The Pierre Auger Observatory offline software

J. Allen[1], S. Argirò[2], S.L.C. Barroso[3], S.Y. BenZvi[4], G. Cataldi[5], M. Ding[1], J. Gonzalez[6], T. McCauley[8], L. Nellen[7], T. Paul[8], T.A. Porter[9], L. Prado, Jr.[10], M. Roth[11], R. Ulrich[11], M. Unger[11] and D. Veberič[12]

[1] New York University, 4 Washington Place, New York, NY 10003, USA.
[2] INFN and University of Torino, Via P. Giuria 1, I-10125 Torino, Italy
[3] Centro Brasileiro de Pesquisas Físicas, Rua Dr. Xavier Sigaud, 150, Rio de Janeiro-RJ, CEP 22290-180, Brazil
[4] Columbia University, 538 W. 120'th St., New York, NY 10027, USA
[5] INFN sez. di Lecce, Lecce, Italy
[6] Louisiana State University, Baton Rouge, LA, 70803, USA
[7] Departamento de Física de Altas Energías, Instituto de Ciencias Nucleares, Universidad Nacional Autonoma de México, México D.F., C.P 04510
[8] Northeastern University, 111-DA, 110 Forsyth St., Boston, MA 02115, USA.
[9] Santa Cruz Institute for Particle Physics, University of California, Santa Cruz, CA, 95046, USA
[10] Instituto de Física Gleb Wataghin, Universidade Estadual de Campinas UNICAMP, Campinas-SP, CP 6165 CEP 13083-970, Brazil
[11] Karlsruhe Institute of Technology KIT, University and Forschungszentrum Karlsruhe, POB 3640, D-76021 Karlsruhe, Germany
[12] University of Nova Gorica, Vipavska 13, PO Box 301, SI-5001, Nova Gorica, Slovenia

E-mail: t.paul@neu.edu

Abstract. The Pierre Auger Observatory aims to discover the nature and origins of the highest energy cosmic rays. The large number of physicists involved in the project and the diversity of simulation and reconstruction tasks pose a challenge for the offline analysis software, not unlike the challenges confronting software for very large high energy physics experiments. Previously we have reported on the design and implementation of a general purpose but relatively lightweight framework which allows collaborators to contribute algorithms and sequencing instructions to build up the variety of applications they require. In this report, we update the status of this work and describe some of the successes and difficulties encountered over the last few years of use. We explain the machinery used to manage user contributions, to organize the abundance of configuration files, to facilitate multi-format file handling, and to provide access to event and time-dependent detector information residing in various data sources. We also describe the testing procedures used to help maintain stability of the code in the face of a large number of contributions. Foundation classes will also be discussed, including a novel geometry package which allows manipulation of abstract geometrical objects independent of coordinate system choice.

1. Introduction

The offline software framework of the Pierre Auger Observatory [1] provides an infrastructure to support construction of the various applications necessary to analyze data gathered by the

observatory. The observatory is designed to measure the extensive air showers produced by the highest energy cosmic rays ($> 10^{19}$ eV) with the goal of discovering their origins and shedding light on their composition. Two different techniques are used to detect air showers. First, a collection of telescopes is used to sense the fluorescence light produced by excited atmospheric nitrogen as the cascade of particles develops and deposits energy in the atmosphere. This method can be used only when the sky is moonless and dark, and thus has roughly a 15% duty cycle. Second, an array of detectors on the ground is used to sample particle densities and arrival times as the air shower impinges upon the Earth's surface. Each surface detector consists of a tank containing 12 tons of purified water instrumented with photomultiplier tubes to detect the Cherenkov light produced by passing particles. The surface detector has nearly a 100% duty cycle. A subsample of air showers detected by both instruments, called hybrid events, are very precisely measured and provide an invaluable tool for cross checks and energy calibration. In order to provide full sky coverage, the baseline design of the observatory calls for two sites, one in the southern hemisphere and one in the north. The southern site is located in Mendoza, Argentina, and construction there is nearing completion, at which time the observatory will comprise 24 fluorescence telescopes overlooking 1600 surface detectors spaced 1.5 km apart on a hexagonal grid. The state of Colorado in the USA has been selected as the location for the northern site.

The requirements of this project place demands on the analysis software not unlike those faced by traditional high energy physics experiments. Though we are not not confronted by the same magnitude of data or instrumental complexity characteristic of the LHC experiments, the software must still support a large number of physicists developing many applications over a long experimental run. Specifically, the offline software supports simulation and reconstruction of events using surface, fluorescence and hybrid methods, as well as simulation of calibration techniques [2] and other ancillary tasks such as data preprocessing. Further, as the experimental run will be long, it is essential that the software be extensible to accommodate future upgrades to the observatory instrumentation (see for example [3, 4]), and the software needs to be adaptable to the requirements of the proposed northern site. The offline framework must also handle a number of formats in order to deal with data from a variety of instruments, as well as the output of air shower simulation codes. Additionally, it is essential that all physics code be "exposed" in the sense that any collaboration member must be able to replace existing algorithms with his or her own in a straightforward manner. This is meant to encourage independent analysis and ease comparison of results. Finally, while the underlying framework itself may exploit the full power of C++ and object-oriented design, the portions of the code directly used by physicists should not assume a particularly detailed knowledge of these topics.

A prototype of the offline framework was first reported at CHEP04. Since that time, the software has gone into production and has been used to generate physics results for about the past two and a half years. Below we outline the design and describe current state of the framework, and comment on some areas of ongoing development. A more detailed description of the offline software design, including some example applications, is available in [6]; this note provides an update of the work presented there.

2. Overview

The offline framework comprises three main parts: a collection of processing *modules* which can be assembled and sequenced through instructions contained in an XML file or in a Python script, an *event* data model through which modules relay data to one another and which accumulates all simulation and reconstruction information, and a *detector description* which constitutes a gateway to conditions data, including atmospheric properties as a function of time. The principal ingredients are depicted in Fig. 1.

These components rely on a set of foundation classes and utilities to support error logging,

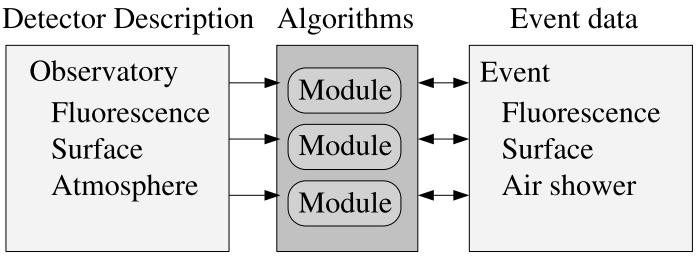

Figure 1. General structure of the offline framework. Simulation and reconstruction tasks encased in modules. Each module is able to read information from the detector description and/or the event, process the information, and write the results back into the event.

physics and mathematical manipulation, and abstract manipulation of geometrical objects, as described in more detail below.

3. User Modules and Run Control

Most tasks of interest of the Pierre Auger Collaboration can be factorized into sequences of self-contained processing steps which can simply be pipelined. Physicists prepare processing algorithms in so-called *modules*, which they register with the framework via a macro. This modular design allows collaborators to exchange code, compare algorithms and build up a variety of applications by combining modules in various sequences.

Modules inherit a common interface, and module authors must implement three abstract methods: a Run method, which is called once per event; and Init and Finish methods, to be called at the beginning and end of a processing job. To make the module available to the offline framework, authors invoke a macro in the module class declaration which registers a factory function. This function is then used by the framework to instantiate the module when requested.

For most applications, run-time control over module sequences is afforded through a *run controller*, which invokes the various processing steps within the modules according to a set of instructions provided in an XML file. We devised a simple XML-based [7] language for specifying sequencing instructions. Fig. 2 shows a simple example of the structure of a sequencing file. This approach has proved sufficiently flexible for the majority of applications, and it is quite simple and easy to learn. Further control over module sequencing is provided by a signaling mechanism in which modules can return flags to the run controller, instructing it to break a loop or to skip all subsequent modules up to the next loop tag.

4. Configuration

All configuration data are stored in a hierarchy of XML files. A globally accessible *central configurator* points modules and framework components to the location of their configuration data, and creates Xerces-based [21] XML parsers to assist in reading information from these locations. We have wrapped the Xerces API in with our own interface which provides ease-of-use at the cost of somewhat reduced flexibility compared to the standard DOM API, and which also adds functionality such as units conversion, including handling of units specified in expressions. The locations of configuration data are specified in a so-called *bootstrap* file, and may comprise local filenames, URIs [8] or XPath [9] addresses. The name of the bootstrap file is passed to the application at run time via the command line.

Provenance is afforded through the configuration mechanism, which can concatenate all configuration data accessed during a run and write it in an XML log file and into the event file. This log file includes a preamble with a format identical to that of a bootstrap file, with

```
<sequenceFile>
   <loop numTimes="unbounded">
     <module> SimulatedShowerReader </module>
     <loop numTimes="10" pushToStack="yes">
        <module> EventGenerator     </module>
        <module> TankSimulator      </module>
        <module> TriggerSimulator   </module>
        <module> EventExporter      </module>
     </loop>
   </loop>
</sequenceFile>
```

Figure 2. Simplified example in which an XML file sets a sequence of modules to conduct a simulation of the surface array. `<loop>` and `<module>` tags are interpreted by the run controller, which invokes the modules in the proper sequence. In this example, simulated showers are read from a file, and each shower is thrown onto the array in 10 random position by an `EventGenerator`. Subsequent modules simulate the response of the surface detectors and trigger, and export the simulated data to file. The `pushToStack="yes"` attribute instructs the Run Controller to store the event when entering the loop, and restore it to that state when returning back to the beginning of the loop. Note that XML naturally accommodates common sequencing requirements such as nested loops.

XPath addresses specifying the locations of all the configuration data in the file. Thus a log file can subsequently be read as though it were a bootstrap file in order to reproduce a run with an identical configuration.

The configuration logging mechanism may also be used to record the versions of modules and external libraries which are used during a run. Each module version is recorded in a static method of the module by extracting the Subversion [10] keyword `$Id:$` at build time. The version is then made available for logging via a method of the module interface.

Validation of XML files is afforded through W3C XML Schema [12] standard validation, which is well-supported in Xerces. Schema validation is used not only for internal framework configuration prepared by developers, but also to check configuration files of modules prepared by framework users. The standard schema types are complemented by a collection of types commonly used in our applications, allowing for quite detailed checking with minimal investment in schema preparation. Even casual users have demonstrated a willingness to invest the (small) time required to learn enough XML schema to check simple configuration files for modules.

The configuration machinery is also able to verify configuration file contents against a set of default files by employing MD5 digests [13]. The default configuration files are prepared by the framework developers and the analysis teams, and reference digests are computed from these files at build time. At run time, the digest for each configuration file is recomputed and compared to the reference value. Depending on run-time options, discrepant digests can either force program termination, or can simply log a warning. This machinery provides a means for those managing production campaigns to quickly verify that configurations in use are the ones which have been recommended for the task at hand.

5. Data Access

The offline framework includes two parallel hierarchies for accessing data: the *detector description* for retrieving conditions data, including detector geometry, calibration constants, and atmospheric conditions, and an *event* data model for reading and writing information that changes per event.

5.1. Detector Description

The *detector description* provides a unified interface from which module authors can retrieve conditions data. Data requests are passed by this interface to a back end comprising a registry of so-called *managers*, each of which is capable of extracting a particular sort of information from a a given data source. Lazy-evaluation is used to cache requests in the client interface classes. Generally we choose to store static detector information in XML files, and time-varying monitoring and calibration data in MySQL [20] databases. However, as the project evolves it sometimes happens that access to detector data in some other format is required, perhaps as a stop-gap measure. The manager mechanism allows one to quickly provide simple interfaces in such cases, keeping the complexity of accessing multiple formats hidden from the user. The structure of the detector description machinery is illustrated in Fig. 3.

Note that it is possible to implement more than one manager for a particular sort of data. In this way, one manager can override or augment data from another manager. For example, a user can decide to use a database for the majority of the description of the detector, but override some data by writing them in an XML file which is interpreted by the special manager. Alternatively, a user may add a manager to append hypothetical detector components to existing components in a Monte Carlo simulation. The specification of which data sources are accessed by the manager registry and in what order they are queried is detailed in a configuration file.

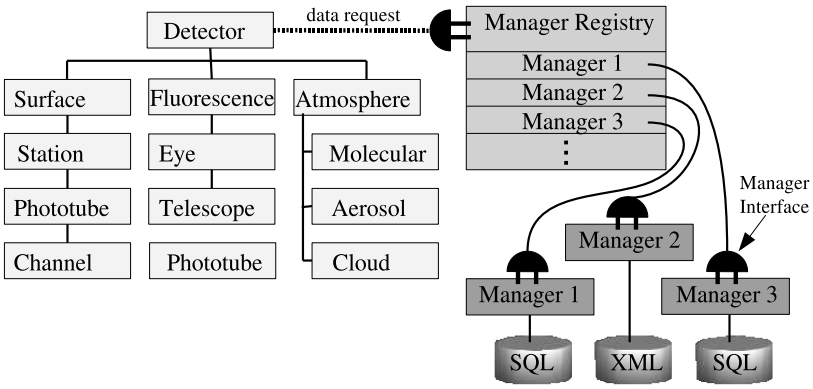

Figure 3. Machinery of the detector description. The user interface (left) comprises a hierarchy of objects describing the various components of the observatory. These objects relay requests for data to a registry of managers (right) which handle multiple data sources and formats.

The detector description also contains set of plug-in functions, called *models* which can be used for additional processing of data retrieved through the detector interface. These are used primarily to interpret atmospheric monitoring data, and like modules, are meant to be prepared by the user community, rather than (just) framework developers. As an example of use, analysis code can invoke a function to evaluate attenuation of light due to aerosols between two points in the atmosphere. This request is passed to a model, which interrogates the detector interface to find the atmospheric conditions at the specified time, and computes the attenuation. Models can also be placed under command of a *super-model* which can attempt various methods of computing the desired result, depending on what data are available for the specified time. The choice of which model(s) to use for a particular application is specified in a configuration file.

5.2. Event

The *Event* data model contains the raw, calibrated, reconstructed and Monte Carlo information and serves as the backbone for communication between modules. The overall structure comprises encapsulated classes organized following the hierarchy normally associated with the observatory

instruments, with further subdivisions for accessing such information as Monte Carlo truth, reconstructed quantities, calibration information and raw data. User modules access the event through a reference to the top of the hierarchy which is passed to the module interface by the run controller.

The event is instrumented with a protocol allowing modules to discover its constituents at any point in processing. This protocol provides the means for a given module to determine whether the input data required to carry out the desired processing is available.

The transient and persistent events are decoupled. When a request is made to write event contents to file, the data are transferred from the transient event through a file interface to the persistent event, which currently uses ROOT [19] for serialization. Conversely, when data are requested from file, a file interface transfers the data from the persistent event to the appropriate part of the transient event interface. The event may be transferred from memory to a file at any stage in the processing, and reloaded to continue processing from that point onward. Various file formats are handled using the file interface mechanism, including raw event and monitoring formats as well as the different formats employed by the AIRES [15], CORSIKA [16], CONEX [17] and SENECA [18] air shower simulation packages. Fig. 4 contains a diagram of this event input/output mechanism. The transient/persistent separation was adopted to avoid locking to a single provider solution for serialization, and to skirt some of the difficulties we encountered streaming our objects using ROOT. As mentioned in section 9, however, the approach does impose an undesirable maintenance burden on developers.

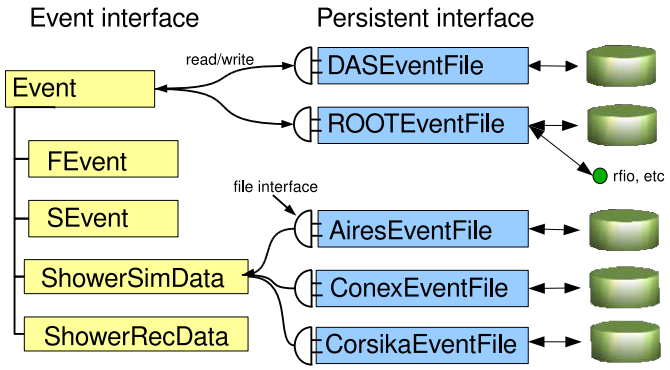

Figure 4. Event input/output. The section labeled "Event Interface" portrays a portion of the transient event. Data are transferred between this transient event and persistent objects through a common file interface. Different file implementations are able to read and/or write in different formats, including those used by the data acquisition systems (`DAS` formats), formats used by other simulation packages, as well as a "native" format (`ROOTEventFile`) which accommodates all raw data, reconstructed quantities, and Monte Carlo truth.

6. Utilities

The offline framework is complemented by a collection of utilities, including an XML parser, an error logger, various mathematics and physics services including a geometry package, testing utilities and a set of foundation classes to represent objects such as signal traces, tabulated functions and particles. In this section, we describe the geometry package in more detail.

6.1. Geometry

As discussed previously, the Pierre Auger Observatory comprises many instruments spread over a large area and, in the case of the fluorescence telescopes, oriented in different directions.

Consequently there is no naturally preferred coordinate system for the observatory; indeed each detector component has its own natural system, as do components of the event such as the air shower itself. Furthermore, since the detector spans more than 50 km from side to side, the curvature of the earth cannot generally be neglected. In such a circumstance, keeping track of all the required transformations when performing geometrical computations is tedious and error prone.

This problem is alleviated in the offline geometry package by providing abstract geometrical objects such as points and vectors. Operations on these objects can then be written in an abstract way, independent of any particular coordinate system. Internally, the objects store components and track the coordinate system used. There is no need for pre-defined coordinate system conventions, or coordinate system conversions at module boundaries. The transformation of the internal representation occurs automatically.

Despite the lack of a single natural coordinate system for the observatory, there are several important coordinate systems available. A registry mechanism provides access to a selection of global coordinate systems. Coordinate systems related to a particular component of the detector, like a telescope, or systems which depend on the event being processed, such as a shower coordinate system, are available through access functions belonging to the relevant classes of the detector or event structures.

Coordinate systems are defined relative to other coordinate systems. Ultimately, a single root coordinate system is required. It must be fixed by convention, and in our case we choose an origin at the center of the Earth. Other base coordinate systems and a caching mechanism help to avoid the construction of potentially long chains of transformations when going from one coordinate system to another.

The following is a simple example of how the geometry and units packages are used together:

```
Point pos(x*km, y*km, z*km, posCoordSys);
Vector dist(deltaX, deltaY, deltaZ, otherCoordSys);

Point newPos = pos + dist;
cout << "X = " << newPos.GetX(outCoordSys)/m << " meters";
```

The variables `x`, `y`, and `z` are provided by some external source, in the units indicated (km), whereas `deltaX`, `deltaY`, and `deltaZ` are results from a previous calculation, already in the internal units. Coordinate systems are required whenever components are used explicitly. Units are used on input and output of data and when exchanging information with external packages.

The surveying of the detector utilizes Universal Transverse Mercator (UTM) coordinates with the WGS84 ellipsoid. These coordinates are convenient for navigation. They involve, however, a non-linear, conformal transformation. The geometry package provides a `UTMPoint` class for dealing with positions given in UTM, in particular for the conversion to and from Cartesian coordinates. This class also handles the geodetic conventions, which define the latitude based on the local vertical as opposed to the angle $90^\circ - \theta$, where θ is the usual zenith angle in spherical coordinates.

The high degree of abstraction makes use of the geometry package quite easy. Uncontrolled, repeated coordinate transformations, though, can be a problem both for execution speed and for numerical precision. To control this behavior, it is possible to force the internal representation of an object to use a particular coordinate system. The geometry package guarantees that no transformations take place in operation where all objects are represented in the same coordinate system. This provides a handle for experts to control when transformations take place.

7. Build System and Quality Control

Unit and acceptance testing are integrated into the offline framework build and distribution system. Our build system is based on the GNU autotools [11], which provide hooks for

integrating tests with the build and distribution system. We have adopted the CppUnit [24] testing framework as an aid in implementing unit tests. In addition to such low-level tests, a set of higher-level acceptance tests has been developed which is used to verify that complete applications continue to function as expected during ongoing development. Such acceptance tests typically run full physics applications before and after each code change and notify developers in case of any unexpected differences in results.

As a distributed cross-platform project, the Auger Offline software must be regularly compiled and checked on numerous platforms. To automate this process, we have employed the tools provided by the BuildBot project [25]. The BuildBot is a Python-based system in which a master daemon is informed each time the code repository has been altered. The master then triggers a collection of build slaves running on various platforms to download the latest code, build it, run the unit and acceptance tests, and inform the appropriate developers in case problems are detected. This has proved to be a very effective system for us; the BuildBot is quite easy to set up and configure, and provides rapid feedback to developers allowing prompt resolution of problems.

8. External packages

The choice of external packages upon which to build the offline framework was dictated not only by package features and the requirement of being open-source, but also by our best assessment of prospects for longevity. At the same time, we attempted to avoid locking the design to any single-provider solution. To help achieve this, the functionality of external libraries is often provided to the client code through wrappers or façades, as in the case of XML parsing described in section 3, or through a bridge, as in the case of the detector description described in section 5.1. The collection of external libraries currently employed includes ROOT [19] for serialization, Xerces [21] for XML parsing and validation, CLHEP [27] for expression evaluation and geometry foundations, Boost [26] for its many valuable C++ extensions, and Geant4 [28] for detailed surface and fluorescence detector simulations.

9. The Future

While the framework described in this note is actively used for analysis, there are a several substantial improvements and enhancements in preparation.

Though several event visualization and browsing packages have been prepared, we are developing a new interactive visualization package which is fully integrated into the framework and which will provide not only graphical display of reconstructed event properties and Monte Carlo truth, but also interactive control over configuration and reconstruction procedures.

Python [29] bindings for the framework are in preparation. Once complete, all of the framework public interfaces will be exposed via Python, allowing users to prepare rapid prototypes of analysis and visualization tasks. Python-based module sequencing will also be supported, allowing more intricate run control than is currently afforded through our XML-based sequencing system for cases when this may be desired.

The user module system described in section 3 is being upgraded to support dynamical loading of modules. This will allow for easier use of modules with the interactive visualization system mentioned above, and support easier module distribution and shorter development cycles.

We are in the process of grid-enabling the offline software. The results of a first Auger offline software data processing grid challenge were recently reported [30].

Finally, we are investigating ways to ameliorate the maintenance burden imposed by splitting the event into transient and persistent versions, including the possility of generating the transient and persistent event classes automatically from an XML meta-description.

10. Conclusions

We have implemented a software framework for analysis of data gathered by the Pierre Auger Observatory. This software provides machinery to facilitate collaborative development of algorithms to address various analysis tasks as well as tools to assist in the configuration and bookkeeping needed for production runs of simulated and real data. The framework is sufficiently configurable to adapt to a diverse set of applications, while the user side remains simple enough for C++ non-experts to learn in a reasonable time. The modular design allows straightforward swapping of algorithms for quick comparisons of different approaches to a problem. The interfaces to detector and event information free the users from having to deal individually with multiple data formats and data sources. This software is now used for production of physics results from the observatory.

11. Acknowledgments

The authors would like to acknowledge the support of their various funding agencies.

12. References

[1] Abraham J *et al.* [Pierre Auger Collaboration] 2004 *Nucl. Instrum. Meth. A* **523**, 50
[2] Ghia P [Pierre Auger Collaboration] 2007, *Proc. 30th Intl. Cosmic Ray Conference (ICRC 2007), Merida, Mexico, 3-10 July 2007 Preprint* arXiv:0706.1212 [astro-ph]
[3] H. Klages [Pierre Auger Collaboration], "HEAT : Enhancement Telescopes for the Pierre Auger Southern Observatory in Argentina", *Proc. 30th Intl. Cosmic Ray Conference (ICRC 2007), Merida, Mexico, 3-10 July 2007*
[4] Medina M, Berisso M, Allekotte I, Etchegoyen A, Tanco G and Supanitsky A 2006, *Nucl. Instrum. Meth.* A **566**, 302;
Etchegoyen A [Pierre Auger Collaboration] 2007, "AMIGA: A muon detector and infilled array for the Auger Observatory" *Proc. 30th Intl. Cosmic Ray Conference (ICRC 2007), Merida, Mexico, 3-10 July 2007*
[5] Argirò S *et al.* [Pierre Auger Collaboration] 2005, "The offline software framework of the Pierre Auger Observatory" *Proc. 29th Intl. Cosmic Ray Conference (ICRC 2005), Pune, India, 3-11 Aug 2005*
[6] Argiro S *et al.* 2007 *Nucl. Instr. and Meth. A*, doi:10.1016/j.nima.2007.07.010 *Preprint* arXiv:0707.1652 [astro-ph]
[7] `http://www.w3.org/XML/`
[8] `http://tools.ietf.org/html/rfc3986/`
[9] `http://www.w3.org/TR/xpath`
[10] `http:subversion.tigris.org`
[11] `http://www.gnu.org/software/autoconf`;
`http://www.gnu.org/software/automake`;
`http://www.gnu.org/software/libtool`
[12] `http://www.w3.org/XML/Schema/`
[13] Rivest R, `http://www.faqs.org/rfcs/`
[14] See for example Patton S 2003, "Concrete uses of XML in software development and data analysis" *Proc. Intl. Conf. on Computing in High-Energy Physics and Nuclear Physics (CHEP 2003), La Jolla, California, USA 24-28 March 2003*
[15] Sciutto S, *Preprint* arXiv:astro-ph/9911331
[16] Heck D, Knapp J, Capdevielle J, Schatz G and Thuow T, 1998 Report FZKA 6019
[17] Bergmann T *et al.* 2007 *Astropart. Phys.* **26**, 420
[18] Drescher H, Farrar G, Bleicher M, Reiter M, Soff S and Stoecker H 2003 *Phys.Rev.* **D67**, 116001
[19] `http://root.cern.ch/`
[20] `http://dev.mysql.com`
[21] `http://xml.apache.org/`
[22] see for example Josuttis N (1999) *The C++ Standard Library*, Addison-Wesley, ISBN 0-201-37926-0
[23] `http://www.w3.org/TR/REC-xml/`
[24] `http://cppunit.sourceforge.net/doc/1.8.0/`
[25] `http://buildbot.sourceforge.net/`
[26] `http://www.boost.org/`
[27] `http://proj-clhep.web.cern.ch/proj-clhep/`

[28] http://geant4.cern.ch/; Agostinelli S *et al.* 2003 *Nucl. Instrum. Meth.* **A 506**, 250
[29] http://www.python.org/
[30] J. Chudoba, "Simulations and Offline Data Processing for the Auger Experiment", Presented at EGEE User Forum, Manchester, UK, 9-11 May 2007.
http://indico.cern.ch/conferenceDisplay.py?confId=7247