupName/GroupName/.../GroupName/ObjectName ObjectType
```

A partial example of the output of h5ls with recursive call option is listed below:

```
/T=0/Testpat_BD_TumorNeu/Points/StandardCartesianChart3D Group
```

```
/T=0/Testpat_BD_TumorNeu/Points/StandardCartesianChart3D/DTI_tensor Dataset
                                                              {56, 128, 128}
/T=0/Testpat_BD_TumorNeu/Points/StandardCartesianChart3D/Positions Group
...
/TableOfContents/TypeInfo Type
```

The level of detail can be increased by adding the command line option *-v*, which stands for verbose. This also shows the elements and data types of all objects, but no data values of the data-sets. The complete description of all options is shown with *h5ls --help*.

H5dump

H5dump dumps the entire content of a HDF5 file as ASCII[1] text to the standard output stream, which normally is directed to the terminal. When a lot of text is displayed the standard output stream should be piped into a file, that can then be displayed and browsed with a text viewer or editor.

```
h5dump test.h5>test.txt (UNIX)
```

This command redirects the standard output stream of h5dump into a new file *test.txt*, which can then be displayed using less:

```
less test.txt (The less command is available on most UNIX systems)
```

Files of large data sets can become really big when doing this. Here is an excerpt of how an ASCII dump looks like:

```
...
  GROUP "T=0.1" {
    ATTRIBUTE "Time" {
      DATATYPE  H5T_IEEE_F64LE
      DATASPACE  SCALAR
      DATA {
      (0): 0.1
      }
...
```

This example shows the beginning of a group block and the content of an attribute. The attribute “Time” contains one scalar float, which holds the number 0.1 as data value.

Objects are defined through keywords and limited by curled brackets. The

1 *American Standard Code for Information Interchange*

full grammar is defined in an data description language, see [DDL07], which is given in BNF[1]-notation, see [BRO98]. *H5dump* transcribes to complete content of the h5 file. To get a more compact view use *h5ls* instead.

1 Backus Naur Form

Writing, Reading and Debugging Data using HDF5

Storing a simple data-set

This example demonstrates how to store a one dimensional array of type *double* to a HDF5 file using C. All steps to arrive at a working example are demonstrated and necessary information is provided where it is necessary.

Programs that want to access hdf5 functions need to include the hdf5.h header file. HDF5 identifies objects through an identifier, which is of type *hid_t*. Each access to a file, a group, a data-set or a data-type will involve such a type.

To create a data-set several hdf5 objects are necessary. Several *hid_t* objects need to be declared at the beginning of the main scope, as well as a *hsize_t datadim[1]*, used for setting the data-set dimensions, and a *herr_t status* for error checking. The one dimensional array of *double* values will be accessed via the pointer *ddata*.

```
hid_t file, dataset, datatype, dataspace;
hsize_t datadim[1];
herr_t status;
double*ddata;
```

Functions that return *hid_t* return a value smaller zero in case of an error. Functions that return *herr_t* also return a value smaller zero in case of an error. If an error occurs hdf5 puts an error message onto its error stack. This stack can be printed into a stream.

```
H5Eprint(stdout)
```

H5Eprint() returns a *herr_t* and prints the hdf5 error stack into a stream specified by a pointer *FILE**.

After space for the array *ddata* was allocated and after it was filled with numbers a HDF5 file is created.

```
file = H5Fcreate("write_00.h5", H5F_ACC_TRUNC, H5P_DEFAULT, H5P_DEFAULT);
```

H5Fcreate() returns a valid *hid_*t if it successfully creates a file. The first parameter is the path and file name, the second a flag that defines the behaviour, when a file with the same name is already existent. With the flag *H5F_ACC_TRUNC H5Fcreate()* will overwrite all data in an already existing

file. The third and fourth parameters are a *hid_t* to a property list specifying creation and access modes, which are here set to default by *H5P_DEFAULT*.

Next, data-space and data-type of the data-set are prepared. Our data-set is an array of *double* values with n elements.

```
datadim[0] = n;
dataspace = H5Screate_simple(1, datadim, NULL);
datatype = H5Tcopy(H5T_NATIVE_DOUBLE);
```

H5Screate_simple() returns a *hid_*t for a created data-space. It takes the rank of an array, an array of the sizes of each dimension and an array of the maximum sizes for each dimension. The third parameter may be *NULL*, then the maximum size is equal to the size specified with the second parameter.

H5Tcopy() returns a *hid_t* for a data-type of a type specified by the parameter. The created data-space and data-type can now be used to create an empty data-set.

```
dataset = H5Dcreate(file,"array_of_doubles", datatype, dataspace, H5P_DEFAULT);
```

With *H5Dcreate()* a data-set is created and a *hid_t* is returned. The first parameter *hid_t* describes the location in the file. It can be the file id or a group id. The second parameter, of type *const char**, is the name of the created data-set. The third and fourth parameter specify the data-type and data-space of the data-set. The last parameter can be used to pass a property list, which allows set more options, like storage layout, compression or external storage.

Next, the empty data-set can be filled and written to the storage device.

```
status = H5Dwrite(dataset, H5T_NATIVE_DOUBLE, H5S_ALL, H5S_ALL,
H5P_DEFAULT, doubledata);
```

H5Dwrite() writes data into a given dataset and returns an error status *herr_t*. The first parameter *hid_t* specifies the data-set that will be filled with data. The second parameter *hid_t* passes the memory type of the data elements of the set. The third and fourth *hid_it* are data-spaces that can be used to write only parts of the data-sets. If both are set to *H5S_ALL* the complete data-set is written. Again, additional options can be passed by property lists, here using the fifth parameter. Finally, the last parameter specifies a pointer of type *void** to the array holding the data.

The function calls listed above opened a data-space, a data-type, a data-set

and a file. These objects should be closed by the according functions.

```
H5Sclose(dataspace);
H5Tclose(datatype);
H5Dclose(dataset);
H5Fclose(file);
```

The c-file can be compiled, for example, by using the h5cc command.

```
h5cc -o write_00 -O3 write_00.c
```

```
$ h5ls -rv write_00.h5
Opened "write_00.h5" with sec2 driver.
/array_of_doubles        Dataset {5/5}
   Location:  1:792
   Links:     1
   Modified:  2007-03-18 16:30:17 WEST
   Storage:   40 logical bytes, 40 allocated
              bytes, 100.00% utilization
   Type:      native double
```

```
$ h5dump write_00.h5
HDF5 "write_00.h5" {
GROUP "/" {
  DATASET "random doubles" {
    DATATYPE  H5T_IEEE_F64LE
    DATASPACE  SIMPLE { ( 5 ) /
                                ( 5 ) }
    DATA {
    (0): 0, 6.90001, 5.05418,
    (3): 5.91491, 5.54785
    }
  }
}
}
```

Executing write_00 creates the file write_00.h5 in the local directory, which can be examined by *h5ls* and *h5dump*:

Reading a simple dataset

The lines of code provide an example to read the data-set created in the first example. Since similar hdf5 objects are necessary as above the declaration block is omitted here.

```
file = H5Fopen("write_00.h5", H5F_ACC_RDONLY, H5P_DEFAULT);
```

H5Fopen() returns a *hid_t* of a opened file. The first parameter specifies the file system path and the file name as a *const char**. The access mode is described by the second parameter. This includes read only, write only or read and write. Additional options that can be passed via a property list are buffered I/O,

unbuffered I/O or parallel file I/O using MPI.

Hdf5 objects inside the file can now be opened.

```
dataset = H5Dopen(file, "random doubles");
```

H5Dopen() returns a *hid_t* of a data-set. It opens the data-set specified by a location (*hid_t)* and the name of a data-set. The location may be a file or a group id. The name of a data-set can, for example, be retrieved by using *h5ls*. Also, other mechanisms to open a data-set without knowing its specific name exist, see *section Reading Groups, Comments, Attributes and Links*.

Size and data type of a data-set have to be extracted before it can be read.

```
datatype = H5Dget_type(dataset);
datasize = H5Tget_size(datatype);
```

With *H5Dget_type()* we get a *hid_t* of a data-type of a specified data-set. *H5Tget_size()* returns the *size_t* of a data-type in bytes.

Also the dimensions (rank) of the hdf5 array have to be determined, what is done via a data-space.

```
dataspace = H5Dget_space(dataset);
rank = H5Sget_simple_extent_ndims(dataspace);
status  = H5Sget_simple_extent_dims(dataspace, datadim, NULL);
```

H5Dget_space() returns a *hid_t* of a data-space of a data-set. From the data-space the rank and the number of elements in each dimension can the obtained using *H5Sget_simple_extent_ndims()*, which returns the rank as an *int*, and *H5Sget_simple_extent_dims()*, which returns the number of elements and the maximum number of elements via two hsize_t pointers. In our case the maximum number is not needed. So, the last parameter is set to NULL.

The data collected above can be used to allocate memory for the data array.

```
doubledata = (double*)malloc( datadim[0] * datasize );
```

And data is read from the data-set.

```
H5Dread(dataset, datatype, H5S_ALL, H5S_ALL, H5P_DEFAULT, doubledata );
```

H5Dread() returns a *herr_t* and reads a data block from an opened data-set into an array. The parameters specify the data-set, the data-type, the memory

data-space, the file data-space, a property list and the array (*void**). The two data-spaces can be used to read sub selections of the data-set.

```
H5Tclose(datatype);
H5Dclose(dataset);
H5Sclose(dataspace);
H5Fclose(file);
```

Finally, the opened HDF5 objects have to be closed.

Creating Groups, Attributes, Comments and Links

In this example a hdf5 structure as shown earlier in *figure 6* is created. Besides groups, data-sets and links some additional attributes and comments are created and assigned to some of the hdf5-named-objects.

First the data-set called “Main_Data” contained in the root group of the hdf5 file is created and a comment is associated.

```
dataset = H5Dcreate(file, "Main_Data", datatype, dataspace, H5P_DEFAULT);
H5Gset_comment(dataset, ".", "Measured by Arno Arnold");
```

H5Gset_comment() returns a *herr_t*. The first parameter *hid_t* specifies the location and the second parameter the name of the object that should get the comment. The last two parameters must be given as *const char** string.

The name is a path in the hdf5 file. A path has a similar syntax to file system paths in UNIX, for example: “/groupA/datasetA”. A path is absolute, when starting with “/” and relative otherwise. If its relative then it is relative to the location specified by the first parameter (a *hid_t*). The “.” is a relative path to the current location.

The following two H5Gset_commt() calls are equivalent to the one above.

```
H5Gset_comment(file, "Main_Data/", "Measured by Arno Arnold");
H5Gset_comment(dataset, "/Main_Data/", "Measured by Arno Arnold");
```

Next an attribute is created and associated with the data-set “/Main_Data”.

```
datadim[0] = 1;
dataspaceA = H5Screate_simple(1, datadim, NULL);
datatypeA = H5Tcopy(H5T_NATIVE_DOUBLE);
```

Here data-space and data-type have been prepared for the attribute.

```
attrib = H5Acreate(dataset, "offset", datatypeA, dataspaceA, H5P_DEFAULT );
H5Awrite(attrib, datatypeA, &attr_val );
```

H5Acreate() returns a *hid_t* for an attribute. The parameters are location, name, data-type, data-space, and a property list. After the attribute was created a number can be stored inside.

H5Awrite() returns a *herr_t* and takes, the id of the attribute, its data-type and a pointer to the attribute value.

```
H5Sclose(dataspaceA);
H5Tclose(datatypeA);
H5Aclose(attrib);
```

Again involved object have to be closed.

Next, the two groups "/ALPHA" and "/BETA" are created.

```
group = H5Gcreate(file, "/ALPHA", 0);
```

H5Gcreate() returns a *hid_t* of the created group and takes a location, a path and a *size_t*. This size is used by hdf5 to allocate memory in bytes for names that will be stored in the group header. This parameter is optional and can be set to zero, since dynamic resizing is supported. It is faster to pre define the sizes, though.

After a group was created it remains open and members of the group can be created. Assume that a data-set called "Local_Data" is stored inside the group. Afterwards the group must be closed:

```
H5Gclose(group);
```

Creation of the second group "/BETA" and its local data-set is done analogously.

The links in the hdf55 file have to be created:

```
H5Glink(file, H5G_LINK_HARD, "/Main_Data", "/BETA/Main_Data");
H5Glink(file, H5G_LINK_HARD, "/Main_Data", "/ALPHA/Main_Data");
```

HSGlink() returns *herr_t* and creates a link at a given position and path. The second parameter specifies if the link is hard or soft. The link points to the path specified by the third parameter. The fourth parameter is the path of the link itself.

In the example above, links inside the groups "/BETA" and "/ALPHA"

called “Main_Data” point to the data-set “Main_Data” in the root group. Thus, Main_Data is shared between both groups because the hard link behaves like a real data-set, when being accessed.

```
H5Glink(file, H5G_LINK_SOFT, "/ALPHA", "/BETA/next");
H5Glink(file, H5G_LINK_SOFT, "/BETA", "/BETA/previous");
H5Glink(file, H5G_LINK_SOFT, "/ALPHA", "/ALPHA/next");
H5Glink(file, H5G_LINK_SOFT, "/BETA", "/ALPHA/previous");
```

The same function is used to create soft links, which point to the groups itself and the other group. They can now be used to access the so called “next” and “previous” object of this data structure.

Examining the written file by *h5ls -r* gives a overview over the structure, compare *figure 6*:

```
$ h5ls -r write_03.h5
/ALPHA                   Group
/ALPHA/Local_Data        Dataset {3}
/ALPHA/Main_Data         Dataset {3}
/ALPHA/next              -> /ALPHA
/ALPHA/previous          -> /BETA
/BETA                    Group
/BETA/Local_Data         Dataset {3}
/BETA/Main_Data          Dataset, same as /ALPHA/Main_Data
/BETA/next               -> /ALPHA
/BETA/previous           -> /BETA
/Main_Data               Dataset, same as /ALPHA/Main_Data
```

h5dump shows the complete structure:

```
$ h5dump write_03.h5

HDF5 "write_03.h5" {
GROUP "/" {
  GROUP "ALPHA" {
    COMMENT "Ordinary Node"
    DATASET "Local_Data" {
      DATATYPE  H5T_IEEE_F64LE
      DATASPACE  SIMPLE { ( 5 ) /
                                ( 5  ) }
      DATA {
      (0):       1, 62.4102, 36.6531,
                 47.8159, 42.8743
      }
    }
    DATASET "Main_Data" {
    COMMENT "Measured by Arno
Arnold"
      DATATYPE  H5T_IEEE_F64LE
      DATASPACE  SIMPLE { ( 5 ) /
                                ( 5 ) }
      DATA {
      (0): 1, 7.90001, 6.05418,
           6.91491, 6.54785
      }
      ATTRIBUTE "offset" {
        DATATYPE  H5T_IEEE_F64LE
        DATASPACE  SIMPLE { ( 1 ) /
                                  ( 1 ) }
        DATA {
        (0): 3.6
        }
      }
    }
    SOFTLINK "next" {
      LINKTARGET "/ALPHA"
    }
    SOFTLINK "previous" {
      LINKTARGET "/BETA"
    }  LINKTARGET "/ALPHA"
  }  }
  GROUP "BETA" {revious" {
    COMMENT "Ordinary Node"
    DATASET "Local_Data" {
      DATATYPE  H5T_IEEE_F64LE
      DATASPACE  SIMPLE { ( 5 ) /
                                  ( 5 ) }
      DATA {"/ALPHA/Main_Data"
      (0): 2, 2, 2, 2, 2
      }
    }
    DATASET "Main_Data" {
      HARDLINK "/ALPHA/Main_Data"
    }
    SOFTLINK "next" {
      LINKTARGET "/ALPHA"
    }
    SOFTLINK "previous" {
      LINKTARGET "/BETA"
    }
  }
  DATASET "Main_Data" {
    HARDLINK "/ALPHA/Main_Data"
  }
}
}
```

Reading Groups, Comments, Attributes and Links

This example shows how to open groups, get comments, get attributes and how to iterate over hdf5-named-objects in the HDF5 file.

```
group = H5Gopen(file, "/ALPHA" );
```

H5Gopen() returns a *hid_t* of the open group. Many operation do not need a group to be open. But some functions only take a location *hid_t* of a opened hdf5 object and so the it has to be opened in before, for example:

```
H5Gget_num_objs(group, &objnum);
```

H5Gget_num_objs() returns *herr_t.* It returns the number of hdf5-named-objects contained in the group into a pointer *hsize_t** (second parameter).

Names and data-types of group elements can now be extracted using an index.

```
for(j = 0; j < objnum; j++)
{
        H5Gget_objname_by_idx(group, j, buffer, 512 );
        t = H5Gget_objtype_by_idx(group, j );
        printf("Object %d: %s type %d\n", (int)j, buffer, t);
}
```

H5Gget_objname_by_idx() returns *herr_t* and the name third parameter (char *). The second parameter is an index (of type *hsize_t*) and the fourth parameter passes the size of the buffer. The index is transient, what means that the index is only valid in the currently open group and can be different if the same group is opened again later.

H5Gget_objtype_by_idx() returns an *int* for the type of the hdf5 object at specified the group and index. Type number 0 stands for a soft link, 1 for a group, 2 for a data-set and 3 for a named data-type.

Another way to iterate through members of a group makes use of the H5Giterate() function. The group need not be open in before in that case.

```
H5Giterate(file, "/", NULL, file_info, NULL);
```

H5Giterate() returns an *int,* which is the return value of the function specified by the fourth parameter. It returns 0 if all elements of the group have been

processed. The function iterates over all members of the group specified by location id and path. The third parameter *int** is a pointer to an index where the iteration starts from. If it is set to *NULL* then the iteration starts with the first element of the group. The fourth parameter is a function pointer of a function that is called during iteration. Information can be passed to the function by using the fifth parameter, a *void** pointer. See [H5R06] for more information.

Comments that have been associated to hdf5 object can be read by the following function.

```
H5Gget_comment(file, "/Main_Data", 512, buffer );
```

H5Gget_comment returns *herr_t* and reads the associated comment of a hdf5-named-object specified by id and path into the *char** string buffer. The third parameter passes the size of the buffer.

An attribute has to be opened before its value can be read. Similar to the group functions attributes can be identified by name or by index. Functions to get the number of attributes and for iteration are provided also, functions to get the attribute´s data-space and data-type.

```
attrib = H5Aopen_name(dataset, "offset" );
H5Aread(attrib, H5T_NATIVE_DOUBLE, &offset );
```

H5Aread() returns *herr_t* and reads the attribute of type *hid_t* at a *void**. In this example into a *double* offset.

Maya Plug-In HDF5_Read_Fluid

A plug-in for the 3D application Maya® [MAY07] was developed under Windows® using Visual Studio® 2005. It demonstrates the previously discussed functionality of HDF5.

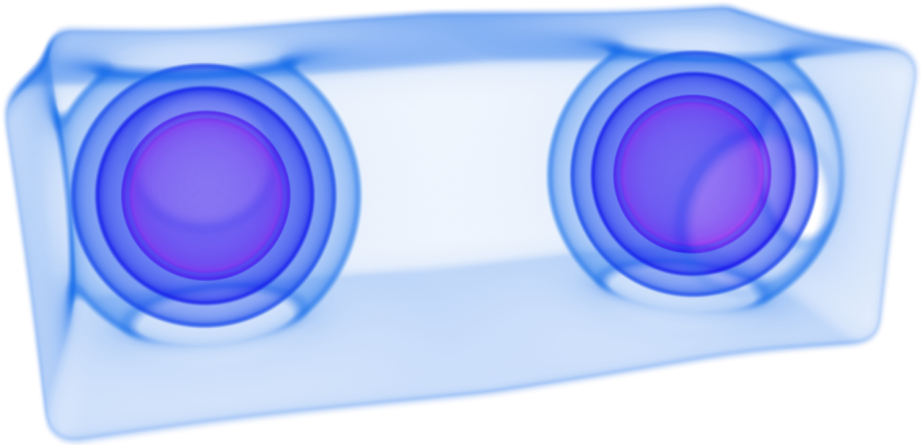


Figure 11: *Visualisation of a 3D density distribution*

The Maya® command plug-in reads a three dimensional density distribution from a HDF5 file into a Maya container for volume data. Before executing the command a Maya scene has to be prepared and must contain a volume container. The command implemented added by the plug-in must be executed while the container is the selected object in the 3D scene.

A Maya® fluid container allows usage of colour maps and to scale the visualised density dependent on the density value of the underlying scalar field. *Figure 11* shows a rendered images of a density distribution, which was provided by the relativity group of the Lusiana State University [LRG06]. The visualised density was scaled so that certain values of the density distribution form volume slices. They create a visual effect that is similar to displaying transparent iso-surfaces. Maya® produces smooth interpolated results with according high quality settings given the discrete data field with a resolution of 50x50x50 points. The images have been rendered with the rendering engine Mental Ray® [MEN07].

For a detailed description of Maya® plug-in programming see [GOU03].

F5

HDF5 provides efficient mechanisms for storing multidimensional arrays together with properties but it does not store additional information on the data like: “what is it” in any form. No information that would identify and allow correct interpretation of the contained data is added to the hdf5-file automatically. The F5 library tries to address this problem for the case of scientific data and offers a mechanisms to store the contextual information together with the data itself. HDF5 grouping, attributes and comments are used for this purpose.

F5 allows to formulate general concepts like a field, in a way that the data can be identified as representing a specific field, for example a vector field. Additional information that is needed for correct interpretation of the data, like the spacial distribution of the points a field is given on, is either stored in hdf5 data-sets or by making use of the hierarchical structures available from hdf5. All mechanisms to access subsets of data are mapped to the same efficient methods that are found in hdf5. In addition, the overhead for working through the F5 API[1] is kept minimal so there is nearly no performance penalty when using the F5 library. A C version of this API is available.

The broader aim of the F5 is to offer a complete data model for applications that operate on scientific data. It aims to avoid the need of repeated re-implementation of data file reader and writer code. F5 offers a sufficiently complete way to operate on, save and load scientific data on a high user level .

The concept, F5 was developed after, is based on the concept of fiber bundles, see *section Fiber Bundles* below. The proposition, developed by Butler and Pendly [BP89] that suggests to create a layered structural representation of manifolds by repeated aggregation of simpler objects and to apply the abstraction provided by the concept of vector bundles is as follows:

“At the lowest level, both the base and fiber are point sets, just collections of points X. For the next layer, the notion of neighbourhoods is added to obtain a topological space $T:=(X,\tau)$ like in def. 1 (see below). Then the notion of coordinates and differentiability is added to get a manifold $M:=(T,\{\{x^{\mu}\}\})$, like in def. 19. The fiber of interest here is a vector space, which can be considered as a manifold again, but with an additional layer of structure, i.e. the structure of linear algebra in the case considered here. The next layer then

1 Application Programming Interface

aggregates the base space B and the fiber F into a bundle *(B, F)*. Finally the bundle is aggregated with a map, which allows to specify values in each fiber to yield a section." [BEN04, 2.2.2 p39].

Fiber

Building on this pioneering idea the fiber bundle data model was developed by W. Benger [BEN04]. It allows the abstraction of the geometrical description of spacial objects from their numerical representation in a specific coordinate system and it offers an abstraction for the physical computation domain for the underlying discretisation scheme. It also allows to formulate grid-independent algorithms. It was implemented in C++ and makes use of generic programming techniques.

Fiber Bundles

Fiber bundles are introduced on the basis of several definitions. The definitions shown here are mainly taken from [BEN04]. Note, that this information is not necessary to work with F5, but explains the background and motivation of the data organisation.

(Def. 0)

Let S be a set. Then $P(S)$ is the set of all subsets of S, called the power set of S.

Example:

$S=\{1,2,3\}$, $P(S)=\{\emptyset,\{1\},\{2\},\{3\},\{1,2\},\{1,3\},\{2,3\},\{1,2,3\}\}$

(Def. 1)

Let X be a set $\wedge$ $P(X)$ be the power set. A subset $\tau \subseteq P(X)$ of the power set is a topology iff :
1) *arbitrary unions of elements of τ are contained in τ, i.e. if I is an arbitrary (also infinite) set of indices and $\forall i \in I : U_i \in \tau$, then $\bigcup_{i \in I} U_i \in \tau$,*
2) *finite intersections of elements of τ are contained in τ, i.e. if $U_0, U_1, .., U_n \in \tau$ then $\bigcap_{i=0}^{n} U_i \in \tau$ with $n \in \mathbb{N}$,*
3) *the emtpy set and the set X itself are contained $\in \tau$, i.e. $\emptyset, X \in \tau$*

(Def. 2)

The pair (X,τ) of a set X together with a topology τ on this set is a tolopogical space. The elements of a topological space are called points.

Examples:

The set $S=\{1,2,3\}$ with topology $\tau=\{\emptyset,\{1\},\{1,2,3\}\}$ is a topological space.

The set S with topology $\tau=\{\emptyset,\{1\},\{2\},\{1,2,3\}\}$ is not a topological space, since the union $\{1,2\}$ is not contained in τ

$\mathbb{R}$ with the set of open intervals $\tau=\{\bigcup_i (a_i,b_i): a_i,b_i\in\mathbb{R}, a_i<b_i\}$ is known as the standard topology on $\mathbb{R}$.

(Def. 5)

A subset $A\subseteq X$ of a topological space (X,τ) is a neighbourhood of an element of $p\in X$ iff it contains an element O of τ that contains p:
$A\subseteq X\ \upsilon(p)\Leftrightarrow\exists O\in\tau: p\in O, O\subseteq A$

(Def. 6)

Two topological spaces X, Y are homeomorphic, if there exists a bijective map $H: X\to Y$ such that open sets of X are mapped to open sets in Y and vice versa, i.e. the neighbourhood relations must be sustained under this mapping.
H is called homeomorphism or topological map.

(Def. 10)

The cartesian product $X\times Y$ of two topological spaces X, Y with the respective neighbourhood sets $\upsilon(x)\subset P(X), \nu(y)\subset P(Y)$ of the points $x\in X$, $y\in Y$ is a topological space, if the neighbourhood sets $\upsilon(x,y)$ of the point $(x,y)\in X\times Y$ are given by $\upsilon(x,y)=\{U\in\upsilon(x), V\in\nu(y): U\times V\subset W: W\}$

(Def. 50)

Let E and B being topological spaces and $f: E\to B$ be a continuous map. (E,B,f) is called a fiber bundle, if there exists a space F, such that the union of the inverse image of f of a neighbourhood $U_b\subset B$ of each point $b\in B$ is homeomorph to $U_b\times F$:

(E,B,f) *fiber bundle* $\Leftrightarrow\exists F:\forall b\in B:\exists U_b: f^{-1}(U_b)\simeq U_b\times F$

Thus, the space E can be locally be described by the product of a base space B (carrier) and a fiber space F. If this property also applies globally in space, E = B x F, then this bundle is called a trivial fiber bundle.

This separation appears within the fiber data model, where data elements are represented as fibers on geometrical entities. For example, a vector field on an uniform grid, describing the velocity of a fluid is a scientific data structure that can be represented by the data model. The vectors would be the fibers and the uniform grid the base space. Here the fiber bundle would even be trivial, because the fiber space is the same in each grid point and also the uniform grid is nested in one space.

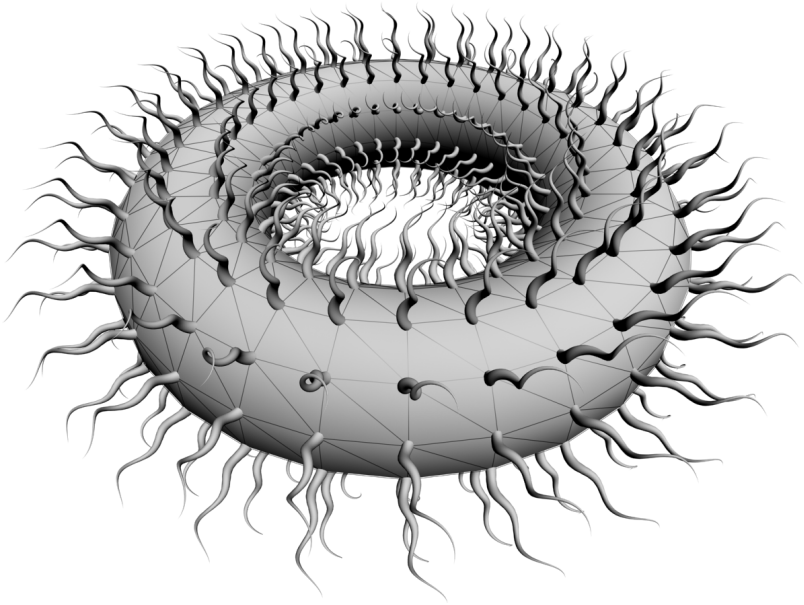


Figure 12: Separation into fibers and base space. The hairs represent the fibers and the torus represents the base space.

Figure 12 illustrates the general concept of the separation of a data structure into base space and fibers, where fibers are shown as “hairs” on a torus (the base space).

F5 File Organisation

The data structure inside a F5 file is organised in an acyclic graph containing a root node. The F5 file stores numerical data in data-sets, which are located at the end nodes[1], the leaves of the graph. A path through the graph starting from the root node to an data-set defines all properties of this data-set.

These properties are separated or grouped in six structuring elements, which are ordered hierarchically:

Bundle → Slice → Grid → Topology → Representation → Field.

Figure 13 shows an example F5 data structure. Data-sets are nested at the leaves, which are encircled with double lines.

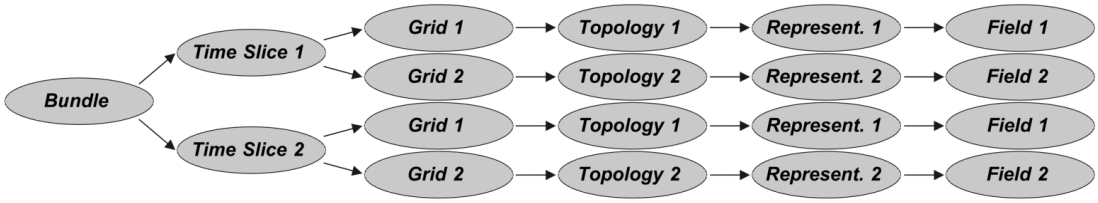


Figure 13: Graph of F5 data organisation.

Bundle

Bundle

Usually, a F5-*bundle* contains some data within a physical space, which consists of spatial dimensions and one time dimension. Thus, the base space can be modelled by a Cartesian product of time and space:

$$\textit{bundle base space: } \mathbb{R} \times \Sigma \qquad \textbf{(Eq. 1)}$$

Since F5 stores discrete data, numerical data is stored at certain points of time. All spatial data at an instant of time is grouped into a so called time *slice*.

The *bundle* is a container for *slices*, which is modelled by the root group in the HDF5 file. It is the uppermost element of the F5 data structure.

1 No data field is contained in the root node.

F5-*slices* are accessed via a float number representing the physical time of the according *slice*. Note, that this is different to many approaches, where integer time steps are used.

Besides time slicing one can also use a different float parameter to slice the base space of a F5-*bundle.* Such a slice is then called a parameter slice, see [Fib06, *Related Pages, Slice*]

Slice

Bundle/Slice

A *slice* represents a instant of time of a *bundle*. It contains geometrical entities of the *bundle* at that time. Such a geometrical entity of a *slice* is called a *grid object*.

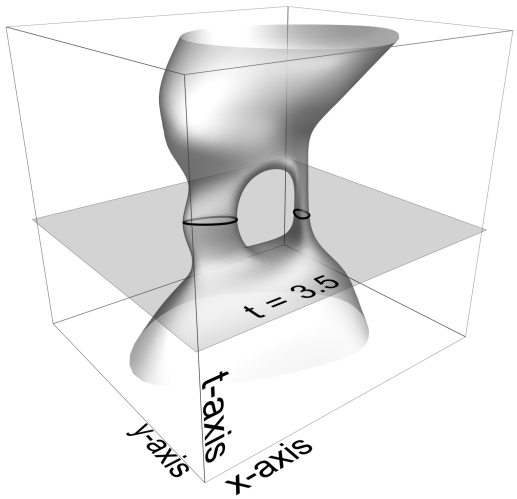


Figure 14: Time slice of an evolving 2D shape.

A F5-*slice* is modelled as a HDF5-group, that contains F5-*grid objects*. A *grid object* representing the same geometrical entity in a different time *slice* has the same name in each time *slice*. So an object can be traced as it evolves in time.

Figure 14 shows an example of a *slice* a t=3.5 of a 2D shape evolving over time, forming a 3D surface in the *bundle*. The intersection lines of the slice plane and the curved surface represent a *grid object* in the *slice*.

Grid

Bundle/Slice/Grid

A F5-*grid object* represents a geometrical entity, a manifold, or to be exact, a triangulation of a manifold[1].

It is modelled as a HDF5-group with a custom name. It is a member of a F5-*slice* and contains an *atlas* and F5-*topology objects*.

Data of a grid may be represented by different coordinate systems. All these coordinate systems contained in the *grid object* are collected in the *atlas*, which is modelled as a HDF5-group called "Charts". Transformation rules between compatible coordinate systems are stored there. For example, if the grid contains some data represented in Cartesian coordinates and some in polar coordinates the *atlas* would contain transformation matrices between those.

A *grid object* is decomposed in grid components, which share certain properties and are grouped to F5-*topology objects*. A *grid object* may contain many different *topologies*. The information of all grid components together define the topological properties of a *grid object*. The union of all "Positions" *fields* (see *section Field*) contained in a *grid* form the base space of the *grid*.

Topology

Bundle/Slice/Grid/Topology

The F5-*topology object* describes information about the topology of the spatial elements it contains and their neighbourhood information. It is modelled as HDF5-group, containing a data-set called "Neighbourhood" and certain attributes. Data represented in certain coordinate systems is grouped into F5-*representations*, which are also members of the *topology*.

The data-set **neighbourhood** defines which spatial elements are located next to a certain spatial element. In the general case, this information is stored as a list of indices of neighboured elements for each element. It is stored procedurally in special cases. For example, the indices of the neighbours in a rectangular grid layout of spatial elements can be calculated by a given index[2].

1 Since F5 deals with discrete data.

2 Neighbours of a 3D spatial element with index (i,j,k):
$(i,j,k) \rightarrow \{(i+1,j+1,k),(i+1,j-1,k),(i-1,j+1,k),(i-1,j-1,k),(i,j,k+1),(i,j,k-1)\}$

Spatial elements of a *topology* are so called *k-cells* or *collections of k-cells*. k stands for the **dimension** of the cell. *Figure 15* shows different examples of k-cells: (a) points are 0-cells, (b) edges are 1-cells, (c) faces are 2-cells and (d) cubic volumes are 3-cells.

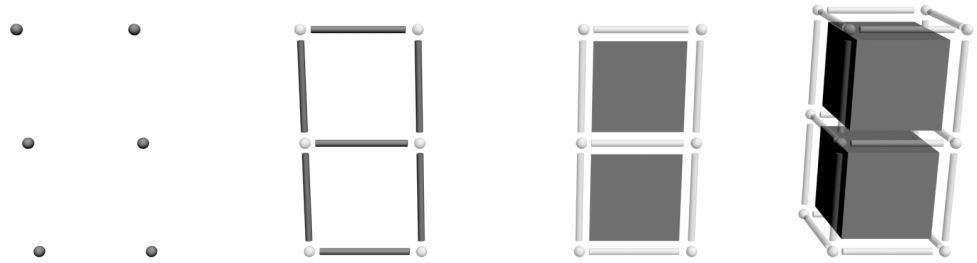


Figure 15: Examples of a 0-cell, a 1-cell, a 2-cell and a 3-cell

A spatial element can also be defined based on other spatial elements. For example, an edge can be defined by two vertices or a path of connected edges can be defined by a number of edges. To describe such 'higher order' structures an **index depth** is introduced. The spatial element that is not based on others has index depth 0 and is a point in space called vertex. The index depth represents the number of dereferencing operations to get to the element with index depth 0. Thus, an edge has index depth 1 and a path has index depth 2. The table below shows different examples. (table taken from [BEN04, table 3.1, page 64])

Spatial element	*Index depth*	*Dimensionality*
Vertex	0	0
Edge	1	1
Face	1	2
3-Cell	1	3
Collection of Vertices	1	0
Path of edges	2	1
Surface built from faces	2	2
3-Cell complex	2	3
Set of cell complexes	3	3

Another property stored in the *topology* is the refinement level of a spatial element. See *figure 16* for an example showing some refinement levels of a triangular surface.

Related spatial elements with different refinement levels must be of same dimensionality and index depth. They can then be associated to spatial elements

of a different *topology*, with a different refinement level and a different number of spatial elements.

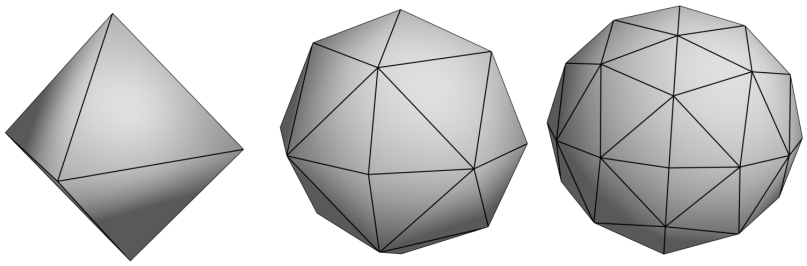


Figure 16: Different refinement levels of a triangular surface.

F5-*topology* objects are unique in a *grid* object and are identified by a string. It holds an arbitrary number of F5-*representation* layers sharing the same coordinate system properties.

F5 provides 4 predefined topologies, but other topology groups may be defined: “Points”, “Edges”, “Faces” and “Connectivity”:

- Points:
  This is the fundamental topological subgroup, because it contains the vertex information of a grid.
- Connectivity:
  This subgroup contains information about k-cells which construct a grid (with k > 0). An example for a connectivity group would be a triangular surface where the triangles are build by triples of indices to points of a Points *topology*, see [FIB06, Related Pages, Topology Objects].

Representation

Bundle/Slice/Grid/Topology/Representation

The F5-*representation* layer holds information about the coordinate systems valid in the data fields it contains and is a member of a F5-*topology* layer. It is a HDF5-group named like the representing coordinate system.

A *representation* always contains one specific F5-*field* called “Positions”, which stores the positions of the spatial elements of a *topology*. Furthermore, it contains an arbitrary number of F5-*fields* with custom names, containing data at

the “Positions”. They have the same spatial dimension as the positions *field*. A data element of a data *field* can then be mapped to the according “Positions” of the spatial elements.

In a general case “Positions” is a HDF5 data-set of vertices using the same indices as the data fields, see *figure 17(a)*.

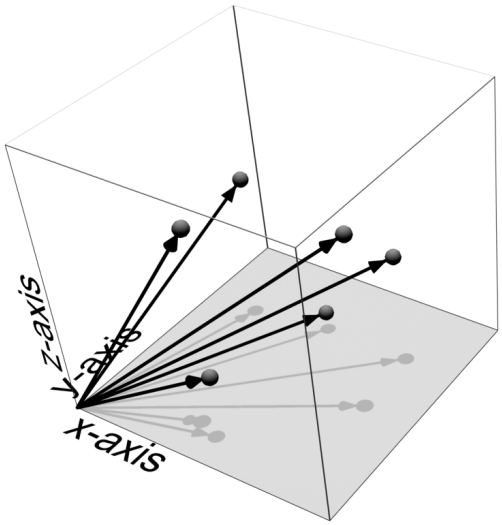


Figure 17(a): General “Positions”

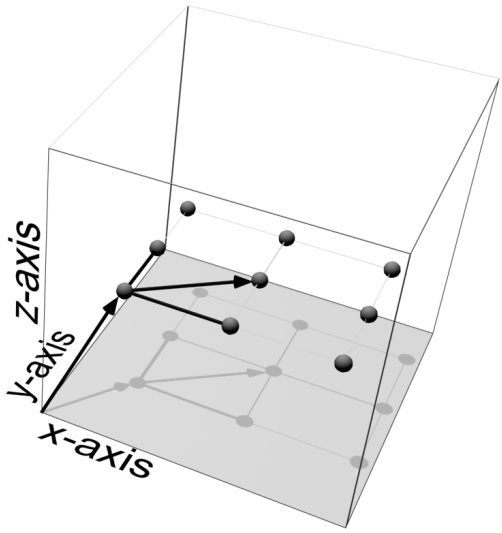


Figure 17(b): Uniform “Positions”

In the case of describing vertices in uniformly distributed 3D Cartesian coordinates, only a position at the index (0,0,0) called *origin* and a vector called *delta* describing the vector between two diagonal positions is sufficient, see *figure 17(b)*. A position dependent on the element index can then be calculated by:

$$\vec{P_{i,j,k}} = \vec{O} + \begin{pmatrix} i \cdot \delta_1 \\ j \cdot \delta_2 \\ k \cdot \delta_3 \end{pmatrix} \qquad \textbf{(Eq. 2)}$$

$\vec{P}$ ... *position*, $\vec{O}$ ... *origin*
i, j, k ... *array indices*

In that special case “Positions” is a HDF5-group containing the HDF5-attributes called “origin” and “delta”.

The data *fields* contained in a *representation* group are the fibers of the *grid* object. But, “Positions” is a F5-*field* and no fiber, since it is part of describing the base space of the *grid*.

Several predefined coordinate systems are provided. Coordinate representations are called “Charts” in F5:

Cartesian 4D	Integer3D	Cartesian 2D	triangular
Polar 4D	Rational3D	Polar 2D	edge
	Cartesian3D	Spherical 2D	quad
	Polar 3D	Axial 2D	tetrahedral
	Cylindrical 3D	Texture 2D	hexahedral
	Texture 3D		

Field

Bundle/Slice/Grid/Topology/Representation/Field

A F5-*field* is an element of a F5-*representation* layer and is identified by a custom string. It contains a discrete data set, an array. It maps an index or a tuple of indices to a data element, which typically is an index, a scalar, a vector, a colour or a tensor of order 2. F5 uses a HDF5 data-sets to store the data in the F5 file. *Figure 18* shows a 2D uniform scalar field as an example for a F5-*field*.

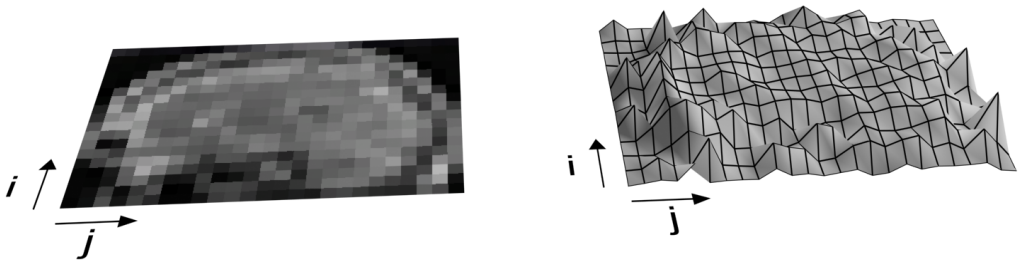


Figure 18: Scalar field visualised as an intensity map(left) and a height map(right).

The F5 data model allows to create many different types of evolving grid objects and store data to their spatial elements or collections of elements. Also representations using different coordinate systems are possible.

Data is shared inside the data structure by HDF5 links.

An unlimited number of structures can be build. Examples reach from simple scalar fields on a uniform grids to colour fields on triangular surfaces to many metric tensor fields on refining irregular meshes.

To work with F5 data it is not necessary to completely understand its data

organisation. The user works on a simple interface that involves only **slices, grids and fields**. Charts, topologies and further data organisation is encapsulated by higher level functions.

The F5 Library

The F5 library is written in C and is available as full source distribution at [FIB06]. Specially adapted distributions for the scientific applications CACTUS [CAC06] and Amira™ [AMI06] are available besides a standalone library.

Included with the standalone distribution is a make system [MEM07] that comes very handy when developing F5 tools for different platforms since many available platforms are supported.

After unpacking the compressed archive, the source files are found in the directory F5/ and split to .h and .c files. Various applications can be found in the F5/apps directory. They include examples, converters and tools.

The tools are: F5ls, F5merge, which was not yet implemented, Q5ls and F5smoothparticels.

Some converters for different file formats to and from F5 are provided. Certain F5 files can be converted to truevision's TGA[1] images or to OBJ, a format developed by alias-wavefront [tm] to store polygon based objects and free form curves and surfaces.

Some formats can only be converted to F5. This includes the following formats: ADCIRC, used by a research group involved in hydrodynamics [ADC06], the CARPET and GeoTIFF format for interchanging geo-referenced raster images, MM5[2] a format for atmospheric data [MM506] and GSSE a format for scientific computing from the GSSE[3] group [GSS07].

The following section demonstrates how to work with existing F5 files using the predefined library objects. Reading and writing files is explained. It would be possible to to add, for example, new topology objects but this would require a detailed description of F5 low-level functions, which cannot be provided here.

1 Truevision Advanced Raster Graphics Array
2 mesoscale model
3 Generic Scientific Simulation Envirnoment

Installation of F5 for Linux

An archive of the current release of the F5 library can be downloaded at [FIB06]. The file *FiberHDF5.tgz* is the standalone version of F5. It should be unpacked and uncompressed into the directory, which will contain the F5 Library. The archive already contains a directory FiberHDF5.

```
tar xfz  FiberHDF5.tgz
```

For a local user installation this command can be executed in the home directory, for a global installation a directory in /opt/ would be appropriate.

F5 provides a sophisticated make file that automates the installation process including insatllation of the hdf5 library. To use this feature a make command must be executed in the *FiberHDF5/hdf5.ref* directory.

The *Makefile* in this directory must be edited in before. The hdf5 target directory should be set (*HDF_INSTALL_DIR*) and the URL[1] of the hdf5 archive should be checked (*HDF5URL*).

```
./make get unpack configure make install_lib
```

This should download, extract and install the necessary hdf5 library files and then compile and install the F5 library. For troubleshooting see the *README* file in *FiberHDF5/hdf5.ref*.

If the installation was successful an example can be compiled, for example in the directory *FiberHDF5/F5/apps/examples/ScalarSimple.* The *TARGET* in the *Makefile* must be changed to *ScalarSimple*.

```
./make exec
```

This compiles and executes the example file. Four F5 files should have been created in the local directory.

F5ls

F5ls is a tool for examining F5 files by printing its content in ASCII to the standard output stream. To compile the F5ls binary execute

```
./make
```

1 Uniform Resource Locator

in the *FiberHDF5/F5/apps/tools/F5ls* directory. The binary will be compiled and placed into a directory, which is named after platform and compilation mode, for example, */FiberHDF5/F5/bin/arch-Linuxi686-Debug*. This binary directory should be added to your PATH environment variable, see *section Install hdf5 for cygwin and Linux*.

For every time slice F5ls prints all *grid* objects and the *fields* they contain. Its output is as follows:

```
******* Timeslice for t=0 *******
  Grid `steam'  (no timestep information)
  Root level vertex fields:
    Positions      : UniformSampling  <0.1.2>  Size: 3x2x2  cartesian coordinates
                     Range: [0,0,0]-[1,0.5,0.5]
    temperature  :  Contigous          <0.1.2>  Size: 3x2x2  scalar
```

'Root level' specifies the refinement level of the following fields, which stands level 0 here. The subsequent information given for the fields then is their name, their internal memory layout, the size of each dimension and their type. '<0.1.2>' is the current version of the field implementation. In case this implementation changes, F5ls can still support different versions. The F5 file shown contains a scalar field on a uniform 3 dimensional spacial grid.

The F5ls provides the user with all necessary information to work with the F5 file.

Q5ls

Q5ls provides the same information as F5ls but is presented as a tree in an graphical user interface window[1]. See *figure 19* for an example:

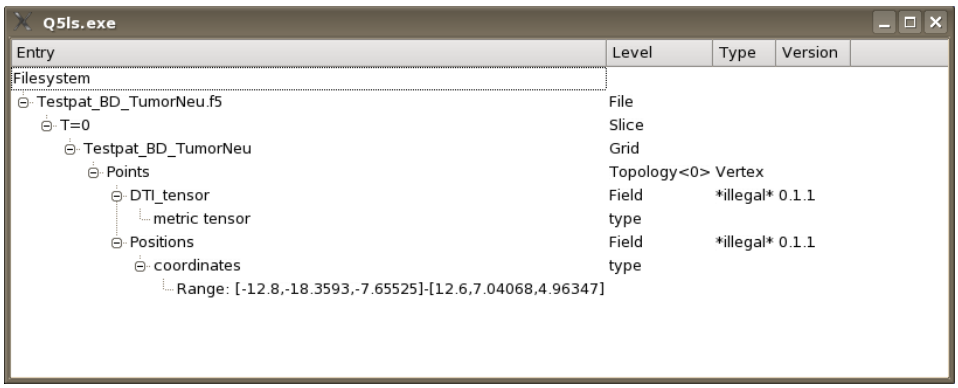


Figure 19: Screenshot of the content of the file *Testpat_BD_TumporNeu.f5*.

1 The library Qt® from Trolltech® is used for this purpose [TRO07].

Important Data Types and Functions

Basic Data Types:

F5 supports a number of basic data structures, which can be used as basic building blocks for more complicated data structures. Declarations are in the header files *F5types.h* and *F5coordinates.h*. Most types are based on the two types *F5_float_t* and *F5_int*_t, which are *typedefs* to a C-*float* and a C-*int*.

The following table shows data structures based on the *typedef float F5_float_t*.

Type	*Description*
F5_vec3_point_t	**struct of three floats called x,y,z representing a point**
F5_vec3_float_t	**struct of three floats called x,y,z representing 3 floats**
F5_metric3_float_t	**struct for metric33 tensor elements: gxx, gxy, gxz, gyy, gyz, gzz**
F5_polar_point3_float_t	**struct with elements r, theta, phi representing a point in polar coordinates**
F5_texture_point_t	**struct with elements u and v representing texture coordinates on a surface**
F5_rgb_real_t	**struct with elements r, g, b representing red, green and blue colours. (as F5_float_t)**
F5_rgba_real_t	**Same as above, but with additional element a for alpha**

Structures based on typedef int F5_int_t.

Type	*Description*
F5_edge_t	**struct with 2 elements i, j defining an edge**
F5_triangle_t;	**struct with 3 elements i, j, k defining a triangle**
F5_quad_t	**struct with 2 element i[2], j[2] defining a rectangle**
F5_quadL_t	**same as above but different element i[4]**
F5_faces_t	**struct with 3 elements i, j, k defining a triangular surface**
F5_tetra_t	**struct with 4 elements i, j, k, l representing a tetraeder.**
F5_hexahedron_t	**struct with 4 elements i[2], j[2], k[2], l[2] representing a cubic.**
F5_hexahedronL_t	**same as above but with element i[8]**

Other structures

Type	*Description*
IntegerFraction	**struct with 2 elements int num, int denum representing a rational number**
IntegerFraction3D	**struct with 3 IntegerFraction elements x, y, z**
F5_refinement3D_point_t	**union representing a 3D point using rational integer coordinates with IntegerFraction3D crd or inedexed IntegerFraction d[3]**
TensorTypes	**A tensor representation based on F5Ttensor_t****

ChartDomain_IDs

The *struct ChartDomain_IDs* declared in *F5Bchart.h* is the internal representation for coordinate systems (charts) and has the following layout:

const char*	**domain_name**
unsigned	**refs**
int	**perm_vector[FIBER_MAX_RANK]**
hid_t	**Point_hid_t**
hid_t	**Vector_hid_t**
hid_t	**Covector_hid_t**
hid_t	**Bilinearform_hid_t**
hid_t	**Metric_hid_t**
F5Ttensor_t**	**TensorTypes**

Each Chart has a name, a reference counter and a permutation vector, that specifies what ordering is used in multidimensional arrays[1]. The ids *hid_t* point then to prototype elements of the coordinate system. These prototypes are stored as HDF5-named-types and store the data types used for the numbers, names of the elements and type of the coordinate. For example *Point_hid_t* could be the id *hid_t* to a hdf5 compound type containing the elements of type *H5T_NATIVE_FLOAT* with names “x”, “y” and “z”. For further examples see [FIB06, Related Pages, HDF5 Chart Objects]

In the current version of F5 most floating point data are based on *floats* one must create a new chart to write *double* floating point data. In contrast, reading of *double* data to data of a compatible chart using *float* would be no problem, since

1 For example, C and FORTRAN use different orders.

HDF5 can convert them automatically.

Functions to create supported chart objects (see *section Representation*) are implemented in *F5coordinates.c*. They use four basic functions, which are declared in *F5B.h* and defined in *F5B.c.* They return a *struct ChartDomain_IDs* can be used to create new user defined charts.

```
F5B_new_global_domain()                 //for positional types,
F5B_new_global_chart()                  //for also tangential and tensor types,
F5B_new_integer_regular_domain3D()      //for mappings between points and
F5B_new_rational_regular_domain3D()     //for mappings using integer fractions.
```

F5Path

The *struct F5Path* declared in *F5Path.h* describes the location of a F5 object like a slice, a grid, a chart, etc. inside the f5-hdf5-file. It contains all *hid_t* IDs of hdf5 objects involved. The *F5Path struct* is important when reading F5 files. All objects that contain information can be accessed via the *F5Path struct.* It has the following members:

```
ChartDomain_IDs *   myChart             hid_t   Grid_hid
ChartDomain_IDs *   FileIDs             hid_t   GlobalChart_hid
char *              field_info          hid_t   Charts_hid
hid_t               field_enum_type_hid hid_t   Chart_hid
hid_t               File_hid            hid_t   Topology_hid
hid_t               ContentsGroup_hid   hid_t   Representation_hid
id_t                Slice_hid           hid_t   Field_hid
```

Writing F5

Some high level functions have been implemented in the F5 library. They all write or append data blocks of certain types to F5 files. For example, to write a vector field specified on a uniform Cartesian 3D space following function can be used:

```
F5_API F5Path* F5Fwrite_uniform_cartesian3D(hid_t file_id, double time,
                                const char*gridname,
                                const F5_vec3_point_t*origin,
                                const F5_vec3_float_t*spacing,
                                hsize_t dims[3],
                                const char*fieldname,
```

```
                                hid_t fieldtype,
                                const void * dataPtr,
                                const char*coordinate_system,
                                hid_t property_id);
```

The parameters include a file id, a time *slice*, the *grid* name, parameters to specify the positions of the uniform grid, a *field* name, the type of the *field* elements, the data pointer, a coordinate system and a additional property list for HDF5.

The following field types declared in *F5coordinates.h* are available:

```
F5T_COORD3_FLOAT          F5T_TRIANGLE
F5T_VEC3_FLOAT            F5T_QUAD
F5T_METRIC33_FLOAT        F5T_FACES
F5T_INT_FRACTION          F5T_TETRAHEDRON
F5T_EDGE                  F5T_HEXAHEDRON
```

These are convenient *#defines* of chart functions and are pointing to elements of a *ChartDomainId*. For example:

```
#define F5T_COORD3_FLOAT    F5B_standard_cartesian_chart3D()->Point_hid_t
```

The high level functions are declared in the header files:

```
F5particles.h       for particle based data
F5surface.h         for surface based data
F5uniform.h         for data based on uniform grids
F5AMR.h             for data based on adaptive meshes
F5image.h           for image based data
```

Detailed information can be found on-line [FIB06] or directly in the source code.

Reading F5

Reading F5 data structures is done by iterators, similar to iterators used in HDF5. F5 iterators are declared in *F5iterate.h*.

To open a certain field iterators over slices, grids and fields should be used. First the time slice iterator must be called and a callback is function passed. The callback is invoked in each slice iteration step and has to be implemented by the user.

```
int F5iterate_timeslices(     hid_t file_id, int *idx,
                              F5_iterate_timeslices_t*ts_callback, void *user_data ) ;
herr_t ts_callback(F5Path*slicePath, double time, void *user_data);
```

The iterator iterates over all time slices of the F5 file starting at group (or slice) with index idx. User data can be passed to the callback function using the pointer user_data. The current id to the F5 object (slice) in the iteration is available via the *struct F5Path*.

To iterate over all grids in the slice group one has to call the F5 grid iterator function inside the slice callback function.

```
int F5iterate_grids(  F5Path*F5Slice, int *idx,
                      F5_iterate_grids_t*gr_callback, void *operator_data);
herr_t gr_callback(F5Path*grid, const char*gridname, void *operator_data);
```

Finally, a field iterator is used inside the grid callback function to iterate over fields contained in a grid, for example for vertex based fields:

```
int F5iterate_vertex_fields( F5Path*grid, int *idx,
                             F5_iterate_fields_t*fd_callback, void *operator_data,
                             const char*coordinate_system, F5_fieldtype_t*what) ;
herr_t fd_callback(F5Path*field, const char*fieldname, void *operator_data);
```

Inside the field iterator the required data-set can be accessed using F5 functions or HDF5 functions, like H5Dread().

The following example shows some pseudo code for an iteration to read a field at a certain time in a certain grid:

```
typedef struct
{
        char* grid_name;
        char* field_name;
        double time;
        ...
} my_Data;

herr_t fd_callback(F5Path*field, const char*fieldname, void *operator_data);
herr_t gr_callback(F5Path*grid, const char*gridname, void *operator_data);
herr_t ts_callback(F5Path*slicePath, double time, void *user_data);

int main()
{
        F5Path Path;
        my_Data dat;
        /* fill data of dat */
        ...
        F5iterate_timeslices(file_id, NULL, ts_callback, &dat);
        return 0;
}
```

```
herr_t field_iterator(F5Path*field, const char*fieldname, void *operator_data)
{
my_Data*dat = (my_Data*)(operator_data);
        ...
        if(strcmp(dat->field_name, fieldname) == 0)
        {
                /* open file, use F5 or HDF5 functions to read data-set */
        }
        return 0;
}

...
```

F5 functions that can be used for reading inside the field callback function are field functions declared in *F5F.h* and high level functions of supported structures like uniform grids (*F5uniform.h*), surfaces (*F5surface.h*), adaptive meshes (*F5AMR.h*) and particles (*F5particles.h*).

Functions declared in F5F.h

All function use a *F5Path* to refer to a certain F5 object. This *F5Path* is automatically provided inside the field iterator callback function.

Open and close a field by given F5Path and name.

```
int F5Fopen(F5Path*f, const char*fieldname);
void F5Fclose(F5Path*f);
```

Check if the field is a group.

```
int F5Fis_group(const F5Path*);
```

Get data-type and data-space of the field elements.

```
hid_t F5Fget_type(F5Path*f);
hid_t F5Tget_space(F5Path*f);
```

Permute a given array of dimensions to the memory ordering of a source coordinate system.

```
hsize_t*F5Tpermute_dimensions(  F5Path*fpath, int rank, hsize_t*target_dims,
                                        const hsize_t*source_dims );
```

Get the number of elements in each dimension of a given field. This is equivalent to calling first F5Tget_space() and second F5Tpermute_dimensions(). Return value is zero in case of an error.

```
int F5Tget_extent(F5Path*f, hsize_t*dims, int maxDims);
```

Get the index_depth of the fields elements, and the refinement level. Results are returned via pointers. Return value is zero in case of an error.

```
int F5Tget_index_depth(F5Path*f, int*result);
int F5Tget_refinement_level(F5Path*f, hsize_t*dims, int maxDims);
```

Get the minimum and maximum values of the fields elements. Minimum and maximum are stored as the same type as the field elements. In case a "Positions" field is read this function can be used to get the bounding box of the positions.

```
int F5Fget_range(F5Path*f, void*min, void*max, hid_t memtype_id);
int F5Fget_fragment_range( F5Path*f, const char*fragment_name,
                           void*min, void*max, hid_t mem_type_id );
```

Get the average value of all elements of a field.

```
int F5Fget_average(F5Path*f, void*avg);
```

Get the standard deviation of the average value.

```
int F5Fget_deviation(F5Path*f, void*dev);
```

Check if a field is a linear mapping of points to values.

```
int F5Fis_linear(F5Path*fpath, const char*fieldname);
```

Similar checks for different mappings.

```
int F5Fis_fragmented(F5Path*fpath, const char*fieldname);
int F5Fis_separatedcompound(F5Path*fpath, const char*fieldname);
```

Read a linear field. A linear field is completely described by dimensions, origin(base) and delta. Return value is zero in case of an error.

```
int F5Fread_linear(F5Path*fpath,
                   hsize_t*dims,
                   hid_t fieldtype, void*base, void*delta);
int F5Fread_linearo(F5Path*fpath, const char*fieldname,
                   hsize_t*dims,
                   hid_t fieldtype, void*base, void*delta);
```

Get the F5Path of the according field in the previous time slice.

```
F5Path*F5FopenMostRecentSlice(hid_t File_id, double*t, const char*gridname,
                              const char*fieldname,
```

```
                                        const char*coordinate_system);
```

Returns a hdf5 named data-type describing the field-type. The named data type is also added to the file specified in F5Path.

```
hid_t F5file_type(F5Path*fpath, hid_t fieldtype);
```

F5Fgrab() returns the hid_t of a field and removes it from the F5Path.

```
hid_t F5Fgrab(F5Path*f);
```

Functions declared in F5L.h

These functions operate on fields but are of a lower level than those shown above. They provide similar functionality and are not further described here.

```
F5LTget_index_depth( hid_t Top_hid );
int F5Lis_linear( hid_t Rep_id, const char*fieldname );
int F5Lread_linear(   hid_t F_id, hsize_t*dims,
                      hid_t fieldtype, void* base, void* delta);
hid_t F5Lget_type( hid_t Field_hid, int FieldIDisGroup );
int F5Lget_field_dimension_and_type(      hid_t Representation_hid,
                                          const char*fieldname,
                                          hsize_t dims[FIBER_MAX_RANK],
                                          hid_t*type_id );
int F5LAget_dimensions(     hid_t Field_id, const char*aname,
                            hsize_t dims[FIBER_MAX_RANK]);
```

Functions declared in F5uniform.h

Get bounding box and dimensions of a uniform grid of vertices in Cartesian 3D coordinates.

```
hid_t F5BgetUniformCartesianGridVertexData3D(  hid_t SliceID,
                                        const char*gridname,
                                        const char*fieldname,
                                       F5_vec3_point_t*bbox_min,
                                       F5_vec3_point_t*bbox_max,
                                       int dims[3]);
```

Example F5 file

Data, collected and derived from a MRI[1] scan of a human brain, was read using the F5 library. More information about the MRI data sets can be found in [BBH06]. The F5 file *Testpat_BD_TumorNeu.f5* is serves as an example in this thesis. First inspect the high level information contained inside the file by using F5ls.

```
$ F5ls Testpat_BD_TumorNeu.f5
******* Timeslice for t=0 *******
  Grid `Testpat_BD_TumorNeu'  (no timestep information)
  Root level vertex fields:
  DTI_tensor    : *illegal*   <0.1.1>  Size: 128x128x56  metric tensor
  Positions     : *illegal*   <0.1.1>  Size: 128x128x56  cartesian coordinates
                  Range: [-12.8,-18.3593,-7.65525]-[12.6,7.04068,4.96347]
```

The file contains a metric tensor *field* called 'DTI_tensor' at time *slice* t=0.0, it resides on a *grid* called 'Testpat_BD_TumorNeu'. The size and the range of the Cartesian coordinates is shown. The 'Range' output informs us on the spacial data range at the positions. At array index (0,0,0) the position of the element is (-12.8,-18.3593,-7.65525) and at (127,127,55) it is (12.6, 7.04068, 4.96347).

Certain error messages are printed when using F5ls with the file. They come from the changes the F5 library has undergone during development. The f5-file was created with an older version of the F5 and hdf5 library. This is also the reason why the array type is shown as *illegal*, but in fact it is “Contiguous” for the “DTI_tensor” and “UniformSampling” for the “Positions”.

To examine the details of a F5 file one can use the hdf5 low level tools. This step is usually not required, but demonstrates the F5 data organisation:

```
$ h5ls -r Testpat_BD_TumorNeu.f5

/Charts                  Group
/Charts/Cartesian3D      Group
/Charts/Cartesian3D/Metric Type
/Charts/Cartesian3D/Point Type
/Charts/Cartesian3D/StandardCartesianChart3D Group
/Charts/Cartesian3D/StandardCartesianChart3D/Coordinates Group, same as
/Charts/Cartesian3D

/T=0                     Group
/T=0/Testpat_BD_TumorNeu Group
```

1 Magnet resonance imaging uses images of magnet resonance tomography

```
/T=0/Testpat_BD_TumorNeu/Charts Group
/T=0/Testpat_BD_TumorNeu/Charts/StandardCartesianChart3D Group
/T=0/Testpat_BD_TumorNeu/Charts/StandardCartesianChart3D/GlobalChart ->
/Charts/Cartesian3D/StandardCartesianChart3D
/T=0/Testpat_BD_TumorNeu/Fields Group
/T=0/Testpat_BD_TumorNeu/Fields/Positions Group
/T=0/Testpat_BD_TumorNeu/Points Group
/T=0/Testpat_BD_TumorNeu/Points/StandardCartesianChart3D Group
/T=0/Testpat_BD_TumorNeu/Points/StandardCartesianChart3D/DTI_tensor Dataset
{56, 128, 128}
/T=0/Testpat_BD_TumorNeu/Points/StandardCartesianChart3D/Positions Group

/TableOfContents        Group
/TableOfContents/Fields  Group
/TableOfContents/Fields/DTI_tensor Group
/TableOfContents/Fields/DTI_tensor/Testpat_BD_TumorNeu ->
/TableOfContents/Grids/Testpat_BD_TumorNeu
/TableOfContents/Fields/Positions Group
/TableOfContents/Fields/Positions/Testpat_BD_TumorNeu ->
/TableOfContents/Grids/Testpat_BD_TumorNeu
/TableOfContents/Grids   Group
/TableOfContents/Grids/Testpat_BD_TumorNeu Group
/TableOfContents/Grids/Testpat_BD_TumorNeu/T=0 -> /T=0
/TableOfContents/Parameters Group
/TableOfContents/Parameters/Time Group
/TableOfContents/TypeInfo Type
```

The root group contains time *slices* groups as well as the groups “Charts” and “TableOFContents”. The “TableOfContest” group is introduced to organise data in a reverse order, using groups and links. No real data is stored there. The aim is to simplify traversing the graph internally. 'Charts' is a global atlas for the F5 file and is similar to a grid specific atlas, see *section Grid*.

The *slice* group is named “T=0” for the *slice* at time t=0. The *slice* contains a *grid object*. Inside the *grid object* resides the atlas “charts” that contains the coordinate system used in subgroups of the *grid*, a ”StandartCartesian3D”.

The *grid object* also contains a subgroup called “Fields”, which is an additional linking in the structure to simplify data access for F5 internally.

“Points” is a *topology* of the *grid* that contains the coordinate *representation* “StandartCartesian3D”. This *representation* houses the group “Positions” (which contains origin and delta, according to *section Representation*) and the actual tensor data-set.

Basic Tensor Type Data Structures in C and C++

To test and demonstrate the usage of F5 some data structures have been implemented in C and C++. These data structures are an collection of objects and functions to operate on tensor data based on uniform grids. Types for scalar, vector and metric tensor fields have been implemented and are applicable off the shelf.

C

This section presents one data structure implemented in C for operating on a metric tensor field data. The other structures for scalar and vector types have been implemented similarly and are therefore not described.

A struct TSMetric33F[1] is declared, which can be used for storing data of one metric tensor field of a uniform grid object of a certain time slice (declaration can be found in TSMetric33F.h). The struct includes elements that store information time, name of the grid, name of the field and name of the tensor elements, which are “gxx”, “gxy”, “gxz”, “gyy”, “gyz” and “gzz”.

Other data collected in the struct are the minimum and maximum value of the tensor elements, the positions of the grid points via origin and delta and a pointer to the float data field containing the numerical data. The multidimensional data is stored in an one dimensional array with a layout in the following order: elements, x, y, z. (elements in the inner loop, z in the outer loop)

```
typedef struct
{
        double slice;           // physical time of slice
        char*grid_name;         // name of the according grid object
        char*field_name;        // name of the according field

        int elements;           // number of tensor elements
        char**elements_names;

        hsize_t dim[3];    // xyz dimensions of data

        F5_vec3_point_t origin; // position at position index xyz <0,0,0> of *data
        F5_vec3_float_t delta;   // distances between two discrete positions
```

1 TSMetric33F stands for **T**ensor **S**lice **Metric33 F**loat

```
        float*elements_max;  // max value of each element in order of memoryorder
        float*elements_min;  // min value of each element in order of memoryorder

        int datasize;
        float*data;         // data block memoryorder elem, x,y,z
} TSMetric33F ;
```

Several functions that operate on the struct have been implemented as well. All function names start with "TSMetric33_" and pass a pointer to the struct as the first parameter.

There are functions to create and allocate memory for the elements in the struct, to reallocate memory and to free a struct:

```
int TSMetric33F_create(      TSMetric33F*TFS, const char*gridname,
                             const char*fieldname, double time,
                              int xdim, int ydim, int zdim)
int TSMetric33F_reallocdata( TSMetric33F*TFS, int xdim, int ydim, int zdim)
int TSMetric33F_free( TSMetric33F*TFS)
```

There are functions to print further information on the console for debugging purposes: information about the data excluding the numerical tensor field data, a dump of all tensor element values and an output of one tensor at a given index.

```
void TSMetric33F_printinfo( TSMetric33F*TFS )
void TSMetric33F_printdata( TSMetric33F*TFS )
void TSMetric33F_print( TSMetric33F*TFS, int i, int j, int k )
```

A function that calculates the minimum and maximum values of each tensor element.

```
void TSMetric33F_setminmax( TSMetric33F*TFS )
```

Data can be accessed through several methods. The user can decide to use set and get functions. They implement index range check and fail if the index is out of range.

```
int TSMetric33F_set( TSMetric33F*TFS, double d, int e, int x, int y, int z )
double TSMetric33F_get( TSMetric33F*TFS, int e, int x, int y, int z )
```

The user can also operate directly on the *float** of the data block using memcpy or similar.

When accessing the data block as an one dimensional array[1] one can use the

1 In C pointer and array references can be used interchangeably.

index_map function that calculates the according one dimensional index for the data array by given tensor element index and 3 indices of the 3D uniform grid.

```
int TSMetric33F_index_map( TSMetric33F*TFS, int e, int x, int y, int z )
```

Inside the index map function a range check can be enabled by a macro definition previous to the inclusion of the *TSMetric33F.h*.

```
#define TS_INDEXCHECK
```

A program can be developed and executed in a "save" mode. After a successful testing, the range check can then be disabled.

The index validation function is used internally and returns 0 if an index is within the correct range.

```
int TSMetric33F_valid_index( TSMetric33F*TFS, int e, int x, int y, int z )
```

Finally, functions for reading and writing the data structure to and from a F5 file are provided. Their parameters include a struct pointer, the file name, the time slice, the grid name and the field name.

```
int TSMetric33F_append( TSMetric33F*TFS, char*filename )
```

The append function writes data of a TSMetric33F struct into the F5 file. It creates a new file if the file specified does not exist, otherwise data is appended to the file. If a field in the file has the same name, slice, grid, charts etc. it is overwritten.

To overwrite a existing F5 file delete it before calling the append function[1].

```
int TSMetric33F_open(      TSMetric33F*TFS, const char*filename, double t,
                           const char*gridname, const char*fieldname )
```

The iterator callback functions are used internally in the open function, according to *section Reading F5*.

```
herr_t TSMfield_iterator(F5Path*field, const char*fieldname, void *operator_data)
herr_t TSMgrid_iterator(F5Path*grid, const char*gridname, void *operator_data)
herr_t TSMtimeslices_iterator(F5Path*slicePath, double time, void *user_data)
```

1 For example, one can use the unlink(char *path) declared in unistd.h under Linux.

The following example opens a metric tensor field from a f5-file, modifies the data and appends it into the same file at a different time slice.

```
//#define TS_INDEXCHECK
#include "TSMetric33.h"

int    main()
{
   int e, i, j, k;
   TSMetric33F tumor;

   TSMetric33F_open( &tumor, “Testpat_BD_TumorNeu.f5", 0.0,
                          "Testpat_BD_TumorNeu", "DTI_tensor" );

   TSMetric33F_printinfo( &tumor );
   TSMetric33F_print( &tumor, 8,24,40 );

   for(k = 0; k < tumor.dim[2]; k++) {
       for(j = 0; j < tumor.dim[1]; j++) {
          for(i = 0; i < tumor.dim[0]; i++) {
             for(e = 0; e < tumor.elements; e++)
             {
                 tumor.data[TSMetric33F_index_map(&tumor, e, i, j, k)] =
                 do_some_computation(e, i, j, k) * tumor.elements_max[e];
             }
   }}}

   tumor.slice = 0.1;
   TSMetric33F_append(&tumor, “Testpat_BD_TumorNeu.f5");

   TSMetric33F_free(&tumor);

   return 0;
}
```

C++

C++ language features allow to simplify and extend the possible functionality of the F5 interface. Using C++ template techniques many structures can be mapped to generic language expressions, which reduced lines of code and simplifies code modification and expansion, see [STR97].

A template basis class *UniformSlice* declared in the file *UniformSlice.hpp* is introduced that is compatible to all data structures defined on an uniform grid. It

stores one data *field* of one *slice*. The basic number type of the tensor elements are determined by the *generic type* of the template (for example *float)*. Only the member functions for reading and writing F5 files have to be implemented especially for each supported *type*. All other functions work with any *type*. The *type* of the template is automatically detected and the according read or write routines are called. It is also stored as a *std::string data_type.*

At the moment the only *generic type* that is fully supported is the *float type*. The user is informed if he tries to use an unsupported *type*. Other *types* can be added with minimal effort by expanding the open and append member functions and adjusting the iterators.

The class supports field the field types scalar, vector and metric33. This has to be specified by an according *std::string* “scalar”, “vector3” or “metric33” as first parameter in the constructor. Class members like number of elements or element names are prepared according to this information.

The members of the class provide the same functionality as the functions described in the *section C* and are not further described. Also a similar macro for enabling index range checking is provided (*US_INDEXCHECK)*.

```
herr_t USFts_callback(F5Path*slicePath, double time, void *user_data);
herr_t USFgr_callback(F5Path*grid, const char*gridname, void *operator_data);
herr_t USFfd_callback(F5Path*field, const char*fieldname, void *operator_data);

template <typename T>
class UniformSlice
{
private:
        double slice;
        std::string grid_name;
        std::string field_name;
        std::string field_type;

        std::string data_type;

        std::vector<std::string> elements_names;
        int elements;

        hsize_t dim[3];

        F5_vec3_point_t origin;
        F5_vec3_float_t delta;
```

```
        std::vector<T> elements_max;
        std::vector<T> elements_min;

        int datasize;
        T*data;

public:
        UniformSlice(std::string field_type, std::string grid, std::string field, double
t, int xdim, int ydim, int zdim);
        virtual ~UniformSlice();

        void printInfo();
        void printElement( int x, int y, int z );
        void printData();
        void minMax();

        void set( T a, int e, int x, int y, int z );
        T get( int e, int x, int y, int z );
        T*getDataPointer();
        T& operator[] (int i);

        int indexMap( int e, int x, int y, int z );
        int validIndex( int e, int x, int y, int z );

        void reallocdata( const int x, const int y, const int z );

        int open( const char* file, double time, const std::string grid,
                const std::string field );
        void append( const std::string file );

        std::string getFieldType();
        std::string getDataType();
};
```

Conclusion

The thesis introduced the scientific file format hdf5 by presenting its concepts, giving a practical guide to use the hdf5 C-library and demonstrating a visualisation of a certain scientific density data-set.

It then introduced the file format F5, which is based on hdf5. The aim of F5, the mathematical motivation and the concept of data organisation were presented. A practical part described the C-library F5 and provided a guide to read and write F5 files.

Finally two specific data structures of tensor based data on uniform grid were implemented in C and C++, supporting different data field types.

These implementations could further be enhanced and extended, for example, by supporting more tensor field types (not only scalar, vector and metric33) or by providing arithmetic functions for element wise addition, or similar.

The F5 library itself also has room for further extensions. For example data *types* based on *doubles* or support of additional high level grid structures. Enhancement of the F5ls or Q5ls to view selected fragments of data contained in a F5 file would also be a nice extension.

Bibliography

[ADC06] Homepage of the Adcirc Development Group, *ADCIRC*, http://www.nd.edu/~adcirc/, 2006

[AMI06] Homepage of Mercury Computer Systems Inc., amira™, http://www.amiravis.com/, Berlin, 2006

[BBH06] Benger, W., Bartsch, H., Hege, H.-C., Kitzler, H., Shumilina, A., and Werner, A, *Visualizing Neuronal Structures in the Human Brain via Diffusion Tensor MRI*, International Journal of Neuroscience 116, 4, pp. 461—514, 2006

[BEN04] Werner Benger, *Tensor Field Visualisation via a Fiber Bundle Data Model*, Department of Mathematics and Computer Science, University of Berlin, 2004

[BP89] David M. Butler and M. H. Pendley, *A visualization model based on the mathematics of fiber bundles*, Computers in Physics 3, 1989, no. 5, 45-51

[BRO98] Manfred Broy, *Informatik Eine grundlegende Einführung, Band 1,* Springer-Verlag, Berlin Heidelberg New York, 1998

[CAC06] Homepage of Cactus, http://www.cactuscode.org, 2006

[CYG07] Homepage of cygwin project, *GNU+CYGNUS+WINDOWS*, http://cygwin.com/, 2007

[FIB06] Homepage of Werner Benger, *The Fiber Bundle HDF5 Librar*y, http://www.fiberbundle.net/, 2006

[GOU03] David A. D. Gould, *Complete Maya Programming*, Morgan Kaufmann Publishers, Elsevier Science, 2003

[GSS07] Homepage by René Heinzl, *Generic Scientific Simulation Environments*, http://www.gsse.at/start/, Vienna, 2007

[H5D06] Homepages of The HDF Group (THG), *DDL in BNF for HDF5*, http://www.hdfgroup.com/HDF5/doc/ddl.html, Champaign, 2006

[H5R06] Homepages of The HDF Group (THG), *HDF5: API Specification Reference Manual,* http://www.hdfgroup.com/HDF5/doc/RM_H5Front.html, Champaign, 2006

[H5U06] Homepages of The HDF Group (THG), *HDF5 User's Guide,* http://hdfgroup.com/HDF5/doc/UG/, Champaign, 2006

[HDF06] Homepages of The HDF Group (THG), *THG Home Page Information, Support, and Software from The HDF Grou*p, http://hdfgroup.com, Champaign, 2006

[LRG06] Homepage of the Relativity Group, Department of Physics and Astronomy, Lusiana State University, http://relativity.phys.lsu.edu/, Baton Rouge, 2007

[MAY07] Homepage of Autodesk Inc., Autodesk® Maya®, http://usa.autodesk.com/adsk/servlet/index?siteID=123112&id=7663079, San Rafael, 2007

[MEM07] Homepage of Werner Benger, *Metamake or Make++ or MCS*, http://www.photon.at/make/, 2007

[MEN07] Homepage of Mental Images Inc., *mental images*, http://www.mentalimages.com, Berlin, 2007

[MM506] Homepage of MM5, MM5 Community Model, http://www.mmm.ucar.edu/mm5/, UCAR, Boulder, 2006

[MVS05] Homepage of Microsoft Coorparation, http://msdn2.microsoft.com/en-us/library/ms950416.aspx, Redmond, 2007

[RAR07] Homepage of win.rar GmbH, *http://www.win-rar.com/*, Bremen, 2007

[STR97] B. Stroustrup, *The C++ Programming Language (3rd edition)*, Addison Wesley Longman, Reading MA, 1997

[SZIP07] Homepage of The HDF Group (THG), Szip Compression in HDF Products, http://hdfgroup.com/doc_resource/SZIP/, Champaign, 2006

[TRO07] Homepage of trolltech, *TROLLTE*CH, http://www.trolltech.com, Redwood City, 2007

[ZLIB05] Homepage of Greg Roelofs, Jean-loup Gailly and Mark Adler, http://www.zlib.net/, 2005