PNI data format concepts and design

Eugen Wintersberger

May 13, 2011

Contents

Chapter 1

Introduction

1.1 Motivation

The success of todays scientific communities depends highly on collaboration. Groups from different departments around the world work together in order to solve a particular scientific problem. This is in particular true for the photon, neutron, or ion-scattering community (PNI-community). Historically, the field of PNI research was dominated by physicists. However, in recent years, scientists from many other disciplines like biology, geology and medicine have discovered PNI scattering methods for their demands. While many physicists are used to write their data evaluation code by themself, users from these new communities usually tend to have a lower affinity to programming and rather rely on standard software tools common within their field of research. Actually every beamline uses a custom data format to stored detector and other data. To make an experiment's data available to such a tool either the beamline scientist must provide software to convert the recorded data into a format understood by the evaluation software or the evaluation software's author provides import routines for each beamline on every facility. Both cases are not very efficient and error prone. But the problem is not only restricted to cases where standard software tools are used for data evaluation. Scientists using their own programs experience the same problem and have to write data importers for every beamline they use for gathering data.

Most of the problems mentioned above can be solved by the definition of a common data standard. Such a standard should provided uniform logical and at some point also semantic access to the data recorded at a beamline or produced by a computer simulation. For standard methods programs would have to look only for a couple of keywords to find the data entries relevant for them. Additional meta-data can be stored in such a standard making data mining long after the data has been recorded possible. In addition a common standard would make it possible to exchange and compare data between different facilities easily.

This paper makes the attempt to develop such a standard which will be refereed to as the PNI-file format (PNIF). The basic design concepts will be presented.

1.2 Requirements to a common data format

The success of a standard highly depends on its acceptance by the user community. Thus, identifying potential user groups and their particular demands is absolutely mandatory for the success of this project. In general four groups of users for PNIF can be identified (see Fig. 1.1):

1. scientific users visiting PNI facilities to get their scientific work done

2. PNI facilities and their staff: beamline scientists maintaining the experiments, IT professionals at the facilities data centers responsible for archiving data and managing remote access to data for users

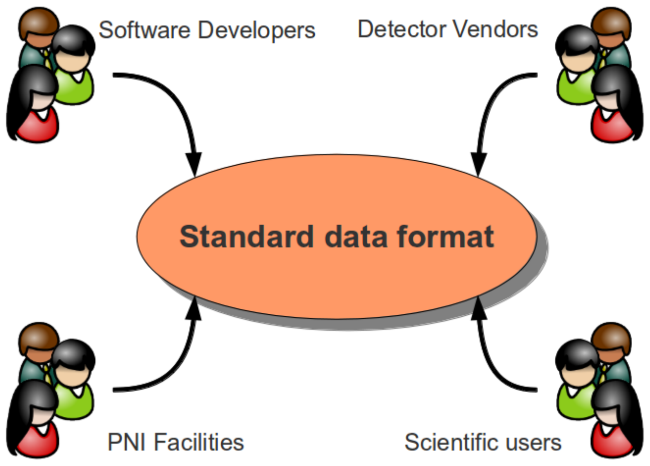


Figure 1.1: The potential users for a new PNI-file format (PNIF) are: beamline users, the large PNI faclities, application developers, and last but not least the detector vendors. Each of them has particular requirements on such a format which must be taken into account during development.

3. programmers writing software for beamlines and data evaluation

4. and finally detector vendors who may want to use such a format to store detector data.

Each of these groups has particular requirements which shall be analyzed in more detail in the subsequent sections.

1.2.1 Scientific users

The most important group of PNIF users is the PNI-user community. From the author's own experience one of the biggest problems for users is the actual abundance of different file formats. Not only the detector vendors with their proprietary binary file formats

This holds not only for detector vendors and their proprietary binary formats but also for the (usually ASCII) file formats used by the data acquisition systems of the different PNI-facilities. In fact each detector vendor uses its own file format for data storage, which requires users to write importers to get the data into their evaluation software.

This can be a tedious job since most of these detector formats are binary and therefore, the binary data stored on disk depends highly on the machine architecture of the computer used to write the data. In addition virtually every beamline uses its own flavor of ASCII data files holding additional information about an experiment. To import this data additional importers are necessary. Although writing ASCII import routines is not that tedious - the formats alter much faster than the detector formats requiring the user to adopt its code many times during its lifetime. A new dataformat should should face this problem and provide a simple and standardized way to access detector as well as supplementary data recorded throughout an experiment. This means that bindings for many common programming languages as well as data evaluation systems (Matlab, IDL, Origin) must be provided to make data import easy for users.

1.2.2 Detector vendors and application developers

On the first glimpse it seems a quite strange that detector vendors are listed as potential PNIF users in the above enumeration. However, one of the biggest problems in accessing data recorded during an experiment is the large number of different binary formats used by detectors to write data to disk. In addition they have nearly the same needs as application developers. The first requirement is that the data format should provide data access at sufficiently high performance. For detector vendors this is important to achieve high frame rates for time resolved measurements, for application developers data IO should not become a bottleneck in the performance of their programs.

Todays storage devices offer a IO bandwidth of typically $100 - 300$ Mbyte/s. From these number it follows immediately that high data rages can only be achieved if the data is compressed before written to storage. The compression algorithms used must show a good scaling behavior and (which is of particular importance for commercial vendors) free of patents.

From these requirements it follows already that PNIF will write binary data and should be implemented as a library in a high level compiler language like C or C++. However the API provided by such a library should be easy to use to keep the implementation effort as low as possible.

1.2.3 PNI-facilities

Although most of the requirements of detector vendors and developers hold for PNI facilities too, there are some extra features they would embrace. Like for developers the format should be easy to use. Bindings to the most commonly used scripting languages (Python,Perl) are necessary to integrate the format in existing data acquisition systems. The library providing access to the data format must be robust in order to guarantee smooth operation of all the systems on the facilities beamlines. This includes a thorough exception handling and memory management.

Actual and future experiments will produce large amounts of data which in many cases cannot be carried home by the users in an easy way (TBytes of data). Thus, facilities must provide storage systems to archive data on site and make it accessible to the users from remote locations. This has certain implications on a standard format. It must be easily extensible in order to adopt to novel experimental techniques but on the other hand-side it must be backward compatible to ensure access to old data from the archive. This has great influence on the design of the library as will be shown in the next chapter. In addition it should be possible to split large data files into smaller portions in order to make archiving on tape storages easier.

1.2.4 Summing up everything

From the above the following requirements for PNIF can be derived

- implemented as a library
- the library should be very robust and thread-safe
- support for many high level and scripting programming environments
- support for many different operating systems (Linux, Windows, OSX, maybe Solaris)
- should be independent of the underlying machine architecture
- provide high performance IO (for instance by providing compressions filters)
- must provide facilities to standardize keywords in the data files
- simple to use (to be attractive for users)
- simple to maintain on long time scales
- must be able to handle large data volumes (splitting large files in smaller portions).

1.3 What do we mean if we talk about a standard?

Before one can start with the development of a *standard* file format the term *standard* must be explained in more detail. When it comes to data files standardization can take place at three different levels

physical layout describes how data is stored on disk (basically one can choose here between binary and ASCII) and how data objects are streamed into bits and stored on the device.

logical design defines which objects are available and how they are related to each other

semantic standardization gives objects a special meaning.

It is of great importance to distinguish between these three levels.

For a better understanding the functionality of each layer will be discussed by an example. Consider therefore an HTML file as it is loaded when your web browser downloads a web site. Lets start with layer 0 - the physical layout of the data. The first thing to recognize is that HTML files are stored as ASCII files. This means that the entire content is represented by printable characters from the ASCII set. Furthermore we know that each object in such a file is encapsulated by and opening and a closing tag. All these things are managed by layer one of the standardization stack described above.

If we look now on an HTML file from the point of layer two - the logical design, we see that each object such a file has a name, optional attributes, and a content. The content itself can consist of other objects. However, there are some rules. For instance nearly all objects within an HTML file reside within the `body` object of such a file. Some attributes are applicable to all objects others are not. The second layer (logical standard) defines which objects can stand for them self and which need a parent object.

In principle the information from the first two levels is enough to read the data and express it as a group of objects in a computers memory. However, finally if we want to render the content on the site on screen each tag must given a special meaning. This is the job of the third, the semantic, layer. This layer is identifies each object with name `p` as a paragraph object and therefore knows what to do with this.

It follows from the above considerations that the first two levels of standardization are mandatory for every data format. Knowing that all information in such a file is a character string allows reading of HTML files. The semantic level is optional. For simply reading a file it is not necessary to know the meaning of an object. However, if you want to further process the data it is.

1.4 How to read

The next chapter, Chapter 2 deals with the logical design of the format. It explains which objects should be available and how they are related to each other. It will furthermore show that it makes sense to split experimental data and log data into separate files. Chapters 3 and 4 discuss additional design problems for datafiles and log files in the PNIF data format. In chapter 5 semantic standardization for PNI-experiments is discussed and its implications on the PNIF file format is shown.

Chapter 2

Basic objects and logical data organization

In this chapter the the basic objects appearing in a PNIF file and their organization with respect to each other will be discussed. The objects presented here are the brickstones from which PNIF data and log files are made of. By saying so we imply already that there will be at least two different file formats: one for data acquired during an experiment or generated by a simulation program and one for log information. The two formats are discussed in more detail in chapters 3 and 4. For many users separate log and data file may sound like a step back in development. However as will be shown in the next section their are good reasons why to provide two files.

2.1 Splitting log and data content into different files

Aside from the data acquired during an experiment or simulation run additional log information must be stored. Log information is data that is not mandatory for the evaluation of the gathered data but can provide useful information for later interpretation of the data. In addition this information can be used to reproduce the experiment's setup.

In the advent of the design process an intense discussion about how log data should be integrated in the data file was held (which is indeed not yet finished). The simplest approach seems to store the logging information directly to the data file (as proposed by several competing file format standards). However, a more detailed analysis of the problem leads to the conclusion that storing log information to a separate file with a distinct format is the more effective approach. We will discuss now the considerations that lead to this conclusion.

If one would store log information in the same file as the *payload* data the first question that needs answering would be: where to store the log data. Data from experiments or computer simulations is organized by runs, scans, or other organizational units. In all cases each of these entities represents a full dataset which can be evaluated by the investigating scientist. In contrary log data is created asynchronously with respect to the measurement or simulation process and is usually organized by time stamps and/or log event issuer. Log events that take place in between two subsequent measurement or simulation runs may can be stored in between their data sections. But where should log data go that appears during such a run? A solution to this problem could be a log section within the data file where all the logging information goes. However, log informations comes not only from the data acquisition system responsible for writing the data file but from many different sources. This would immediately imply that gathering and writing log information should be done by a separate server process (like syslog on Unix like operating systems). But making different processes writing data to the same file is a non-trivial task. Therefore, one might things about delegating the job of log administration to the software instance which is responsible for writing the data files. However this would lead to problems that the performance of the acquisition system can drop in situations where high volumes of log messages appear. This situation should be avoided in particular in cases of experiments producing high data rates.

Furthermore one should take into account that data files can become quite large and are maybe stored on large scale storage system provided by the PNI facilities. In contrary log information is comparatively small and can be stored on the experiment computers or even on the users PC at its home department. If the log data would be stored in the data files the user has to access the large data files residing in the worst case on type archives just to extract some small peace of log data.

There is an additional technical issue. If the log data is stored in a separate file two distinct APIs can be created: one for data files and one for log files. This makes both APIs simpler and easier to use. A user that only wants to write data (for instance a detector vendor) does not need to case about log facilities and someone who wants to write code dedicated log code has no need to learn about the data API.

From all this considerations storing log and *payload* data in different files seems to be a wise decision. The data files will will have the file extension `.pnif` and the log files `.pnil`. In order to make the identification of those files easier MIME-types should be created.

2.2 The basic objects

The possibility to organize the data within the file is an absolute requirement for a modern file format. In Fig. 2.1 an example tree of the fundamental objects is shown. From this figure one can immediately distinguish tow basic types of objects: structural object (File and Node) which can hold other objects as childs and leaf-objects (all others) which hold the data itself. Figure 2.2 gives an overview over all structural and leaf objects available for PNIF. Each object can hold an arbitrary number of additional attributes which are connected directly to the object. As attributes the leaf-objects Scalar, Array, and String are allowed. For more details about attributes see section 2.4. It is clear that each object must have a name so that it can be identified within the tree (see section 2.3). In the next subsections the objects shown here will be presented in more detail. Clearly, in the implementation of PNIF much more objects are available. However, these objects are not of importance for the common user and many of them will not even be visible. For this reason they will not be discussed here.

2.2.1 `Node`-object

The `Node`-object is the most fundamental structural element in PNIF files. Its only function is to act as a container for other node and leaf-objects. It is important to note that `Node` objects are factories for all objects that can reside below the node including other nodes. If nodes are removed two situations can occur: the node is empty (no children) and thus can be removed safely, or the node contains child-objects and an exception is raised. It is suggested to add a `recursive`-flag to the node's remove routine in order to remove a non-empty node. The name of an object must be unique only among the object below a node. It is therefore important to access objects using their full path in order to avoid ambiguity (see section 2.3).

2.2.2 `File`-object

File objects behave in principle like `Node`-objects. However, unlike those they have no parent objects. Therefore, `File`-objects represent the root of the entire object tree. Once instantiated this object can create, open, or close a file stream. The standard behavior should be as follows: if a file already exists it can be only opened and and exception should be raised if the create-method is invoked. If the file does not exist the create as well as the open method can be invoked. In both cases a new file will be created. This behavior should avoid data losses by accidentally creating a new file with an existing name. Like a Node object a File object, in the most general case, is a factory for other node and leaf objects residing below it. To restrict this behavior in order to keep restrictions made by the standard, the file object comes in two flavors: `DataFile` and `LogFile`.

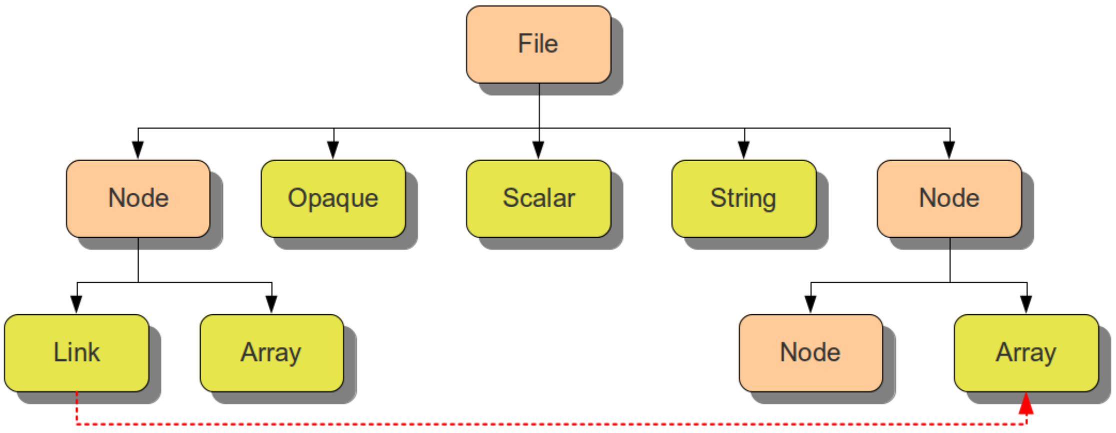


Figure 2.1: The data within the file is organized in a hierarchal structure. Node objects can hold leaf objects like: Arrays, Scalars, Links, Strings, and Opaque streams. In addition each object in the file can hold attributes which further describe the object. All data resides below a *root*-node represented by the File-Node.

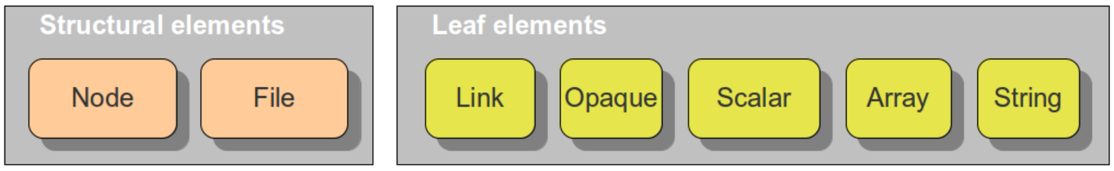


Figure 2.2: From Fig. 2.1 two basic types of objects can be identified: node- and leaf-objects. The former ones cannot hold other objects as children while the latter ones hold the data itself.

These two specializations of the File class do not provide the full factory interface for all objects allowed in PNIF - they can only produce node objects (see chapters 3 and **??** for details).

In addition, each file object should posses a unique ID making it possible to identify a file without relying on the filename (the filename can be mangled by file system operations and other accidents).

2.2.3 `Link`-object

`Link`-objects belong to the leaf-elements in the PNIF object hierarchy. Like symbolic links in a file system they can be used to refer to an object at a different position in the object tree. Links can not only refer to an object within the same file but also to objects in other data files. This makes it possible to access data physically stored in several files from within a single file. Since data and log files use the same objects it should be possible to use links to reference objects in a datafile from within a log file. This capability of the system should overcome the few disadvantages arising from the splitting of log and *payload* data.

2.2.4 `Opaque`-object

`Opaque`-objects behave like file streams. They can be used to hold data that cannot be represented as numeric or character string data. A typical application for such objects would be to store HTML text or image data like JPEG,PNG, or TIFF files. However, all kind of raw data can be stored within these objects. The responsibility for the interpretation of the data lies entirely by the user. Thus this data object might not be as portable as the others but can act as a last resort if everything else fails.

2.2.5 `Scalar`-object

The `Scalar`-object is the simplest numerical data type. It can be used to store a single number. Although this does not seem to be very useful on the first glimpse this object has many applications. Consider for instance the energy of an x-ray beam which must be stored as a single number or the initial motor positions that must be stored in the very beginning of a scanning experiment. For the motors that are not used in the experiment only a single number will be stored - which can be done using the `Scalar`-object. For Scalar objects the physical unit of the number the store is a mandatory piece of information and will be stored along with the pure numerical data. Using this object ensures that all numbers stored have a physical unit associated with them.

2.2.6 `Array`-object

Objects of type `Array` are most probably the workhorse of all data objects. They are used to store numerical data as a multidimensional arrays. Like the `Scalar`-object a string with the physical unit of the data values stored in an array must be given during object creation.

2.3 Accessing objects within the tree

Each object within a tree as shown in Fig. 2.1 can be accessed using a Unix like path. To access a leaf or node object the path has the following structure:

`/nodes/to/the/object/objectname`.

Even attributes can be accessed using the path scheme with a small extension at the end. Consider we want to access an attribute of the object described by the above path, then the following path syntax would give access to this attribute

`/nodes/to/the/object/objectname@attribute`.

The attribute name is separated from the object name by a `@` sign indicating that the following name referes to an attribute rather than an object. Finally, external tools can access data within a file by using the following syntax

`filename:/nodes/to/the/object/objectname`.

2.4 Objects and attributes

Each object in a file can hold an arbitrary number of attributes which are either of type Scalar, Array, or String. Besides user-created attributes there is also a couple of mandatory attributes objects must have in order to satisfy the standard. Unlike Nexus, the PNIF API ensures that all mandatory attributes are available for an object. In other words: the process of instantiating an object requires the user to pass all information including the mandatory attributes a the moment the object is created. From the tree structure of the file format it is clear that every object must have a name. Therefore, there is no need for a mandatory attribute holding the name of an object. The only mandatory attribute each object in the PNIF world must have id `description` which is of type String. It holds a short literal description of every object in the file and is therefore a minimum source of documentation. The numeric objects, Scalar and Array, have both the mandatory attribute `unit`. Which is a String representing the physical unit of the numerical values stored in these objects. The value of the `unit`-attribute should conform to the UDUNIT[1] **REFERENCE MISSING** standard. An Array object must hold in addition an attribute `axes` which is an array of Strings where each entry represents a description (axis) for every array dimension. The Opaque object has a mandatory attribute `mime-type` determining the file type of the information stored in a particular object.

[1]This is also suggested by the NIAC for the Nexus standard.

Chapter 3

Data files

In chapter 2 the fundamental objects were introduced. It is clear that these objects alone are not enough to define a proper data standard. In this chapter, the *rules* and concepts for a datafile are explained in more detail. It was already mentioned that the simple `File`-object is to general for a standard file format since it allows all objects to reside below the root node (represented by the `File`-object). For PNIF data files two simple rules are valid

1. leaf-nodes must node reside directly below the root node of the data tree and
2. the data file must contain all the information required to evaluate the data.

The API can ensure that the first rule is not violated, the second, however, must be managed by the persons writing the IO system for a particular experiment or simulation. To keep the first rule the `File`-object is specialized by a derived type `DataFile`. Unlike the `File`-object which is a factory for all objects in the PNIF universe, the `DataFile`-objects can only act as a factory for `Node`-objects. Therefore, it is not possible to create a Leaf-object below the root node as shown in the following code snippet

```
DataFile *f = new DataFile("test.pnif");
Array *a;
Node *n;

n = f->createNode("dir1","a test node"); //works
a = f->createArray(...); //will not work - method does not exist

//but
a = n->createArray(...); //this will work.
```

The basic structure of a datafile is shown in Fig. 3.1. As mentioned above all leaf-objects reside below a node object below the DataFile-object (the root node of the file). In principle this would be enough to store data an access it again. All objects must at least hold their mandatory attributes (see section 2.4) and thus provide a minimum level of documentation. What is still missing is a facility for a semantic standardization of a measurement or simulation method. The basic concept of such a facility is shown in Fig. 3.2. An object of type `MethodNode` is located directly below the `Node` object. This node contains basically only links to other objects within the data structure above. The idea is now that the names for these links are standardized. This means that all data entries necessary to describe a method are available below the `MethodNode` object via standardized names independent of the name of the original data entry. This gives the beamline staff the freedom of choosing whatever name and node structure they want to use. If, for the experiment performed, a standardized method exists, the data entries can be linked to the method node using the names defined by the standard. The `MethodNode` class has a mandatory attribute named `URL` which referes to a simple XML file containing the names and the structure of

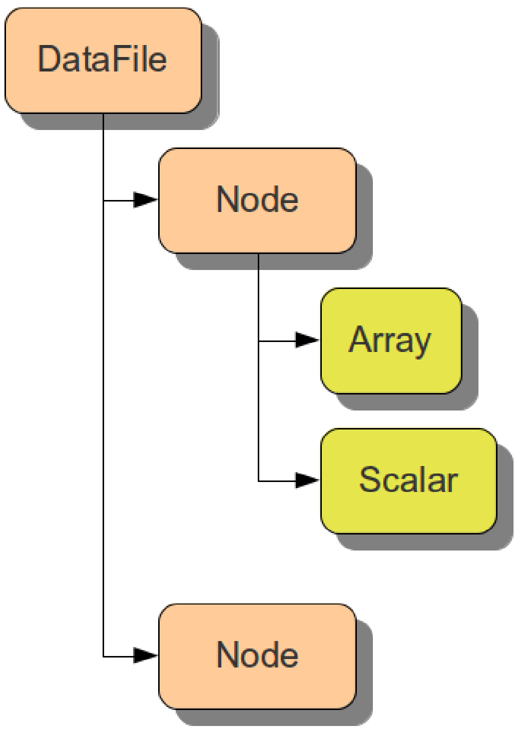


Figure 3.1: The root element of a PNIF data file is the `DataFile` object. However, no leaf object can be stored below this object directly. Instead a `Node` object must be created first which encapsulates all other objects. Below a `Node` the basic objects can be used again.

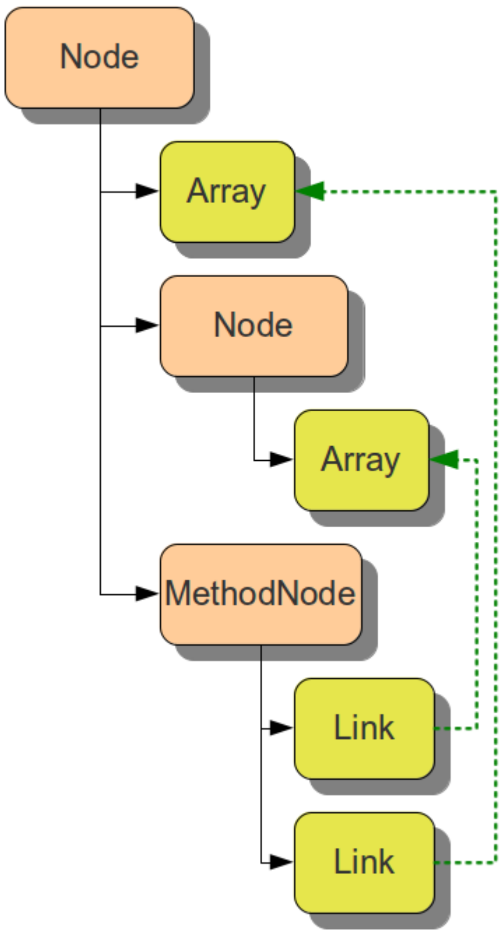


Figure 3.2: A `MethodNode` is a special node used to link all objects holding data necessary for a special method. These links have standardized names and provide therefore a facility for semantic standardization as will be explained in chapter 5.

the entries below a method node. This makes it easy for an application to verify if a method object conforms to a particular standard. For a procedure of how to define a *standard* see chapter 5. It is important to note that the method node is not mandatory. If an experiment does not follow a particular standard procedure it can be omitted. Furthermore there is no limitation in the number of method nodes that appear below the root node of a measurement. If an experiment performs several standard methods during a single run as many method nodes as required can be used.

3.1 A simple example

To see how this works in practice lets have a look on a very simple x-ray reflectivity (XRR) setup. A simple setup for XRR is shown in Fig. 3.3. Only three parameters are required for data evaluation (in the simplest case):

ω the angle of incidence

2Θ the detector angle

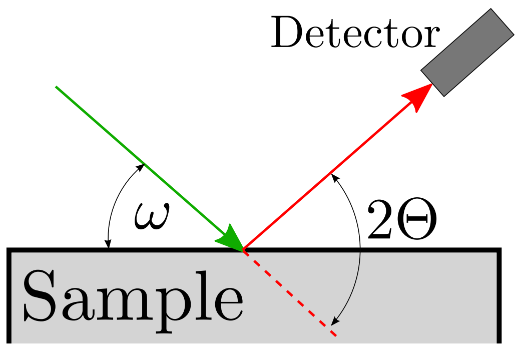


Figure 3.3: The principle setup for an x-ray

Intensity the count rate at each point as recorded by the detector (here we use a point detector).

In Fig. 3.4 the data tree of a single scan in PNIF file is shown. In the upper part of the tree we see the motors and detectors available at the experiment. The names of nodes and arrays are entirely arbitrary as well as the node structure. This can be freely laid out by the beamline staff in a manner that is convenient for them. However, in the bottom part of Fig. 3.4 we see the method group `XR-reflectivity` holding links to the data elements relevant for x-ray reflectivity. The names of these links are no longer arbitrary but are defined by an XRR-standard. An application for the evaluation of XRR data can now search the data file and look for method nodes describing XRR data. If such a program finds an appropriate node it can access the data elements required using always the same names.

Experts in the field of XRR may complain that a lot of additional information, like the wavelength of the beam and some additional quality parameters like the divergence and aperture sizes, is missing. It will be the task of a standardization commission to define all the quantities needed for the description of a particular method. A formal method of how to develop such standards is introduced in chapter 5.

In cases were a quantity mandatory for a particular standard is not available directly from the data acquisition system but can be computed from data provided by the system, the resulting value can be stored directly in the method group. Although, this is not considered to be best practice. Instead, it is recommended to create a special node in the upper part of the tree to store such data in and then use again links in the method node.

Chapter 4

Log files

Log information can be considers as supplementary material to the content of the data file. While the later one contains all the information required to evaluate the data gathered during an experiment or simulation, the log file holds data not absolutely required but maybe useful for finding data or checking for errors in the experiment's setup.

Log data is gathered by a central server - the PNI log server (PNILD) as shown in Fig. 4.1. This decouples the log system from the data acquisition system allowing the log system to run a different physical host than the acquisition system. PNILD receives log messages via network and dumps it to a log file. The entire system is quite similar to the classical UNIX syslog service. One server is responsible for one beamline. The sources for log messages can be other software components at a beamline or a user client. This user client is maybe the really interesting part. It acts as a very rudimentary electronic lab book. One or more instances of this program can run at the beamline network. The user client looks a little bit like a social network client. User entries are presented like posts on such web sites.

The log files can be examined using a log viewer. A log entry is identified by the following string

```
<timestamp>:<source IP>:<source id>:<message title>.
```

The log files use the same data objects presented in chapter 2.

The log files are rotated by the server like syslog does with UNIX log files. This avoids that the logfiles become too large.

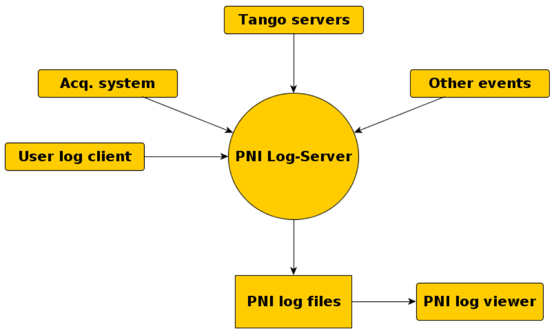


Figure 4.1: Concept of the PNI log system. A central server gathers log events from various sources and dumps them into a log file. The log file itself can be examined with a log-viewer.

4.1 Log file format

The basic structure is presented in Fig. 4.2. `LogFile` acts as a root node for all log entries which are simply appended one after each other below the root-node. The data for each log entry is stored as ASCII. Each entry node holds the data relevant to identifiy a log entry. It might be usefull to create some kind of table of contents in the log file to increase search performance. This can be done offline once the log file is closed due to file-rotation.

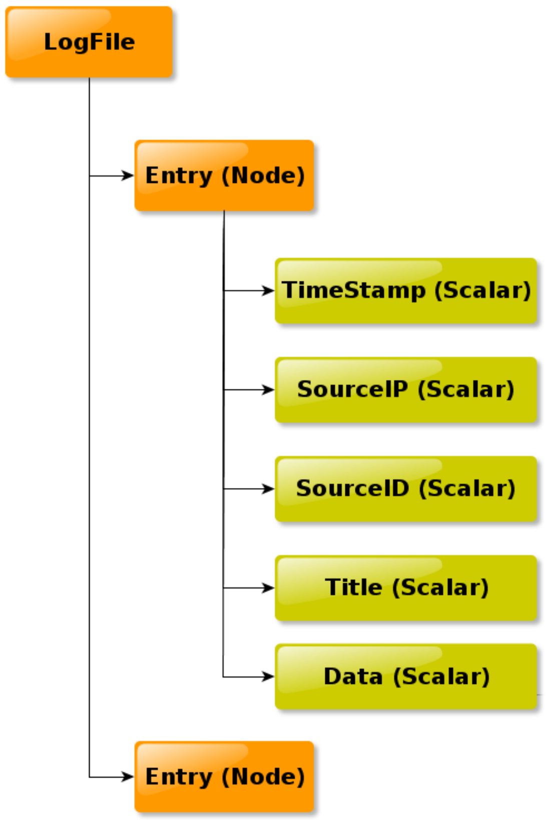


Figure 4.2: Layout of the data tree for a log entry. All entries are stored directly below the `LogFile`-root. Each entry is of type node. Furthermore, all data is stored as strings in a `Scalar`-object. For the technical implementation of this storage format see chapter **??**.

4.2 Logging beamline status

Nearly all beamline scientists that I have interviewed requested a possibility to log the status of their beamline during an experiment. It must be mentioned right at the beginning that logging of the complete beamline status is absolutely impossible. Luckly this is not really necessary. We do not need to log the position of components that have a fixed position because this is documented (hopefully) in the construction plans of the beamline. In addition if someone wants to fully document its beamline state the procedures suggested by the ISO 9000 norms might be a useful hint.

In consequence a full documentation of the beamline cannot be achieved with reasonable effort. However, some things can and should be documented. The idea for PNI logging here is to describe

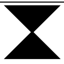
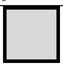
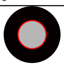
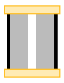
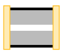
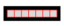
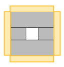
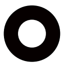
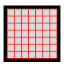
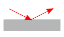
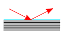
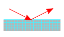
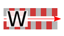
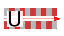
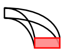
symbol	description	symbol	description	symbol	description
	vent		x-ray window		point detector
	a vertical slit		horizontal slit		(1D) strip-detector
	vertical/horizontal slit		spherical aperture		(2D) area-detector
	mirror		multilayer mirror		crystal
	wiggler		undulator		bending magnet

Table 4.1: A small selection of symbols for the documentation of beamline components along the beam-path.

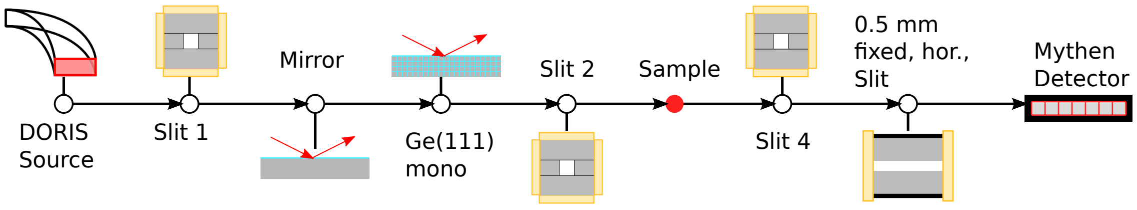


Figure 4.3: This sketch shows a rudimentary documentation of the beamline D4 at DESY. The shown configuration corresponds to an experiment performed in May 2011.

the components along the beam path. Standardized symbols are used for a couple of standard components and placed along a line which represents the path of the beam. It must be mentioned that in this case no absolute positions will be stored but rather the position of the elements with respect to each other along with the distance between the components. Table 4.1 shows a first small sets of such components and Fig. 4.3 a very simple documentation of the optical components of beamline D4 at DESY for an experiment taken place during May 2011.

Chapter 5

Standardization of methods and views (semantic standardization)

5.1 A little math - how to define a standard at all?

The primary problem with defining standards is that once defined they should hold for a long time without changes. Such changes usually appear in the very beginning because necessary quantities have been forgotten in the definition. In order to reduce this risk a formal procedure of how to define a standard for PNI experiments will be presented here.

Consider a vector $\mathbf{q}$ of dimension n with elements q_i and $i = 1, \ldots, n$. The q_i denote independent quantities from which a scientist can draw conclusions about a system under investigation. The term independent here means that $\forall i, j = 1, \ldots, n q_i \neq q_i(\ldots, q_j, \ldots)$. In other words the q_i are mutually independent of each other. In addition we define an additional vector $\mathbf{p}$ of dimension m with the property that $\forall i = 1, \ldots, n \exists j \in \{1, \ldots, m\} : q_i = q_i(p_j)$. $\mathbf{p}$ is called the parameter vector of $\mathbf{q}$. We can define now a *method* as a vector $\mathbf{M}$ of $m + n$ quantities with

$$\mathbf{M} := (q_1, \ldots, q_n, p_1, \ldots, q_m) \,. \tag{5.1}$$

Therefore, the first step in defining a method is to define the quantities in $\mathbf{M}$. Unfortunately in many cases the members of $\mathbf{M}$ are not accessible directly in an experiment. Instead, other quantities can be obtained from an experiment which can be used to compute the components of $\mathbf{M}$. Thus we define a vector $\mathbf{E}$ of dimension k holding all the experimentally accessible quantities. Now let us assume that there exist a function F defined with

$$\mathbf{Q} = F(\mathbf{E}) \text{ and } \mathbf{E} = F^{-1}(\mathbf{Q}). \tag{5.2}$$

From the point of defining a standard method, the situation looks now as follows: the methods defines quantities $\mathbf{M}$ which are used to draw scientific conclusions from - $\mathbf{M}$ is what we would like to have in the end. $\mathbf{E}$ describes quantities accessible during an experiment and F a function converting $\mathbf{E}$ to $\mathbf{M}$. From this considerations it follows that a method can be defined by the three quantities: $\mathbf{M}$, $\mathbf{E}$, and F.

On a first glimpse this approach seems unnaturally complicated. However, it has a big advantage: instead of standardizing a huge amount of quantities hoping that they are sufficient to describe a method, this approach clearly determines which quantities are necessary (the input parameters for F). Besides, F acts as a proof of concept - it can be evaluated only if all quantities are available.

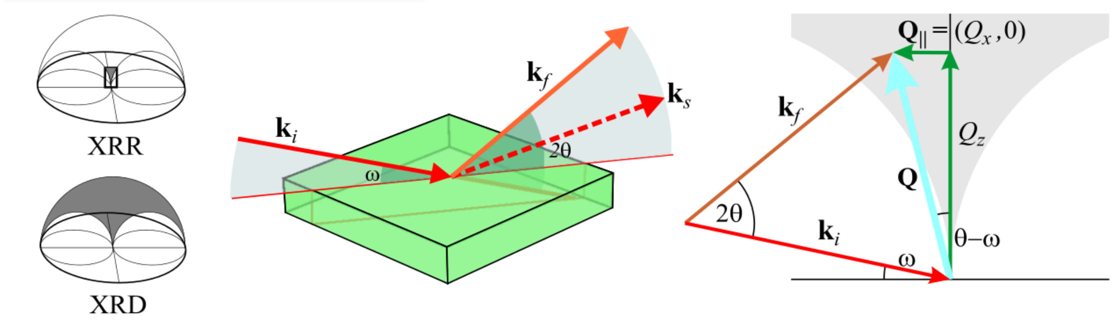


Figure 5.1: Experimental and **Q**-space setup for coplanar x-ray reflectometry and diffraction. (Figure from the PHD-thesis of Julian Stangl).

quantity	unit	description
ω	$(^\circ)$	the angle of incidence
2Θ	$(^\circ)$	the detector angle
$I(\omega, 2\Theta)$	(count-rate)	the intensity recorded by the detector
τ	(sec.)	the exposure time of the detector
monitor	(a.u.)	a monitor counter that represents the fluctuations of the primary beam intensity
λ	(Å)	the wavelength of the x-ray beam.

Table 5.1: Experimental quantities for XRR and XRD experiments required to obtain a reciprocal space map.

5.2 Some example standard-methods

5.2.1 Coplanar x-ray diffraction (XRD) and reflectometry (XRR)

XRD and XRR differ only in the magnitude of the angle of incidence ω as shown in Fig. 5.1. The quantities recorded during an experiment are shown in Tab. 5.1.

These quantities are components of the vector **E** as defined above. The target quantities for bot methods to evaluate data are q_x, q_z, and $I(q_x, q_z)$. Which are the components of the method vector **M**. What is now missing is the function that transforms **E** into **M**. These formulas are given with

$$I(q_x, q_z) = I(\omega, 2\Theta)/\tau/monitor \tag{5.3}$$

$$q_x = 2k \sin\left(\frac{2\Theta}{2}\right) \sin\left(\omega - \frac{2\Theta}{2}\right) \tag{5.4}$$

$$q_z = 2k \sin\left(\frac{2\Theta}{2}\right) \cos\left(\omega - \frac{2\Theta}{2}\right) \tag{5.5}$$

with $k = 2\pi/\lambda$.

5.2.2 Gracing incidence diffraction

The physical situation for a GID experiment is shown in Fig. 5.2. Like for XRR and XRD the target quantity for this method is a reciprocal space map $I = I(q_x, q_y, q_z)$. The experimental quantities are shown in Tab. 5.2. To come from the experimental input data vector **E** to the

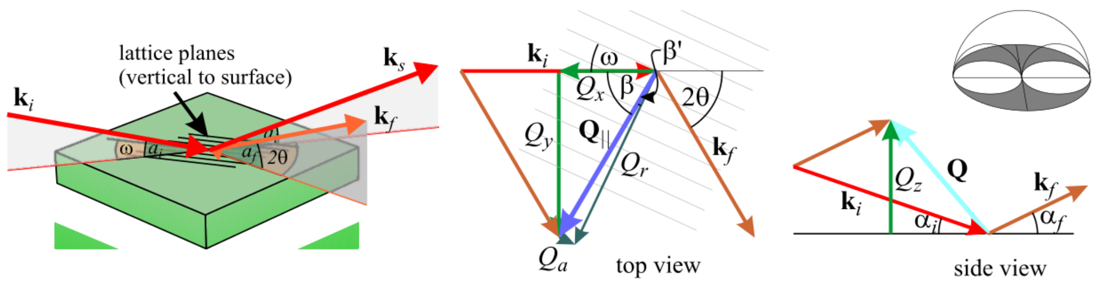


Figure 5.2: (figure from the PHD-thesis of Julian Stangl).

quantity	unit	description
ω	$(^\circ)$	azimuthal angle with respect to surface
2Θ	$(^\circ)$	the detector angle
α_i	$(^\circ)$	angle of incidence
α_f	$(^\circ)$	exit angle with respect to surface
$I(\omega, 2\Theta)$	(count-rate)	the intensity recorded by the detector
τ	(sec.)	the exposure time of the detector
monitor	(a.u.)	a monitor counter that represents the fluctuations of the primary beam intensity
λ	(Å)	the wavelength of the x-ray beam.

Table 5.2: Experimental quantities required for GID experiments to obtain a reciprocal space map.

method's target vector $\mathbf{M} = (I, q_x, q_y, q_z)$ the following formulas can be used

$$I(q_x, q_y, q_z) = I/\tau/monitor \tag{5.6}$$
$$q_x = -q_{||}\cos(\beta) \tag{5.7}$$
$$q_y = -q_{||}\sin(\beta) \tag{5.8}$$
$$q_z = k\,(\sin(\alpha_i) + \sin(\alpha_f)) \tag{5.9}$$

with

$$q_{||} = k\sqrt{\cos^2(\alpha_i) - 2\cos(\alpha_i)\cos(\alpha_f)\cos(2\Theta) + \cos^2(\alpha_f)} \tag{5.10}$$
$$\sin(\beta) = \frac{k\cos(\alpha_f)\sin(2\Theta)}{q_{||}} \tag{5.11}$$

5.3 Terminology - a physical access

In the previous sections a formal procedure to define a standard was developed. However, we still need a terminology to define quantities necessary for the description of a method. This section deals with this topic by trying to take a very physical point of view. In order to handle a wide range of experimental methods, a very general case of a PNI-experiment will be considered.

5.3.1 The physical situation

The simplest experiment that can take place at a PNI facility is shown in Fig. 5.3. In this experiment a beam of probe particles (photons, neutrons, or ions) impinges on a target (usually the sample under investigation). A detector records scattered particles or particles emitted by the target as a reaction on the incident beam. In the most general case there can be even multiple beam sources and detectors available during an experiment. However, in nearly all cases only one

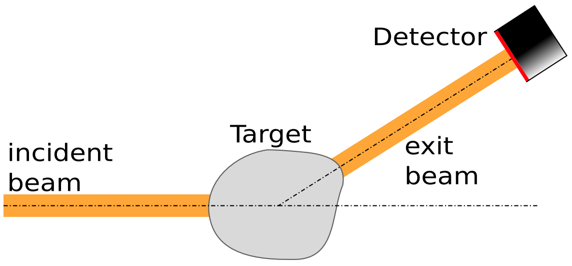


Figure 5.3: The simplest scattering experiment that one can imagine

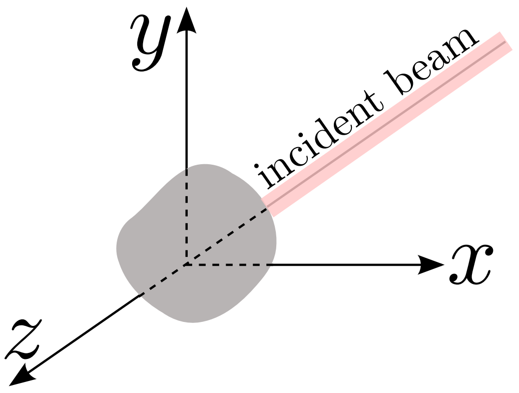


Figure 5.4: Cartesian coordinate frame defined around the sample. The z-axis points along the direction of the incident beam.

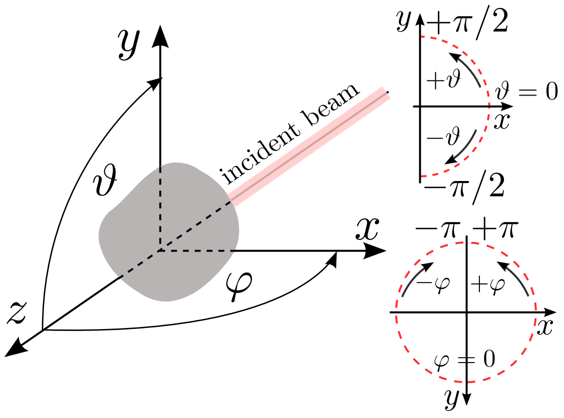


Figure 5.5: Spherical coordinate frame around the scattering target. φ runs from $-\pi$ to π and ϑ from $-\pi/2$ to $+\pi/2$.

sample (target) is under investigation at a time. Therefore, the sample can be considered as a kind of fix point in the entire system.

Data can be gathered at a single or several points in time. In the later case usually experimental parameters are varied between two readouts of the detector data. Theses parameters might be the spatial detector position, the angle under which the probe particles impinges on the surface of the target, the energy of the beam, and many more. The time intervals between two readouts are not necessarily constant. It is therefore important to record a time-stamp at each point of data acquisition. In general such a procedure can be considered a two step process:

5.3.2 Coordinate frames

For the choice of a coordinate frame is a tricky thing which depends on the concrete application. In particular two applications can be identified

1. to define directions and positions in order to evaluate the experimental data

2. to describe the position of components at the beamline.

Unfortunately, the requirements on the coordinate frame for this tow applications are in some points contradictory. Since the focus lies on the evaluation of experimental data this will also determine the choice of the coordinate frame. For data evaluation usually we need all positions with respect to the direction of the incident beam and the sample. Therefore a proper choice for the coordinate system will be based on these two parameters. The origin is defined by the center of the sample (or the center of rotation of the sample). The coordinate frame used here was basically taken from the Nexus documentation. Figure 5.4 shows the Cartesian frame used to define coordinates. Figure 5.5 the spherical frame and its angular ranges. Cartesian coordinate can be computed from the spherical ones by using

$$x = R\cos(\vartheta)\sin(\varphi) \tag{5.12}$$
$$y = R\sin(\vartheta) \tag{5.13}$$
$$z = R\cos(\vartheta)\cos(\varphi). \tag{5.14}$$

5.3.3 Time

It is clear the at some point in time each recorded data point needs to have a kind of time stamp. The question which raises is what source one should to use as a time marker. The best solution is most probably to use UTC time which counts seconds starting from the 01/01/1970. Most computer systems provide a timer for UTC which has a much better resolution than one second. In cases where timers with higher resolution are required external hardware solutions are a better choice.

The fundamental advantage of using UTC over common date-time-strings is that they are independent of the time-zone. This gives us the possibility to express a point in time when a certain data point has been recorded also in the local time zone of the location where the user is evaluating the data. To get the local time of the facility where the experiment was performed can be achieved by storing its local time zone as a string.

5.4 Views

Views shall make data visualization easier. They explain how data is mapped onto spatial positions. Three different views are supported for now: regular grids, rectilinear grids, and unstructured points. The used terminology is borrowed from Kitwares Visualization ToolKit (VTK). In this section we will explain how such data structures are represented by PNIF data objects. An arbitrary set of datasets is associated with the positions defined by the grids or a point set.

5.4.1 `Regular Grid-view`

A regular grid is the most simple data structure for data visualization. Let d be the number of dimension of the grid and n_i with $i = 0, \ldots, d-1$ the number of elements along each dimension of the grid. With each dimension an axis of equidistant points is associated. For such an axis only three numbers must be stored to reconstruct the position of each grid point along this very axis: x_i^0 the starting value of the axis, dx_i the steps size along the axis, and n_i the number of steps. See Fig. 5.6 for a sketch of an unstructured grid (left panel). The object structure for such a grid is shown in Fig. 5.7.

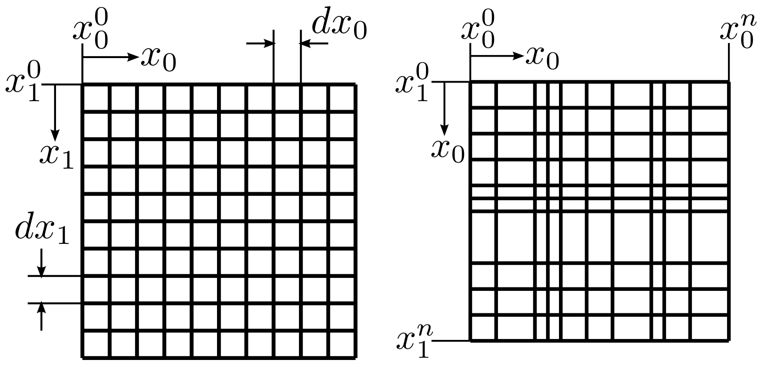


Figure 5.6: Regular (left panel) and rectilinear (right panel) grids differ only in the fact that for the later ones the points along the axes must not be equidistant.

5.4.2 `Rectilinear-Grid`-view

A rectilinear grid is in principle the same than a regular grid. The only difference is that the values along one or more axes are not equidistant. Thus for each axis all values must be stored. An axis is therefore described by n_i numerical values. In Fig. 5.6 (right panel) a rectilinear grid is shown in 2 dimensions. Figure 5.7 shows the object structure of such a view.

5.4.3 `Unstructured-Points`-view

`Unstructured-Points` describes a set of points at arbitrary positions in space. Such a point set is the most general way of storing data. For each data value the point positions must be stored. See Fig. 5.7 for the object structure for a unstructured point view.

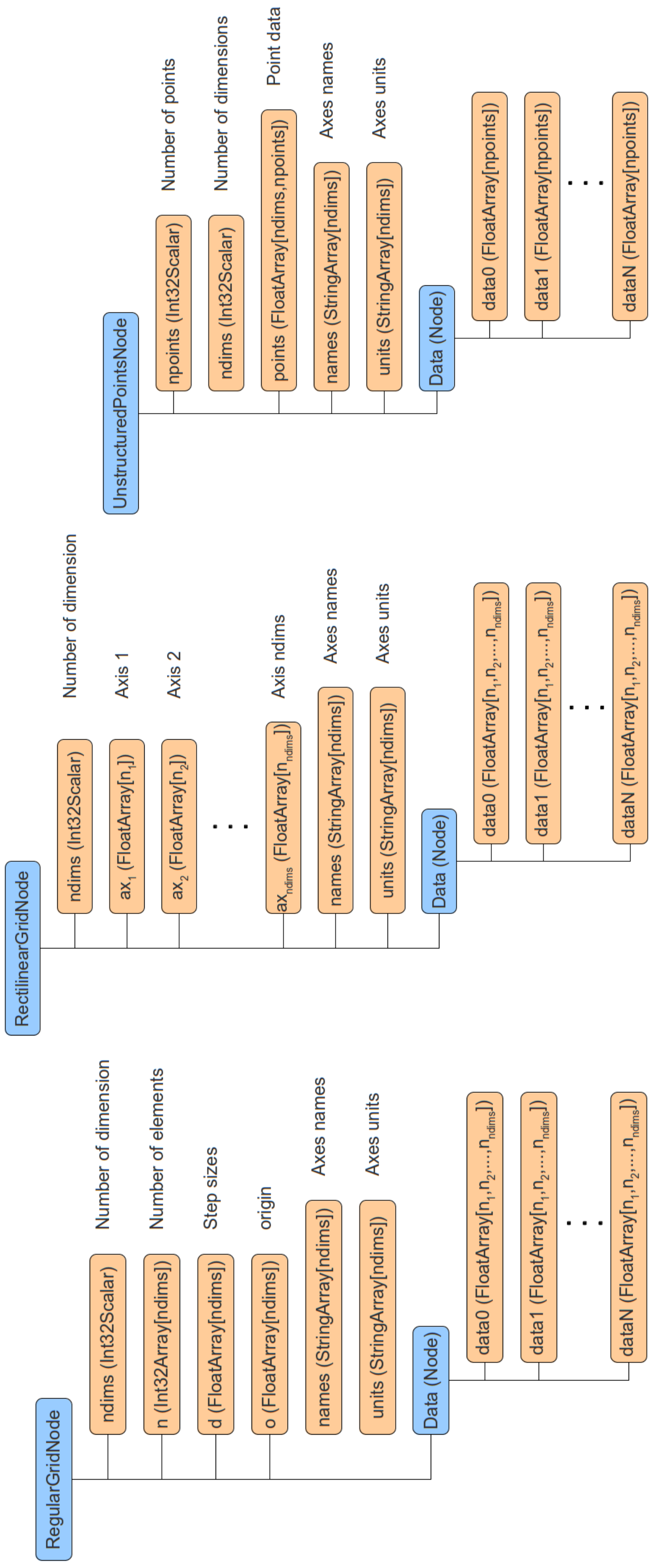

RegularGridNode
ndims (Int32Scalar)
Number of dimension
n (Int32Array[ndims])
Number of elements
d (FloatArray[ndims])
Step sizes
o (FloatArray[ndims])
origin
names (StringArray[ndims])
Axes names
units (StringArray[ndims])
Axes units
Data (Node)
data0 (FloatArray[$n_1,n_2,\ldots,n_{ndims}$])
data1 (FloatArray[$n_1,n_2,\ldots,n_{ndims}$])
dataN (FloatArray[$n_1,n_2,\ldots,n_{ndims}$])
RectilinearGridNode
ndims (Int32Scalar)
Number of dimension
ax_1 (FloatArray[n_1])
Axis 1
ax_2 (FloatArray[n_2])
Axis 2
ax_{ndims} (FloatArray[n_{ndims}])
Axis ndims
names (StringArray[ndims])
Axes names
units (StringArray[ndims])
Axes units
Data (Node)
data0 (FloatArray[$n_1,n_2,\ldots,n_{ndims}$])
data1 (FloatArray[$n_1,n_2,\ldots,n_{ndims}$])
dataN (FloatArray[$n_1,n_2,\ldots,n_{ndims}$])
UnstructuredPointsNode
npoints (Int32Scalar)
Number of points
ndims (Int32Scalar)
Number of dimensions
points (FloatArray[ndims,npoints])
Point data
names (StringArray[ndims])
Axes names
units (StringArray[ndims])
Axes units
Data (Node)
data0 (FloatArray[npoints])
data1 (FloatArray[npoints])
dataN (FloatArray[npoints])

Figure 5.7: Object structure for a regular, a rectilinear grid, and a set of unstructured points.