PROCEEDINGS OF SCIENCE

Technical Foundations of CORSIKA 8: New Concepts for Scientific Computing

Hans Dembinski*
Max Planck Institute for Nuclear Physics, Heidelberg, Germany
E-mail: hdembins@mpi-hd.mpg.de

Lukas Nellen
National Autonomous University of Mexico, Mexico City, Mexico
E-mail: lukas@nucleares.unam.mx

Maximilian Reininghaus
Karlsruhe Institute of Technology, Karlsruhe, Germany
Instituto de Tecnologías en Detección y Astropartículas (CNEA, CONICET, UNSAM), Buenos Aires, Argentina
E-mail: reininghaus@kit.edu

Ralf Ulrich
Karlsruhe Institute of Technology, Karlsruhe, Germany
E-mail: ralf.ulrich@kit.edu

for the CORSIKA 8 collaboration†

CORSIKA is the leading simulation code for air showers in the field of astroparticle physics. CORSIKA 8 is a new project aiming to make CORSIKA ready for the next decades of research; a rewrite of CORSIKA in modern C++ with a flexible, efficient, and modular design. CORSIKA 8 makes full use of open development, being a collaborative project with contributors from around the world. The modular design makes modifications and contributions very straightforward and lowers the technical barrier for users to become active developers. CORSIKA 8 is written in C++17, which brings new powerful features useful for scientific high-performance computing. We discuss work on its technical foundations, the geometry and quantity system (a quantity is a number with a dimension). The goal of these systems is to make physical and geometric calculations easy and safe in CORSIKA 8, while maintaining highest computational speed. We further discuss how continuous integration is used to maintain high code quality standards.

36th International Cosmic Ray Conference -ICRC2019-
July 24th - August 1st, 2019
Madison, WI, U.S.A.

*Speaker.
†https://gitlab.ikp.kit.edu/AirShowerPhysics/corsika/wikis/ICRC2019-author-list

PoS(ICRC2019)236

1. Introduction

Cosmic and gamma rays from GeV up ZeV energies are a key research subject in astoparticle physics. The interactions of such particles with magnetic fields and matter are of central importance for many ongoing or planned projects [1–4]. In many cases cosmic rays are directly the main messenger particles, and in other cases they are the main physics backgrounds. Excellent understanding of cosmic ray interactions with matter and magnetic fields is one of the baseline requirements for all experiments in the field of cosmic rays as well as in gamma ray and neutrino astronomy. CORSIKA 8 will provide a modern and powerful framework for this.

2. Goal and strategy of the project

The goal driving the development the CORSIKA 8 is to provide a modular, flexible, and efficient framework to support physics applications in the most optimal way. The framework is specifically designed to deal with the computational problems of redistributing enormous amounts of energy from single particles into huge secondary particle cascades. One particular physics driven issue is the precision of consistently handling particles on vastly different energy and length scales. Algorithms must be sufficiently precise and very robust to support this. There are also specific difficulties that arise in the context of huge particle cascades; the storage and memory management of handling single or billions of particles simultaneously in the same framework must be very efficient and completely transparent to the user.

It is a typical and very common problem of physics software that the implemented physics models, theories and algorithms can become extremely complex and it is up to the physicist programmer to take care of the correctness and consistency. Examples where special care is required are: coordinate systems, geometric transformation, physical units, and particle codes. In CORSIKA 8 we provide technology to help preventing the most problematic of such difficulties and mistakes. The driving concept behind this is that in physics most calculations are done in specific reference frames or with important assumptions like unit systems. For the first time in such a simulation framework we enforce all of such external assumption on the level of software syntax. The software will not compile or work if the physicist by accident violates some of the underlying physical assumptions and reference systems. Very importantly this means: the software will never work incorrectly because of such mistakes.

Furthermore, the CORSIKA 8 framework is developed in a modular way that makes it straightforward to separate code to handle specific tasks. This is of critical importance since it will allow to extend the framework in the future and to replace or complement existing parts with new knowledge. A full structural outline of the CORSIKA 8 shower simulation framework is given in fig. 1.

3. Unit system

The incorrect usage of physical units of measure is regarded as one of the three most common errors in scientific computing [6]. The perhaps most illustrative example is the loss of the Mars Climate Orbiter during its orbital insertion maneuver the cause of which is largely attributed to the mixture of imperial and metric units [7].

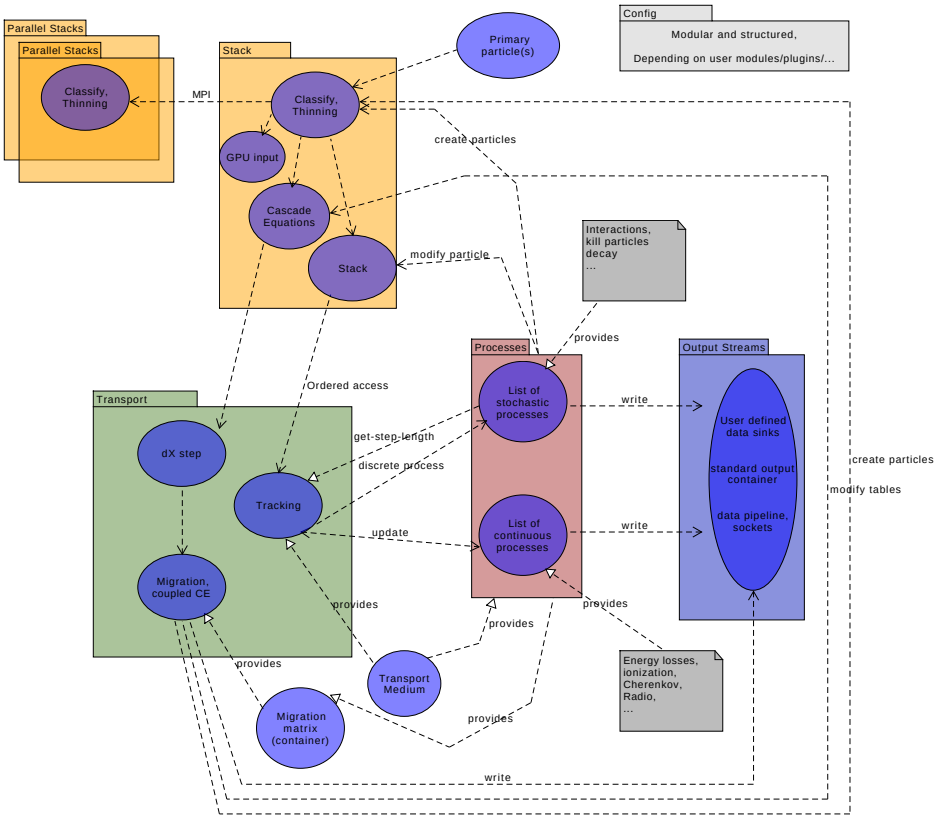

Figure 1: The basic structure of the data and control flow in CORSIKA 8 [5]. Modularity is a key aspect of the design.

Especially in a collaborative environment such as CORSIKA 8, a strategy to avoid these kinds of errors is a key asset. A possible solution, annotating data types with units, was proposed already more than 40 years ago [8] but hardly any programming language today provides built-in support. This usually requires the use of custom libraries [9]. Unit support for C++, making use of the technique of template metaprogramming, was first described in ref. [10] and a conceptually very similar solution is employed in CORSIKA 8, based on the *PhysUnits C++11* library [11] with some custom modifications described below. For the typical developer, working with code with units has the following implications:

- Literals converting quantities to the CORSIKA 8-internal units are provided. One can write `height = 1.425_km` without having to worry about whether or not CORSIKA 8 handles lengths internally in centimeters, or whether cross-sections are handled in millibarn, as it is the case e.g. in SIBYLL [12], or rather in femtometer[2] as in PYTHIA [13].

- Quantities of distinct physical dimensions correspond to distinct datatypes, e.g. `GrammageType` and `CrossSectionType`, making the meaning of arguments and return values in function signatures much easier to grasp at first glance and therefore improving the readability of the code.

- A dimensional analysis is performed during compilation. Multiplications and divisions involving quantities typically yield quantities of different dimensions, e.g. dividing `LengthType` by `TimeType` yields `SpeedType`. Additions and subtractions are only possible with quantities of the same type. Calculations which violate these rules result in compilation errors, thereby preventing errors that would otherwise likely stay unnoticed.

The implementation is based on template metaprogramming and keeps track of the dimensions of quantities via a set of integers (since fractional powers of dimensions do not occur in physical equations [14]) representing the powers of each base dimension, i.e., mass, length, time, etc., of the International System of Units (SI). While in principle sufficient to express any calculation, the strict enforcement of correct dimensions by the compiler prevents volitional "sloppy" handling of units as we often want to do it in particle physics with natural units, e.g. writing $E^2 = p^2 + m^2$ would be impossible. On the other hand, imposing $c = \hbar = 1$ would reduce the number of dimensions and, among other things, time and length dimensions would become equivalent, spoiling the usefulness of the whole machinery. As a compromise solution, noticing that there are only very few places in which both macroscopic and microscopic units are relevant at the same time, we therefore complemented the base dimensions by a "HEP energy dimension" with the base unit eV. Conversion functions are provided to convert SI to natural units and vice versa by multiplying by appropriate powers of $\hbar c \approx 197\,\mathrm{MeV\,fm}$ determined mostly automatically.

We would like to stress here that the whole unit algebra is performed during compilation and impacts neither the memory footprint nor the execution speed of CORSIKA 8.

4. Geometry and environment system

The geometry system is one of the core parts of the framework and handles all low-level geometric computations, i.e., computations involving points, vectors, lines, spheres, etc. The main aspects of its design are inspired by the Offline software framework of the Pierre Auger Observatory [15].

There is a clear distinction between points, vectors and their respective coordinates or components in given coordinate systems (CS). For example, when defining new points from their coordinates (rather than obtaining them as a result of computations with existing objects) one needs to specify the CS in which the coordinates are valid. There is one predefined CS, the *root CS*, and new CSes can be defined by specifying a transformation which transforms an already exiting CS to the new CS. As possible transformations we support elements of $SE(3)$, i.e. rotations and translations. The developer can then query the coordinates of the point in any CS and the appropriate chain of required transformations applied to the coordinates is determined automatically in the background. While coordinates of points are affected by both rotations and translations, vector components are affected only by the rotational part of a transformation since they can be considered as displacement between two points, therefore invariant under translations.

It should be noted, however, that the explicit usage of coordinates or vectors, and therefore also the construction of new CS, is only rarely necessary and computations can often be expressed purely symbolically. For example, the mass density distribution $\rho(\boldsymbol{r})$ at some point $\boldsymbol{r}$ in the atmo-

sphere according to the isothermal model in the approximation of a planar Earth is given by

$$\rho(\boldsymbol{r}) = \rho_0 \exp\left(\frac{1}{\lambda}(\boldsymbol{r} - \boldsymbol{p}_0) \cdot \vec{a}\right), \tag{4.1}$$

where $\vec{a}$ denotes the normal vector of the plane, ρ_0 the density at the reference point $\boldsymbol{p}_0$, and λ the scale height. In code, this could be expressed as `rho0 * exp((r - p0).dot(a) / lambda)` without the need to refer to any CS. In case $\boldsymbol{r}, \boldsymbol{p}_0$, or $\vec{a}$ are defined in different CSs, the transformation to a common CS happens automatically.

The actual linear algebra happening in the background is handled by the Eigen (v3) library [16]. On top of that, a wrapper class called `QuantityVector` integrates the unit system with the bare Eigen data types so that vectors of different quantities, e.g. lengths and magnetic fields, can be used together and their scalar and cross products yield the appropriate quantities. Points on the other hand live in Euclidean space by definition and therefore only lengths are supported as their coordinates. The `Vector` and `Point` classes use a `QuantityVector` internally to store their components / coordinates and furthermore keep a reference to the CS they are defined in (see fig. 2).

The geometry system is extensively used within the environment module. With that, users can specify their own models of the "world" in which they want to propagate particle cascades. The basic idea consists of specifying volumes of simple geometric shapes which are then assigned to models of their relevant physical properties [17]. The single volumes are combined in a tree structure, the *volume tree*, which represents containment: An inner volume that is fully contained within an outer volume is a child node of the latter. This principle was chosen to facilitate computationally efficient particle tracking and is inspired by common practice in ray tracing in computer graphics: In every step of the particle propagation, it is necessary to determine the next point of intersection of its trajectory "ray", based on current position and direction of flight, with a volume boundary. With the described hierarchical volume tree the number of such intersection tests are minimized.

A second aspect concerning the particle tracking in an environment with multiple nodes is the numerical precision at boundary crossings between two nodes [18]. Particles which are to be moved to the point of intersection will either end up numerically slightly after or before the correct boundary point, causing a mismatch between the logical (where it should be) and numerical (actual) location in the volume tree. Especially the first case can be problematic since the particle would encounter the same boundary crossing again during the next step and essentially become stuck. By explicitly storing the current logical node of the particle as one of its properties and updating it at every boundary crossing we are able to prevent such spurious boundary crossings in an efficient and robust way.

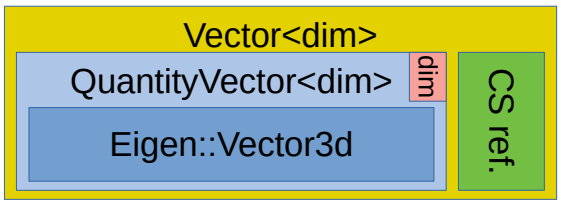

Figure 2: A sketch of the `Vector` class; `dim` denotes the dimension of the quantity stored inside and is a template parameter.

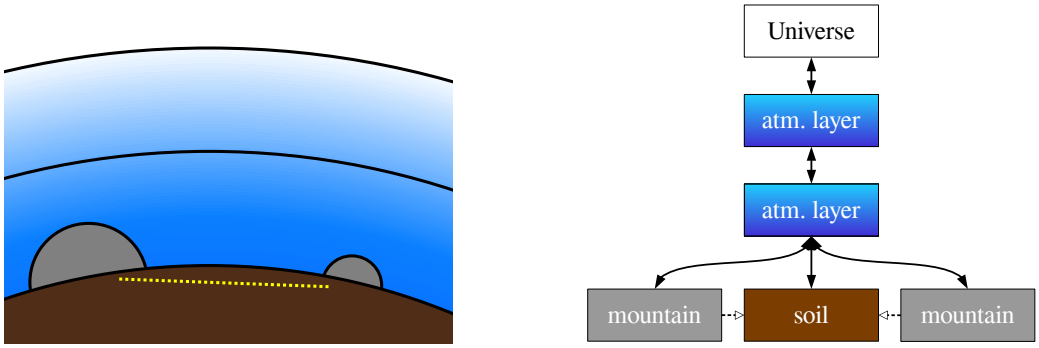

Figure 3: *Left:* example environment with a two-layer atmosphere, two mountains, and an underground observation level (indicated by the yellow, dashed line). *Right:* its representation as a volume tree.

5. Continuous integration

CORSIKA 8 uses *continuous integration* (CI) to permanently keep the software in a valid state. The CI system runs a suite of automated tests of the software for each feature that is about the be merged into the master branch. The compilation test is the most basic, it checks that the software compiles correctly on all supported platforms. We use *Docker*[1] instances to set-up identical blank copies of the test platforms. If the software compiles, a suite of *unit tests* is run inside the Docker instance. A unit test executes a single isolated aspect (a unit) of the public interface of CORSIKA 8 – typically a single function, class, or method – on a fixed input and compares the output with an expectation. A unit test requires that the tested aspect of the software behaves deterministically and triggers no side-effects that change the results of other unit tests. If any tests fails, the CI system reports the failure and the developer is required to patch until all tests pass again.

Ideally, unit tests check the full specification of the software unit by testing all possible inputs, including inputs that trigger an error state. Literally testing all input values is often unfeasible. In practice, a few representative input samples for each unique code path inside the software unit are checked. For example, for the function `double sqrt(double)`, one could test the input values `-1, 0, 1, 4, 9,` $\pm\infty$, and `NaN`. The C++ compilers `gcc`[2] and `clang`[3] offer options to instrument executables with counters that check how often a source code line was executed. This is used to measure *code coverage*, the fraction of source lines tested in unit tests. Coverage data is generated and checked, current coverage is $> 90\%$. Changes and features which add new code paths to CORSIKA 8 are required to have accompanying unit tests so that they do not reduce the code coverage.

The C++ compilers also offer to instrument the generated executables with *sanitizers*[4]. Sanitizers detect common mistakes in C++ programs, in particular when an object is accessed through a pointer or reference whose life-time already ended. Sanitizers offer a fast alternative to running

[1] www.docker.com
[2] gcc.gnu.org
[3] clang.llvm.org
[4] clang.llvm.org/docs/AddressSanitizer.html

tests in `valgrind`[5], which slows down execution by orders of magnitude compared to a factor of 2 to 3. Sanitizers will be used for unit tests where the performance penalty is acceptable.

CORSIKA 8 will use a semi-automatic *validation system* in addition to unit tests, which is only run in regular intervals and before releases and checked by humans. Ideally, CI keeps a software project in a valid state so that a release can be made at any time, but unit tests cannot test all aspects of a Monte Carlo simulation. When (pseudo) random numbers are involved, a tiny change to the code and the input can cause an unpredictable changes in the output. The validation will compare averages over many simulation runs, which are expensive to compute and may still randomly fluctuate outside of defined bounds. Therefore, a human interpretation is needed.

6. Static polymorphism for physics processes

CORSIKA 8 handles physics processes, such as continuous energy loss, interactions, decay, and so on, as a set of user-extensible classes. Process instances are placed into a process sequence, which is iterated over for each particle on the particle stack.

Dynamic polymorphism is the classic approach to implement such a process sequence. A base class with virtual methods defines the uniform interface. Implementation classes inherit from the base and override these methods. Instances of implementation classes must be allocated from the heap. Algorithms can uniformly act on them by accessing the instance through the base class pointer. Behind the scene, a call to a virtual method through the base pointer passes through a hidden pointer table and is forwarded to the correct method function pointer of the derived class.

The approach is not performant when light-weight virtual methods are accessed in a tight loop. Firstly, instances of implementation classes allocated on the heap are in general not placed local in memory. Local memory access is important, since modern CPUs are limited by memory latency and need to use CPU caches effectively. Secondly, since the method function pointer resolution happens at run-time, the C++ compiler cannot inline the method body. Without inlining, one has to pay for two extra pointer jumps and the compiler cannot optimize the surrounding code.

Static polymorphism is the modern alternative, when the process sequence is known at compile-time. In this case, one writes implementation classes with a common interface. A common base class is not required. Instances of different types can be placed in a `std::tuple` and iterated over. Memory access is local, the C++ compiler can inline methods and apply optimizations across several method calls. CORSIKA 8 primarily uses static polymorphism, but dynamic polymorphism is also considered. It would allow users to modify the process sequence without recompiling the code, for example, to change the process sequence from Python.

7. Conclusion

The CORSIKA 8 simulation framework provides a novel solution for modular, robust, and efficient computing in astroparticle physics. Modern tools and best practices for distributed development are employed. The framework actively prevents typical mistakes and helps the application programmer to yield correct results fast and reliably. The internal structure of CORSIKA 8 is optimized for modularity and extensibility as well as for efficient memory and resource usage.

[5] valgrind.org

Acknowledgements

M.R. acknowledges support by the DFG-funded Doctoral School "Karlsruhe School of Elementary and Astroparticle Physics: Science and Technology".

References

[1] CTA CONSORTIUM collaboration, *Design concepts for the Cherenkov Telescope Array CTA: An advanced facility for ground-based high-energy gamma-ray astronomy*, *Exper. Astron.* **32** (2011) 193 [1008.3703].

[2] PIERRE AUGER collaboration, *The Pierre Auger Cosmic Ray Observatory*, *Nucl. Instrum. Meth. A* **798** (2015) 172 [1502.01323].

[3] TELESCOPE ARRAY collaboration, *The surface detector array of the Telescope Array experiment*, *Nucl. Instrum. Meth. A* **689** (2013) 87 [1201.4964].

[4] ICECUBE collaboration, *First Year Performance of The IceCube Neutrino Telescope*, *Astropart. Phys.* **26** (2006) 155 [astro-ph/0604450].

[5] R. Engel, D. Heck, T. Huege, T. Pierog, M. Reininghaus, F. Riehn et al., *Towards a Next Generation of CORSIKA: A Framework for the Simulation of Particle Cascades in Astroparticle Physics*, *Comput. Softw. Big Sci.* **3** (2019) 2 [1808.08226].

[6] ISO/IEC JTC1/SC22/WG5, *Information Technology — Programming languages — Fortran — Units of measure for numerical quantities*, Tech. Rep. N2113, 2016.

[7] Mars Climate Orbiter Mishap Investigation Board, *Phase I report*, 1999.

[8] N. Gehani, *Units of measure as a data attribute*, *Comput. Lang.* **2** (1977) 93 .

[9] O. Bennich-Björkman and S. McKeever, *The Next 700 Unit of Measurement Checkers*, in *Proc. 11th ACM SIGPLAN Int. Conf. Softw. Lang. Eng.*, SLE 2018, pp. 121–132, 2018, DOI.

[10] Z. D. Umrigar, *Fully Static Dimensional Analysis with C++*, *SIGPLAN Not.* **29** (1994) 135.

[11] M. Moene, "PhysUnits C++11." https://github.com/martinmoene/PhysUnits-CT-Cpp11.

[12] E.-J. Ahn, R. Engel, T. K. Gaisser, P. Lipari and T. Stanev, *Cosmic ray interaction event generator SIBYLL 2.1*, *Phys. Rev. D* **80** (2009) 094003 [0906.4113].

[13] T. Sjöstrand, S. Ask, J. R. Christiansen, R. Corke, N. Desai, P. Ilten et al., *An Introduction to PYTHIA 8.2*, *Comput. Phys. Commun.* **191** (2015) 159 [1410.3012].

[14] Á. P. Raposo, *The algebraic structure of quantity calculus II: Dimensional analysis and differential and integral calculus*, *Measur. Sci. Rev.* **19** (2019) 70 .

[15] S. Argirò, S. L. C. Barroso, J. Gonzalez, L. Nellen, T. C. Paul, T. A. Porter et al., *The offline software framework of the Pierre Auger Observatory*, *Nucl. Instrum. Meth. A* **580** (2007) 1485 [0707.1652].

[16] G. Guennebaud, B. Jacob et al., "Eigen v3." https://eigen.tuxfamily.org.

[17] M. Reininghaus and R. Ulrich, *CORSIKA 8 – Towards a modern framework for the simulation of extensive air showers*, *EPJ Web Conf.* **210** (2019) 02011 [1902.02822].

[18] B. M. Smith, *Robust Tracking and Advanced Geometry for Monte Carlo Radiation Transport*, Ph.D. thesis, Fusion Technology Institute, University of Wisconsin-Madison, 2011.