`text` files *vs.* SQL databases for running and analysing experiments

Håkan T. Johansson
`f96hajo@chalmers.se`

9th November 2013

Contents

1 Introduction **1**

2 Types of data **2**
2.1 Raw data (events; list-mode) . . 2
2.2 Processed data (events) 3
2.3 Source code 3
2.4 System topology / DAQ setup 3
2.5 Slow-control settings 3
2.6 Slow-control logs 3
2.7 Calibration parameters 4

3 Players — Actors of the drama **5**
3.1 Relational (SQL) databases . . . 5
3.2 `text` files 5

4 Req. features and recurring tasks **6**

5 SQL propaganda **7**

6 `text` file solutions **8**
6.1 Handling 8
6.1.1 Viewing 8
6.1.2 Modifications 8
6.1.3 Even faster access 9
6.1.4 History keeping 9
6.1.5 Different versions 9

7 Remarks **9**
7.1 Policing work-flow 9
7.2 No XML 10
7.3 Trees, not tables 10
7.4 Reliability 10
7.5 Rants 10
7.6 `text` files in use 10

1 Introduction

At seemingly regular intervals, the discussion about how calibration parameters and other meta-data used in experiment analysis should be stored and handled comes up. As the "database vs. text-file" dispute. By caching the arguments of both camps, this document aims at bringing any further such discussions to a next level, by avoiding the tedious rehashing of basic arguments.

We first describe the kinds of information (data) that is handled when preparing, performing and analysing nuclear physics experiments. Then follows a short overview of the two approaches, to have a baseline of terms. Thus armed, we continue by enumerating a number of desired features and common tasks from the point of view of a user (and developer). Then the arguments of the two approaches are presented.

Anticipating the conclusions, the reader should be cautioned: While this document strives to be fair, it is not intended to be unbiased. For the task at hand, the author(s) does not refuse to use SQL databases out of being cheap, lazy or ..., but because a better solution exists, *both* with respect to format and tools: plain text files. This document argues why.

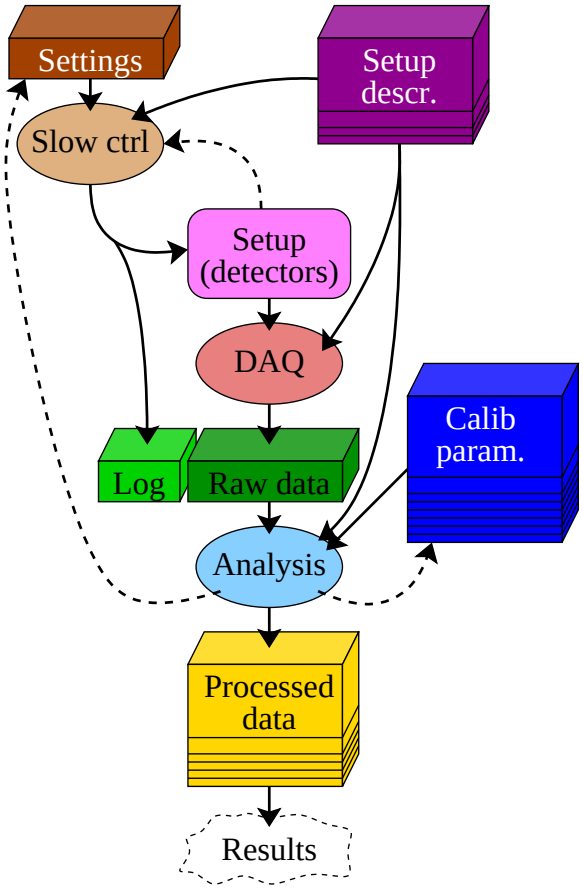

Figure 1: Connections between the different kinds of data / information involved in nuclear physics experiments, illustrated as 3d-boxes, with vertical stacking marking multiple versions during analysis. Processes are shown as ellipses, and the actual setup as a round-corner box.

2 Types of data

Quite a few kinds of information are used and generated in experimental (nuclear) physics, see Figure 1. They are characterised in this Section. The first few will not be discussed further, but are included for completeness, and as they introduce some useful tools.

2.1 Raw data (events; list-mode)

The raw digitised data collected by the data acquisition system (DAQ) during an experiment is the essential outcome of a beam-time. It is generally stored in a streaming fashion in raw binary files. The reason for this is twofold:

1. Keeping the DAQ simple! The binary data stored is most often copied unaltered, directly from the data acquisition modules (DAMs).

2. The sheer size of the data. Unsurprisingly, experiments tend to generate data with amounts on the order of, or well exceeding, the size of single units of state-of-the-art current capacious storage media.[1] Either it was tapes in the last century or early in the current, or hard drives today.

We do not believe that this storage scheme is under discussion or challenged. We however note that sometimes suggestions to use the storage format of processed data (more explicitly: ROOT-files; see next section) are put forward. It is advised that such ideas are in stark contradiction with point 1 above — it includes unnecessary complexity in the DAQ. A faulty DAQ can render the collected (then also mangled) data of an experi-

[1] There are of course exponential variations in both directions: data sizes swelling even further challenging or into the more managable realms.

ment useless. For further brain-unwashing, see Section 5.2.2 in [1].

It should also be noted that raw data, once collected, never changes. Altering them once written on file would open a door to scientific misconduct.

2.2 Processed data (events)

Unpacked and partially treated raw data is today generally stored in so called trees in `.root`-files, in-between processing by programs using the ROOT framework [2].

The `.root`-files are effectively a kind of database. A custom one, designed to efficiently be able to handle the vast amounts of data streaming through analysis, organised by column-wise variables. The events are rows in this thought matrix. This scheme is also undisputed — if, then possibly by other equivalent tools and frameworks; a decade ago, it could have been `hbook` files for use with the ROOT ancestor CERNLIB/PAW [3].

2.3 Source code

The source of the programs used during all stages of an experiment is professionally kept in version control systems (VCS), e.g. GIT or CVS, and handled using each user-/developer's favourite editor(s). This worldwide common practice is only mentioned here as it introduces an extremely well-tested category of software: version control systems. While also capable of keeping track of binary blobs of data, they excel at handling the history of line-oriented text files. Shepherding of source code also implies acquaintance with the likewise standard armoury of UNIX text tools: LESS, GREP, DIFF, SED/AWK, PERL/PYTHON, ... As well as an understanding of the underlying VCS.

We also consider shell scripts, ROOT macros, and other small pieces of code as a part of the heading "source code".

2.4 System topology / DAQ setup

The description of the setup is a vital component during an experiment. Mapping information is needed by the control system (slow control) to operate detectors and other equipment. Some parts of the configuration are used by the DAQ to read the correct modules, producing the raw data files. The more detailed information is needed by on- and off-line analysis to map the detector channels when unpacking the raw data before/during analysis.

The actual information used by the DAQ needs to be retained, but will not change after use (the beam-time). The detailed mapping information in principle also does not change, but may have to be modified if one during analysis discovers that the information entered was incorrect (mismatch between map and reality at the time of experiment).

2.5 Slow-control settings

Run-time adjustable parameters of the setup are handled by the "slow control". This can be high voltages, magnetic fields, slit positions, gas flows, etc. Some setup parameters may be assigned (optimised) iteratively by analysing data during the setup-phase of the experiment, to e.g. gain match elements of segmented detectors. Control parameters are often adjusted by the user during experiment setup, but preferably kept constant during production runs.

2.6 Slow-control logs

The actions taken by the slow-control system need to be logged, both for run-time debugging and later analysis. It is also, for the same

purposes, useful to have recordings of various (slowly) varying parameters of the setup. The number (storage size) of such parameters can easily be more than the amount of event-wise recorded data is, thus also for that reason recorded more seldomly.

2.7 Calibration parameters

The processing of raw data to obtain higher level quantities is called reconstruction. The calculations and conversions involved use several layers of parameters going from one step to the next. Determining the parameters is called calibration, and can often be seen as a sort of inverse reconstruction[2], dead-ending at the level where the particular sought-after parameters are used.

These two steps are a major part of the analysis work, and the handling of calibration parameters is the core subject of this document. It certainly is the kind of "metadata" most people come into direct contact with. Depending on counting fashion, there are three or four dimensions to these parameters (see Figure 2):

- Parameter **type** and **index**:
  - The **type** of a parameter depends on the quantity it describes. Some parameters are scalars, while other might logically be vectors, e.g. describing a spline curve correction. Other still may come in pairs, e.g. correlated offsets and slopes. Yet another might choose from a set of distinct options: `method1`, `method2`, `method3a`, `none`...

    Note how the different layouts bring heterogeneity to how various parameters are best viewed, stored and handled.
  - The **index** (usually including a detector name) tells with which particular detector channel a parameter is associated. This dimension grows as the segmentation of detectors increase.
- **Time** during **experiment**. Due to wanted and unwanted changes, the calibration parameters may vary during the experiment. Drifts due to temperature changes, varying leakage currents and other gradual damage may force the physicist to use several sets of parameters, especially for the high-resolution final analysis of an experiment.

  The ranges in time where a set of parameters is valid is however not common for the entire setup — different parts of the setup change differently. Meaning that a "separation of controlling vari-

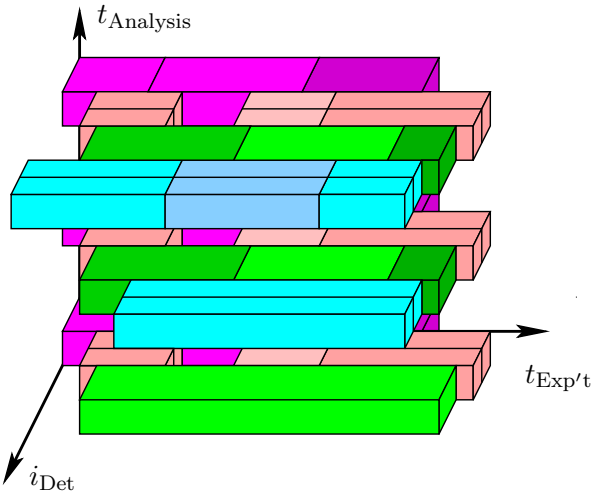

Figure 2: Dimensions of calibration parameter storage. The horizontal line, $t_{\mathrm{Exp't}}$, represents time during the experiment and out of the paper comes detector channels, i_{Det}. The vertical layers, t_{Analysis}, pile up as analysis progresses by refining parameters (shown exploded). (Concept from D. Bertini.)

[2] Inverse, as the same equations are used, but other variables are solved for.

ables" between this and the previous indexing dimension(s) cannot be performed.

- **Time** during **analysis**. As analysis work progresses, as the setup is better understood, the calibration parameters used vary — they are the language in which the improvements are quantified.

  This dimension is very akin to software (version control). Most of the time one wants to use the latest — presumably best — set of calibration parameters. But occasionally one wants to use the ability to go back in time and analyse using an earlier best set of parameters.

  For rules of good scientific conduct it should be possible to track back wrong calibrations and recover/reproduce spectra seen online even after updating calibration files or procedures. Example: someone shows a very nice result in a conference as preliminary but it turns out to be an artifact; to show no ill intent one should be able to track the root of the mistake. This example is extreme but typically a wrongly calibrated detector could give a peak online and justify a longer beam-time, making backtracking important.

One further dimension come into play during volatile stages of analysis:

- **Per-user** calibration parameters. Depending on the storage scheme used, this may need to be explicitly catered for.

3 Players — Actors of the drama

This section shortly describes the two solutions. For some background, the reader may want to look at the Wikipedia articles for "SQL" and "lexical analysis" and "parsing". They are probably equally scary.

3.1 Relational (SQL) databases

SQL stands for "structured query language", and is a language for interacting with relational database (RDB) engines. The actual storage of the information is handled by the database engine, or longer: relational database management system (RDBMS). The information is logically handled in one or several tables, with columns as fields and rows as entries. Information is inserted and retrieved using SQL queries, using field entries as keys.

As RDBMS one uses either a commercial or open-source solution. The custom-produced code needed is the interfaces between DAQ / slow-control / analysis and the RDBMS that uses SQL queries. One also needs to develop tools that use/hide the SQL interfaces to let the user insert and inspect the RDB information in efficient ways.

Centralised - can they be decentralised (still allowing merging)?

History keeping? (edit log?) versioning?

3.2 `text` files

With `text` files we mean human-readable information structured with a well-defined syntax. Think of `C` code (or better: structures; as in declarative syntaxes like QML, CSS, . . .), with elements added to the syntax to aid each use case; see Listings 1 and 2. While a structured text file also can be regarded as a *database*, we reserve that word for RDB schemes in this document.

With the resemblance of source code, the full arsenal of tools to handle source are also available for the human interaction with the `text` files. Where the soup sticks to the spoon, and custom code is required, is for the

```
// Split the signal into two, one
// direct for timing to a CFD, the
// other delayed for integration
// going to a QDC.

SPLIT_BOX(r13c1s1)
{
 in1_8:   "N101" <- , rNP1c21/out1_8;
 in9_16:  "N102" <- , rNP1c22/out1_8;

 e1_16:   "N11"  -> , r13c2s18/in0_15;

 t1_8:    "N11"  -> , r12c2s1/in1_8;
 t9_16:   "N12"  -> , r12c2s2/in1_8;
}
```

Listing 1: Example of a structured, human- and machine-readable, `text`-file, describing part of an experimental setup.

machine reading of the information. Anyone who has written manual parsers using `scanf` and the like, know the associated pains and bug-proliferation this approach induces.

Fortunately, tools to handle this task exist: lexer and parser generators, e.g. `flex` and `bison`. Given a description of the syntax, they generate code to lexically analyse and parse the text input. With small fragments of appropriate (`C`) code in the syntax description, the produced program will at runtime fill data structures with the interpreted information. These structures are then directly usable by the DAQ / slow-control / analysis code. The generated lexer/parser also provide error-handling, e.g. point out locations of syntax errors.

Throughout this document, we assume that `text` files are used together with lexer and parser generators.

```
// Global detector synchronisation

TIME_SYNC_OFFSET((POS,1),(  91.5));
TIME_SYNC_OFFSET((N  ),  (-118.81));
TIME_SYNC_OFFSET((TFW),  ( -78.7));

// More adjustments (additive)

TIME_SYNC_OFFSET((TFW),  (-6.3+0.315));

// Further selective corrections,
// per blocks of raw data files

LT_RANGE("64: 2: 5: 1: 222 ",
         "64: 2: 5: 1: 232 ") {
  TIME_SYNC_OFFSET((TFW),(0.17));
}
LT_RANGE("64: 2: 5: 1: 233 ",
         "64: 2: 5: 1: 236 ") {
  TIME_SYNC_OFFSET((TFW),(0.13845));
}
```

Listing 2: Example of calibration parameters, with validity range limitations. (Slightly edited to fit two-column printing.)

4 Required features and recurring tasks

There are a number of features that a helpful meta-data handling system has to exhibit, given by common tasks and work-patterns.

R1 Easy inspection. It must be easy for a user to see and inspect the parameters / settings and values that are used. Note that one must be able to find out both which parameters are applied, and equally easily find which ones are *not*. I.e. obscure overlay features which cannot easily be followed must be avoided.

R2 Easy modification. It must also be fairly easy to add or modify parameters and values. Also in batch, e.g. from (semi)-automatic calibration programs.

R3 Easy debugging. It must also be fairly easy to selectively enable or disable sets of parameters, or use alternative tables without destroying the temporarily unwanted bunch. This kind of operation is common during debugging, but should leave no or little trace in the history (i.e. along t_{Analysis}), as the copious test changes otherwise clutter the interesting development.

R4 Simple interface in the program code. Program developers need easy-to-use routines and programming patterns to access the parameters. Preferably relieving them of having to perform (i.e. program) recurring functionality — thus avoiding lots of copy-paste code. This requirement, in particular its fulfilment, can of course be interpreted very subjectively.

Easy access is required not only for the big full-featured analysis codes, but is equally or even more important also for small temporary scripts (hacks) to investigate parts of a parameter set. This restricts the use of complex access schemes, or calls for several but equally powerful ways of accessing the information.

R5 Global and local history. For an experiment collaboration as a whole to be able to conduct itself scientifically, it must be possible to keep a history of used parameter (values). It must likewise be possible for single (or groups of) users to keep local histories during their development, without jumbling the global history with dead-end attempts. The global history need not be linear, it can also contain branches mirroring local development paths.

R6 Assure possibility to analyse any experiment 10 years after last paper written (proper scientific practice following e.g. BNBF regulations). We know of experiments which could not be looked at again due to missing logbook and setup descriptions.[3] Whichever solution is chosen, it should help to avoid this situation.

5 SQL propaganda

This section should present the features, advantages and arguments for using RDB databases to store calibration parameters and other meta-data of an experiment. As the author is not well versed in RDB databases, SQL or otherwise, the list is short and incomplete?

The reader is encouraged to help extend the document!

- The centralised database-keeping enforces the participants of a collaboration to use established sets of calibration parameters for analysis.
- Access to needed parameters is fast.
- Declaring the time range of validity of a parameter is easy by tagging of entries in the database tables.
- Solutions exist allowing interfacing of simulation/analysis/slow-control. This should allow more realistic simulations of a beam time by taking into account e.g. the failing of a PMT and the impact in the final result.

Help! This cannot be it??

[3] In addition to DAQ/slow control logs and experiment logs, also analysis logs (per user) need to be retained and available.

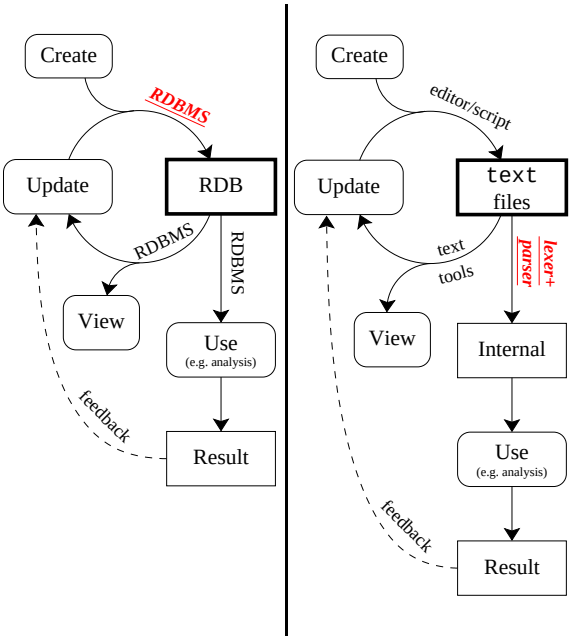

Figure 3: Flow of data in an RDB approach (left), and with `text` files as primary storage (right). The difficult (binary) structuring work is marked in (red) underlined-bold-italics.

6 `text` file solutions

Together with standard UNIX tools, a full toolchain for using text files as the primary information storage can be provided.

Information used by a computer needs to be arranged in (binary) structures in one way or another. The SQL approach mainly does this at the storage level, thus making the input stage into a program using the information simple. The text file solution defers the non-trivial structuring work until right before use. See Figure 3 for a comparison. By taking advantage of lexer/parser-based tools this otherwise very difficult stage becomes rather cookbook-like. Later use stages are then again straight-forward.

6.1 Handling

Handling text files is a lot like handling any other source code.

The bulk of data (especially for calibration parameters) is generated by programs. It can be as simple as `printf` invocations. (For good readability of the information, good indentation of the output has to be provided.) By using the file system well, one can keep quite some order and separation of different sets of parameters. I.e. by dividing and conquering the parameters in different files and use `#include` (or similar) directives bookkeeping is simplified.

The size of a number represented in decimal is about 2.4 times larger than a binary representation. It evens out a bit by decimal values only storing the significant digits, while binary formats are fixed-width. In any case, we are not even close to an order-of-magnitude difference in storage needs.

6.1.1 Viewing

In addition to using an editor, the files with information can be searched using e.g. GREP and LESS, differences highlighted by DIFF. One drawback is the line-orientation of these tools. They have a hard time handling context information that is spread in statements over multiple lines.

Leveraging the human brain! With its exceptional pattern-recognition capabilities, these built-in classification facilities can be used on the easy direct-view text files, at least when well formatted. Much less so on the RDB tables, if one even manages to at all wring them out of the RDBMS.

6.1.2 Modifications

Simple, but tedious modifications, e.g. due to some nomenclature change in the ana-

lysis programs can often be performed using some general text mangling tool: SED and PERL/PYTHON.

The most common form of modifications to large sets of parameters is post-processing / filtering of data from a calibration tool, performed by some dedicated script or program.

6.1.3 Even faster access

Even with similar storage sizes, parsing a text file is computationally more expensive than direct reading of binary data. For most use cases (e.g. slow-control, DAQ setup), this additional overhead is irrelevant. In analysis, it can become a bottle-neck[4]. But while the text files are the primary storage, they need not be used directly by an analysis process. By organising the lexing/parsing of the primary input into an intermediate storage in a manner like a usual compile-link-execute scheme, only changed files need to be re-read. As the intermediate storage is volatile, it can be carefully designed to match the needs of the analysis process, allowing it to be even treated memory-mapped and accessed as structures directly, without any expensive API calls. Yielding unrivalled access speeds.

6.1.4 History keeping

Version control systems (VCS), as used for source code control, can be applied to metadata like setup descriptions and calibration parameters. Everything spanning t_{Analysis} is handled by the VCS. Analysis tools, parameter parsing, etc. then one only has to care about one layer in the t_{Analysis} dimension.

GIT by its distributed nature fulfils the R5 requirement of being able to handle both a global history, and allow individual users to keep track of their separate developments and scratch-pad attempts.

[4] At least for short-cycle iterative development work.

With a little care, some VCS systems even can co-exist, e.g. CVS as an central (upstream) repository and GIT used side-by-side.

6.1.5 Different versions

During debugging, one is often faced with the question why something that worked before does not work any longer. When testing, one may then end up with inconsistencies between code and parameters. The old code does not read new sets of parameters or vice versa. For successful testing, modifications to both must then be possible.

How to handle that in a database? Local system? One should keep in mind that it is at these occasions that one has the most use of powerful history-juggling tools. Only having a crippled tool-chain available at these times makes debugging more painful than necessary.

7 Remarks

7.1 Policing work-flow

Policing is counter-productive. People end up with their own code in their own corner...

While there at "the end of analysis" really should be only one set of calibration parameters to describe an experiment, used to derive all extracted physics results, this unfortunately tends not to be the case in practice. People have a habit to use their own "optimal" parameters. And there lies the scare: have they thus biased their results? This certainly is an important issue for a serious collaboration.

It may be tempting to try to police this situation using technical means — essentially only allowing one central repository of parameters. This is likely to just lead to a plethora of local hacks to handle and override parameters individually. Which then become

very hard indeed to follow and ever reconcile. Better then to, from the technical point, provide a system which can deal with both central (global) and local parameter sets, using common tools to allow merging. And to do the harmonisation efforts where they belong: as collaborative and social discussions (the Spanish Inquisition?).

7.2 No XML

While superficially being text files, and with a known format, files on XML format might be suggested as a solution. Not so. They run afoul of several principles:

- Readability. An XML file is so full of "syntactic sugar" (keywords and other markup), that the actual payload becomes severely obscured. Extensive XML edits using any normal text editor is a wicked exercise without pleasure.

- Wrong level parsing / grammar description. While the well-established grammar of XML files lend themselves to efficient parsers, it at the same time makes it almost impossible to leverage the productive lexer/parser chain described above for the actual file content. In other words, the XML storage is as useless for the programmer as an SQL storage. The API provides the pieces of data, but not the relationships "automatically".

Why use a blunt axe, when a sharp one is available? [5]

7.3 Trees, not tables

An experimental setup is like a Matryoshka doll — facility, setup, detector, segment, channel. It is natural to describe the topologies of such a system as a tree. (Naturally, when several identical leaves or subbranches occur, one declares just one and references it multiple times.)

7.4 Reliability

With text files handled together with a distributed VCS, the parameters as well as the full utility toolbox are available for analysis work whenever the used (local) computer and its file-systems are operational.

For a centrally administered remote database, the uptime and load of the RDBMS has to be taken into account, as well as the necessary network connectivity and access.

7.5 Rants

Some proposed ways of using SQL databases tout "simple" wrappers around the allegedly otherwise difficult to use SQL interfaces. Which for the author begs the question: if SQL database engines are so good, why hide the back-end and only use a very small subset? It seems to become little more than just a big hash table (of key-value pairs)?

7.6 `text` files in use

Handling of calibration parameters and setup description by human-readable text files is in production use at the LAND/R^{3}B setup since more than 10 years. Essentially as described in Section 6, with the exception of the performance optimisations described in 6.1.3 not being implemented yet. Users so inclined are using VCS systems to keep track of their (and others') parameter sets.

Experience (from trying the opposite) is that calibration parameters are best kept separate for separate experiment beam-times. To ensure lasting synergy effects from having

[5] Because it hurts more?

similar setups, analysis code however needs to be kept common. Manual sharing of improvements and bug-fixes is very limited in practice.

References

[1] H. T. Johansson, *Hunting tools beyond the driplines*, PhD thesis, Göteborg, 2010, http://fy.chalmers.se/~f96hajo/phd/htj_thesis.pdf.

[2] *The ROOT system homepage*, http://root.cern.ch (2013-08-26).

[3] *CERN Program Library*, http://cernlib.web.cern.ch/cernlib/ (2013-08-26).