PoS PROCEEDINGS OF SCIENCE

# Chromo: An event generator frontend for particle and astroparticle physics

**Hans Dembinski,[a,*] Anatoli Fedynitch[b,c] and Anton Prosekin[b]**

[a] *Department of Physics, TU Dortmund, Dortmund, Germany*
[b] *Institute of Physics, Academia Sinica, Taipei, Taiwan*
[c] *Institute for Cosmic Ray Research, The University of Tokyo, Tokyo, Japan*

*E-mail:* hans.dembinski@tu-dortmund.de, anatoli@gate.sinica.edu.tw

Chromo (formerly known as IMPY) is a Python frontend that provides a unified interface to popular generators of hadronic interactions, such as EPOS, DPMJet, QGSJet, Sibyll, and Pythia, which are used to simulate air showers or minimum bias events at colliders. Chromo is a thin wrapper on top of these codes, which are written in Fortran or C++, and does not impose a notable performance penalty. As a Python library, Chromo runs in Jupyter notebooks or Python scripts and also comes with a command-line mode similar to the program CRMC. Events can be written to HepMC and ROOT files or exposed as Numpy arrays. These can be inspected and transformed with Python code, directly accumulated as histograms, and saved to disk even after modifications. Chromo's source code is distributed via GitHub and is automatically tested and built by a continuous integration service. The installation process is extremely simple, since the software package is distributed as a binary wheel via PyPI for Linux, macOS, and Windows. Therefore, it can be readily used in education, for new projects, or as a drop-in replacement for CRMC. Chromo is the central tool for the computing of secondary particle distributions in the MCEq cascade solver, and has been extensively used for the development and testing of the DPMJet and Sibyll event generators.

38th International Cosmic Ray Conference (ICRC2023)
26 July - 3 August, 2023
Nagoya, Japan

*Speaker

© Copyright owned by the author(s) under the terms of the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License (CC BY-NC-ND 4.0).

https://pos.sissa.it/

PoS(ICRC2023)189

## 1. Introduction

Simulations of hadronic, photo-hadronic, and nuclear interactions are indispensable across all domains of high-energy particle and astroparticle physics. At colliders, they are used to model the underlying event in analyses which work with nuclear beams or targets, like the LHC or RHIC. These interactions also transpire in cosmic-ray acceleration sites, during interstellar and intergalactic transit, and within Earth's atmosphere. They give rise to extensive air showers, detectable via surface or fluorescence detectors. Interpreting these observations requires air-shower simulations, enabling the transformation of detector signals into meaningful physical quantities, such as the energy and mass of the primary particle.

In the air shower application, the unknown initial state and interaction energy often exceeding the range probed by accelerators. The forward phase space (small-angle scattering), mainly probed at fixed target colliders with energies below 100 GeV, requires extrapolation up to PeV energies based on phenomenological models, which cannot be directly validated against collider data. Efforts are currently being made to improve these models at the highest energies using air shower, atmospheric muon, and accelerator data [1].

General-purpose event generators, like `Sibyll` [2, 3], `DPMJet` [4, 5], and `Pythia` [6, 7], synergize phenomenological models with quantum-chromodynamical (QCD) calculations. These tools are designed to simulate authentic event topologies and kinematics while preserving energy, momentum, and quantum numbers. Developed over decades using diverse fixed target and collider data, their use is restricted due to the narrow particle production phase space exposure of a single dataset or accelerator experiment. While `Pythia 8` is preferred for high-energy collider simulations, `DPMJet` is closely integrated with the `Fluka` cascade code for nuclear interactions. `EPOS-LHC` [8] finds extensive usage in collider, air shower, and heavy-ion physics.

Although the codes are validated against available data and tunable to match data within the experimentally viable energy and phase space range, no code can claim to reproduce all data of interest satisfactorily. In HEP analyses and in extensive air shower simulations one often employs various event generators to estimate the theory uncertainty of results. A significant hurdle is the absence of a unified user interface. Each generator uses varying procedures to define kinematics, generate events, and to configure the generator. Outputs also differ with varying data structures, particle identification codes (PIDs), and levels of event information.

The Cosmic Ray Monte Carlo (`CRMC`) package [9] has effectively streamlined user interfaces and outputs, utilizing a blend of `EPOS-LHC` subroutines, C++ code, and the `HepMC` library. However, `CRMC` requires a full developer setup, a Unix operating system with compatible compilers, and bespoke tools for post-processing results. It lacks unit tests to ensure functionality across all target architectures and operating systems.

In the following sections, we introduce Chromo, our new tool embedded in the Scientific Python Ecosystem, designed to address most challenges associated with utilizing event generators.

## 2. Features

Chromo is a unified front-end to general-purpose event generators used in particle and astroparticle physics. It supports `DPMJet-III 3.0.6`, `19.1`, and `19.3`; `PHOJet 1.12-35`, `19.1`, and



PoS(ICRC2023)189

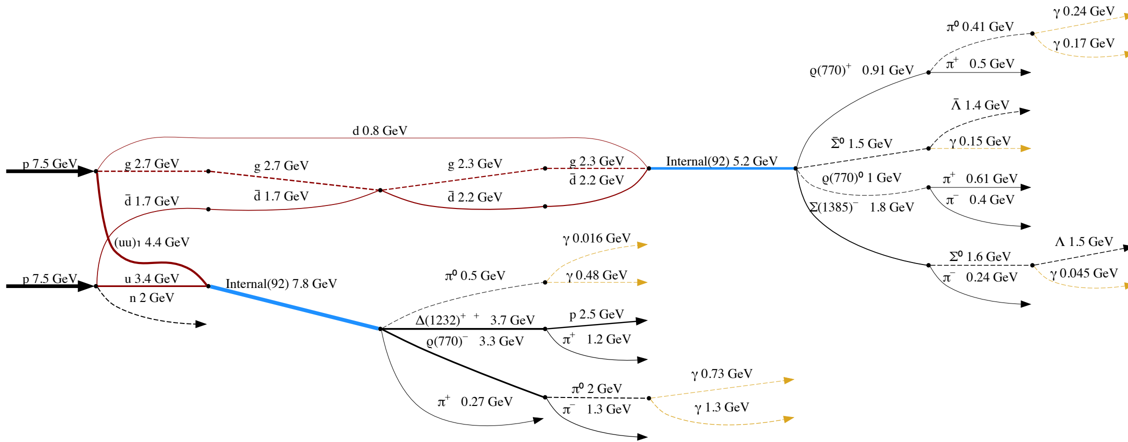


**Figure 1:** Event visualization via HepMC

`19.3`; `EPOS-LHC`; `Pythia 6.4` and `8.3`; `QGSJet-01`; `QGSJet-II-03` and `04` [10]; `Sibyll-2.1` and `2.3d`; `SOPHIA 2.0`; and `UrQMD 3.4`. It allows nonexperts to run various event generators and to compare their output to measurements or to estimate the current model uncertainty in regard to certain processes. Like its precursor `CRMC`, it can be used from the command-line to generate events in standard formats used in HEP: HepMC3 and ROOT. In addition, it can be used as a library and called interactively from Jupyter notebooks and other Python code. This allows one to process the generated event directly without the time-consuming conversion to another format.

A key feature is effort-less installation. The command `pip install chromo` installs a precompiled package on Linux, macOS, and Windows, including all required dependencies, which is immediately ready to use. Chromo makes use of the standard Python packaging architecture and infrastructure to achieve this. Previously, users had to set up a compilation chain for C++ and Fortran and follow specific build instructions for each event generator. All that complexity is removed to make state-of-the-art event generators accessible to everyone.

Chromo was carefully designed to incur negligible runtime overhead, so that event generation with Chromo is effectively as fast as running the bare generator. More about Chromo's outstanding performance can be found in Section 4.

Chromo further provides a modern object-oriented interface for event generators with a shallow learning curve. A generator is started with its default settings, further setup is only required if these ought to be changed. Related parameters and values are sensibly grouped together and help is available through Python's built-in documentation system. Switching generators in Python is as simple as replacing one class by another. In Jupyter notebooks, generated events are automatically visualized by using the `pyhepmc` library [11]. An example is shown in Fig. 1.

## 3. Workflow

As an example `Chromo` workflow, we consinder to generate 1000 inelastic proton-oxygen collisions at 1 TeV in the center-of-mass frame. The parameters of a collision are encapsulated by the `CenterOfMass`, a subclass of the more general `EventKinematics` class. Here, we set collision parameters as `kinematics = CenterOfMass(1*TeV, "p", "O")`. Nuclei can be set as a tuple



PoS(ICRC2023)189

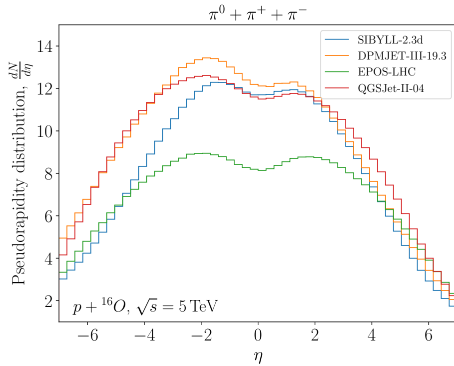


**Figure 2:** Pseudorapidity distribution of pions in proton-oxygen collisions at $\sqrt{s} = 5\,\text{TeV}$ for various hadronic interaction models.

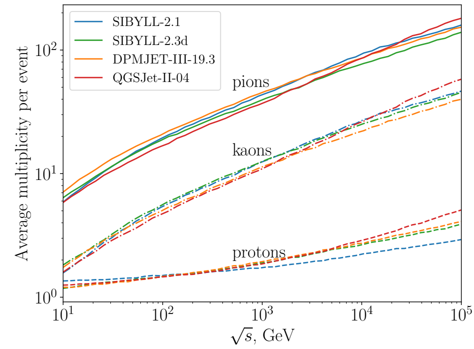


**Figure 3:** Energy dependence of average multiplicity per event in proton-proton interactions for various hadronic interaction models.

`(A, Z)`, with A as atomic mass and Z as charge, or by using the `CompositeTarget` class for multi-nuclei targets.

These parameters are passed to an event generator to simulate the interaction. `Chromo` supports multiple models which can be selected from the `models` module. An instance of the event generator is created by passing the `kinematics` object to a specific model's constructor, such as `event_generator = EposLHC(kinematics)`. An optional seed for the random number generator can be set with the `seed` keyword for reproducible events.

All models have similar methods, being subclasses of the abstract `MCRun` class. Attributes and methods of the `MCRun` class configure the event generator, such as `kinematics` and `random_state` properties to set and receive current `EventKinematics` object and seed. The `set_stable` and `set_unstable` methods specify where an unstable particle should be decayed by the generator or become part of the final state. The `cross_section` method returns a `CrossSectionData` object with cross section information.

Finally, 1000 events are generated with `for event in event_generator(1000)`. Each iteration returns an `MCEvent` object with collision results, which can be further processed. Fig. 2 and Fig. 3 demonstrate how Chromo enables effortless comparisons of common quantities of interest across multiple event generators.

## 4. Performance

Chromo was designed to generate events as fast as possible without incurring noticeable overhead compared to running the bare event generators. This is achievable even though most of the library is written Python, an interpreted language which runs a factor 100 to 1000 slower than fast compiled languages like Fortran or C++. The runtime required per generated event is limited from below by the runtime of the Fortran code in the wrapped event generator. We designed the Python code to add negligible overhead by avoiding hot loops in Python and minimizing unnecessary work such as copying memory between buffers.

To demonstrate the excellent performance of Chromo, we compare the event generation rate of proton-proton collisions at different center-of-mass energies to CRMC, which is written in C++, in



PoS(ICRC2023)189

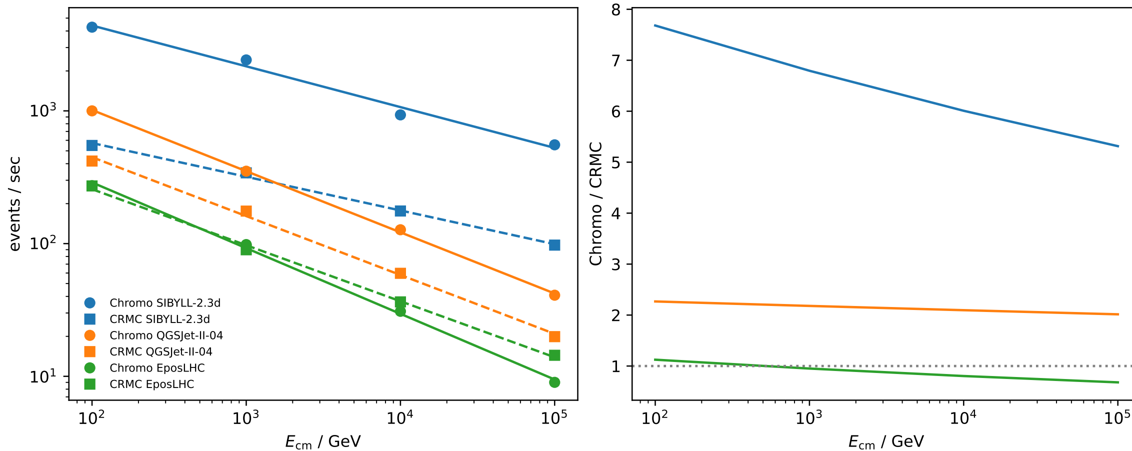


**Figure 4:** Event generation rate by the programs CRMC and Chromo for three event generators for proton-proton collisions as a function of the center-of-mass energy. Shown on the left-hand side are event rates, shown on the right-hand side are the rate ratios. The benchmark was run on an Intel 2.8 GHz Quad-Core i7.

Fig. 4. We use the models EPOS-LHC, QGSJet-II.04, and Sibyll-2.3d. The events were generated using the respective command-line interfaces and output was produced in the HepMC format.

In all tested cases, the event generation rate of Chromo is comparable to CRMC or much better (up to a factor 7). The gains scale with the overall event generation rate, which is largest for Sibyll-2.3d. No significant gains are observed for EPOS-LHC, whereas QGSJet-II.04 is running about twice as fast in Chromo. The gains are likely related to overheads in copying memory which Chromo avoids where possible. We probably see no gains for EPOS-LHC, because CRMC is based on EPOS-LHC code so that running EPOS-LHC is already optimized in CRMC.

Our comparison shows that one can obtain high performance with a software written in a slow interpreted language like Python, that is on par or even surpassing a fast compiled language like C++, if bottlenecks are identified and carefully avoided.

## 5. Program structure

Chromo is composed of multiple layers. It includes the original Fortran/C++ code for event generators, a custom Fortran/C++ layer for integration with Chromo, `F2PY/Pybind` instructions for building Python C/API extension modules, and a top layer of Python code that implements the library. The code primarily follows an object-oriented approach, with some functional-style code employed for internal auxiliary tasks. Inheritance is widely used to avoid code repetition and impose a common interface on similar classes. As a result, many classes form hierarchical structures based on inheritance. The core functionality is implemented by following distinct inheritance hierarchies.

The hierarchy handling interaction parameters is centered around `EventKinematics` class. It starts from `EventKinematicsBase` data class that stores details about the initial state of a collision, such as the particles involved, their energy or momenta, and the frame of reference. It also manages frame transformations. The `EventKinematics` subclass extends the data class with a flexible interface for specifying the initial state and frame. The subclasses `CenterOfMass` and `FixedTarget` provide simplified versions of general interface for common special cases.

PoS(ICRC2023)189

`MCRun` is an abstract base class that represents an event generator. Its abstract methods need to be implemented by concrete subclasses, which ensures that each generator adheres to the common interface. The constructor accepts an `EventKinematicsBase` object and a seed for the random number generator. Methods and properties allow access and modification of event kinematics, the state of random number generator, and to specify which unstable particles should be decayed by the event generator. For a given initial state, the event generator can return a `CrossSectionData` object that contains cross section information. Event generation follows the conventional generator protocol in Python, where events are generated within a for-loop for a specified number of iterations.

`EventData` is a data class that mainly stores information about the internal history and final state of the collision, such as the particles produced with their particle ID, momentum, energy, mass, production vertex, parents, and children. Each of these quantities is represented collectively by a NumPy array with entries corresponding to specific particles. The class has methods and properties that return useful derived quantities such as transverse momentum, rapidity, pseudorapidity, and Feynman $x$. Also, it provides basic functionality for making deep copies and particle filtering. `MCEvent` derives from `EventData` and provides a common infrastructure for the creation of `EventData` instances by a specific event generator. The specific implementations are provided by subclasses of `MCEvent`. Wherever possible, these implementations create NumPy arrays as views into the memory in Fortran common blocks or C++ vectors to reduce unnecessary copies.

The serialization of events into file formats common in particle physics is carried out by writer classes `Root` and `HepMC`. An `Svg` writer saves a graph of each event in the SVG image format. It is useful for illustration and debugging purposes. These classes are designed as Python context managers, which allows them to perform automatic cleanup actions when the writing task is complete. The writer object has a single public method `write`, which accepts an `EventData` object and converts it into the appropriate internal representation. Some writers implement internal buffering to improve the writing speed by minimising the number of disk accesses.

Chromo provides a command line interface (CLI) that mimics the CRMC interface to ease the transition for former CRMC users. The CLI allows one to generate events for various initial states and save the formats previously described. The CLI was designed to give a good user experience by providing comprehensive help output and a flexible system to select models via a string. Model names do not need to be spelled out completely if the already provided string is not ambiguous. In case of ambiguity, a list of matching models is printed. The CLI also prints informative summaries on the initial state and configuration, as well as a progress bar with the current status of a task, the estimated remaining run time, and the current generation rate in events per second.

To conclude, a brief tour of the modules in Chromo is provided. `kinematics` contains classes that describe the initial state of the collision and handles transformations between different frames of reference. `common` contains the abstract base classes that define the unified interfaces to the generator and its output, `MCRun`, `EventData`, and `MCEvent`. `models` contains model-specific classes derived from `MCRun` and `MCEvent` which implement the specific translation code between the respective model and the unified interface. `writer` includes classes for saving events in the HepMC, Root, and Svg file formats. `cli` contains functions that implement the command-line interface (CLI). `constants` collects physical constants, unit conversion factors, and global default parameters. `util` contains various helper functions and classes used internally by other classes.



PoS(ICRC2023)189

## 6. Automated validation and distribution

In order to save end-users from the time-consuming and potentially cumbersome build process, Chromo is distributed as Python wheels through `PyPI` for a variety of platforms and Python versions. At present, Chromo wheels are available for Windows 64-bit, Linux 64-bit, macOS Intel, and macOS Apple Silicon platforms, and Python versions 3.8, 3.9, 3.10, and 3.11. A wheel consists of about 20 pre-compiled extension modules. The compilation and wheel construction is automated using `CMake`, which is integrated with the `setuptools` packaging system. A wheel needs to be compiled for each combination of platform and Python version. It is essential to test and validate each wheel prior to distribution. The Chromo project employs established CI/CD principles and uses unit tests, GitHub Actions, and the `cibuildwheels` package to build, test, and deploy the software.

Any code changes committed to the GitHub repository initiate the `pre-commit.ci` code style validation and the `test` workflow. The latter compiles, builds, and installs Chromo on Windows, Ubuntu, and macOS. The installed package is extensively tested using a set of 790 unit tests, managed by the `pytest` framework. Each module of `Chromo` is scrutinized, with a substantial number of tests (approximately 580) dedicated to evaluating the results of event generators across various permutations of projectiles, targets, and reference frames. The Monte Carlo methods of the event generators are naturally sensitive to small changes in floating point calculations that stem from differences between mathematical libraries, such as `glibc`, among the different operating systems. It is therefore not possible to perform bitwise comparisons among the different platforms. Instead, we test the correctness of the output by performing probabilistic comparisons of event distributions with respect to pre-generated reference values.

The `release` workflow builds wheels for all combinations of platforms and Python versions, which are automatically tested and uploaded to `PyPI` if all tests pass. This task is largely automated by the `cibuildwheel` tool, which performs the required steps to create system-agnostic wheels.

## 7. Summary and future development

In this paper, we introduced Chromo, a Python frontend that provides a unified interface to popular generators of hadronic interactions. Event generators such as EPOS, DPMJet, QGSJet, Sibyll, and Pythia, which are integral for simulating air showers or minimum bias events at colliders, can be managed efficiently using Chromo. As a thin wrapper over these codes, Chromo ensures no significant performance penalty is imposed.

The ease of installation and integration with Python's interactive environment are some of Chromo's distinguishing features. It is designed to be user-friendly, minimizing the necessity of dealing with unique procedures to define kinematics, generate events, and parse outputs associated with each event generator.

Chromo comes with event visualization capabilities, support for multiple event generators, and efficient memory usage. It offers an intuitive object-oriented interface, thus making it a valuable tool for both experts and beginners. In addition, Chromo utilizes the Python packaging architecture and infrastructure, making it easy to install on various operating systems including Linux, macOS, and Windows.



PoS(ICRC2023)189

The tool has been extensively used by the authors for the development and testing of the DPMJet and Sibyll event generators, for event simulation at the LHCb experiment. Non-expert users have used the code successfully for various HEP and astroparticle-related studies. It is the central tool for the computing of secondary particle distributions in the MCEq cascade solver. Going forward, Chromo aims to continue addressing challenges associated with event generators and contribute to the broader realm of particle and astroparticle physics.

## Acknowledgements

HD acknowledges funding by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – project no. 449728698. This work acknowledges the resources at the Academia Sinica Grid Computing Center (ASGC), supported by the Institute of Physics of Academia Sinica.

## References

[1] Johannes Albrecht et al. "The Muon Puzzle in cosmic-ray induced air showers and its connection to the Large Hadron Collider". In: *Astrophys. Space Sci.* 367.3 (2022), p. 27. DOI: 10.1007/s10509-022-04054-5. arXiv: 2105.06148 [astro-ph.HE].

[2] Felix Riehn et al. "Hadronic interaction model Sibyll 2.3d and extensive air showers". In: *Phys. Rev. D* 102.6 (2020), p. 063002. DOI: 10.1103/PhysRevD.102.063002. arXiv: 1912.03300 [hep-ph].

[3] Anatoli Fedynitch et al. "Hadronic interaction model sibyll 2.3c and inclusive lepton fluxes". In: *Phys. Rev. D* 100.10 (2019), p. 103018. DOI: 10.1103/PhysRevD.100.103018. arXiv: 1806.04140 [hep-ph].

[4] S. Roesler, R. Engel, and J. Ranft. "The Monte Carlo event generator DPMJET-III". In: *Advanced Monte Carlo for radiation physics, particle transport simulation and applications.* 2001, pp. 1033–1038. DOI: 10.1007/978-3-642-18211-2_166. arXiv: hep-ph/0012252.

[5] Anatoli Fedynitch. "Cascade equations and hadronic interactions at very high energies". PhD thesis. KIT, Karlsruhe, Dept. Phys., Nov. 2015. DOI: 10.5445/IR/1000055433.

[6] Torbjorn Sjostrand, Stephen Mrenna, and Peter Z. Skands. "PYTHIA 6.4 Physics and Manual". In: *JHEP* 05 (2006), p. 026. DOI: 10.1088/1126-6708/2006/05/026. arXiv: hep-ph/0603175.

[7] Christian Bierlich et al. "A comprehensive guide to the physics and usage of PYTHIA 8.3". In: (Mar. 2022). DOI: 10.21468/SciPostPhysCodeb.8. arXiv: 2203.11601 [hep-ph].

[8] T. Pierog et al. "EPOS LHC: Test of collective hadronization with data measured at the CERN Large Hadron Collider". In: *Phys. Rev.* C92.3 (2015), p. 034906. DOI: 10.1103/PhysRevC.92.034906. arXiv: 1306.0121 [hep-ph].

[9] Ralf Ulrich, Tanguy Pierog, and Colin Baus. *Cosmic Ray Monte Carlo Package, CRMC*. Version 2.0.1. Aug. 2021. DOI: 10.5281/zenodo.5270381. URL: https://doi.org/10.5281/zenodo.5270381.

[10] Sergey Ostapchenko. "Monte Carlo treatment of hadronic interactions in enhanced Pomeron scheme: I. QGSJET-II model". In: *Phys. Rev.* D83 (2011), p. 014018. DOI: 10.1103/PhysRevD.83.014018. arXiv: 1010.1869 [hep-ph].

[11] Hans Dembinski et al. *scikit-hep/pyhepmc: v2.12.0*. Version v2.12.0. Feb. 2023. DOI: 10.5281/zenodo.7614602. URL: https://doi.org/10.5281/zenodo.7614602.



PoS(ICRC2023)189