CORSIKA 8 Documentation

Release prototype-0.1.0

CORSIKA 8 Collaboration

Aug 21, 2023

CONTENTS

1 CORSIKA 8 Framework for Particle Cascades in Astroparticle Physics **3**
- 1.1 Get in contact . . . 3
- 1.2 Installation . . . 4
- 1.3 Installation (using docker containers) . . . 5
- 1.4 Compiling . . . 5
- 1.5 Runing Unit Tests . . . 5
- 1.6 Running examples . . . 5

2 Output **7**

3 Particle Storage and Stack **9**

4 Particle Properties **11**
- 4.1 Particle Classes . . . 11

5 Medium Properties **15**
- 5.1 Media Classes . . . 15

6 Physics Units **19**

7 Geometry and Environment **21**

8 Particle storage in memory **23**

9 Utilities **25**

10 Physics modules and processes **27**

11 Howto create new physics modules **39**

12 Reference Documentation **43**

Index **45**

Welcome to the CORSIKA 8 air shower simulation framework.

CHAPTER

ONE

CORSIKA 8 FRAMEWORK FOR PARTICLE CASCADES IN ASTROPARTICLE PHYSICS

The purpose of CORSIKA is to simulate any particle cascades in astroparticle physics or astrophysical context. A lot of emphasis is put on modularity, flexibility, completeness, validation and correctness. To boost computational efficiency different techniques are provided, like thinning or cascade equations. The aim is that CORSIKA remains the most comprehensive framework for simulating particle cascades with stochastic and continuous processes.

The software makes extensive use of static design patterns and compiler optimization. Thus, the most fundamental configuration decision of the user must be performed at compile time. At run time model parameters can still be changed.

CORSIKA 8 is by default released under the GPLv3 license. See license file which is part of every release and the source code.

If you use, or want to refer to, CORSIKA 8 please cite "Towards a Next Generation of CORSIKA: A Framework for the Simulation of Particle Cascades in Astroparticle Physics", Comput.Softw.Big Sci. 3 (2019) 2. We kindly ask (and require) any relevant improvement or addition to be offered or contributed to the main CORSIKA 8 repository for the benefit of the whole community.

When you plan to contribute to CORSIKA 8 check the guidelines outlined here: coding guidelines. Code that fails the review by the CORSIKA author group must be improved before it can be merged in the official code base. After your code has been accepted and merged, you become a contributor of the CORSIKA 8 project (code author).

IMPORTANT: Before you contribute, you need to read and agree to the collaboration agreement. The agreement can be discussed, and eventually improved.

We also want to point you to the MCnet guidelines, which are very useful also for us.

1.1 Get in contact

- Connect to https://gitlab.iap.kit.edu register yourself and join the "Air Shower Physics" group. Write to one of the steering comittee members (https://gitlab.iap.kit.edu/AirShowerPhysics/corsika/-/wikis/Steering-Committee) in case there are problems with that.
- Connect to corsika-devel@lists.kit.edu (self-register at https://www.lists.kit.edu/sympa/subscribe/corsika-devel) to get in touch with the project.
- Register on the corsika slack channel.

1.2 Installation

CORSIKA 8 is tested regularly at least on gcc11.0.0 and clang-14.0.0.

1.2.1 Prerequisites

You will also need:

- Python 3 (supported versions are Python >= 3.6), with pip
- conan (via pip)
- cmake
- git
- g++, gfortran, binutils, make

On a bare Ubuntu 20.04, just add:

```
sudo apt-get install python3 python3-pip cmake g++ gfortran git doxygen graphviz
```

On a bare CentOS 7 install python3, pip3 (pip from python3) and cmake3. Any of the devtools 7, 8, 9 should work (at least). Also initialize devtools, before building CORSIKA 8:

```
source /opt/rh/devtoolset-9/enable
```

CORSIKA 8 uses the conan package manager to manage our dependencies. Currently, version 1.55.0 or higher is required. If you do not have Conan installed, it can be installed with:

```
pip install --user conan~=1.57.0
```

1.2.2 Compiling

Once Conan is installed, follow these steps to download and install CORSIKA 8:

```
git clone --recursive git@gitlab.iap.kit.edu:AirShowerPhysics/corsika.git
mkdir corsika-build
cd corsika-build
../corsika/conan-install.sh
cmake ../corsika -DCMAKE_BUILD_TYPE="RelWithDebInfo" -DCMAKE_INSTALL_PREFIX=../corsika-
↪install
make -j8
make install
```

1.3 Installation (using docker containers)

There are docker containers prepared that bring all the environment and packages you need to run CORSIKA. See docker hub for a complete overview.

1.3.1 Prerequisites

You only need docker, e.g. on Ubuntu: `sudo apt-get install docker` and of course root access.

1.4 Compiling

Follow these steps to download and install CORSIKA 8, master development version

```
git clone --recursive https://gitlab.iap.kit.edu/AirShowerPhysics/corsika.git
sudo docker run -v $PWD:/corsika -it corsika/devel:clang-8 /bin/bash
mkdir build
cd build
../corsika/conan-install.sh
cmake ../corsika -DCMAKE_INSTALL_PREFIX=../corsika-install
make -j8
make install
```

1.5 Runing Unit Tests

To run the Unit Tests, just type `ctest` in your build area.

1.6 Running examples

To see how a relatively simple hadron cascade develops, see `examples/cascade_example.cpp` for a starting point.

To run the cascade_example, or any other CORSIKA 8 application, you must first compile it wrt. to the CORSIKA 8 header-only framework. This can be done best by copying e.g. `corsika-install/share/corsika/examples/` to your working place (e.g. `corsika-work`).

```
cd corsika-work
```

Next, you need to define the environment variable `corsika_DIR` to point to, either, your build, or your install area. Thus, e.g.

```
export corsika_DIR=<dir where you installed CORSIKA 8 to, or where you build it>
```

You also need to define the environment variable `CORSIKA_DATA` to point to your modules/data folder where you cloned the corsika repository. Thus, e.g.

```
export CORSIKA_DATA=<modules/data dir where you cloned CORSIKA 8 to>
```

Then compile your example/application with

```
cmake .
make
bin/cascade_example
```

1.6.1 Generating doxygen documentation

To generate the documentation, you need doxygen and graphviz. If you work with the docker corsika/devel containers this is already included. Otherwise, e.g. on Ubuntu 18.04, do:

```
sudo apt-get install doxygen graphviz
```

Switch to the corsika build directory and do

```
make docs
make install
```

open with firefox:

```
firefox ../corsika-install/share/corsika/doc/html/index.html
```

CHAPTER

TWO

OUTPUT

The format of CORSIKA 8 is designed to allow simple and robust managmenet of large libraries, as well as high reading performance. There is a dedicated python library to help processing data.

The basic structure of output is structured on the filesystem itself with a couple of subdirectories and files. Each run of CORSIKA~8 creates a library that can contain any number of showers. The format is equally suited for single huge showers as well as for a very high number of very low-energy showers. Each module included in the run can produce output inside this directory. The module output is separeted in individual user-named sub-directories, each containing files produced by the module. The file format is either yaml for basic configuration and summary data, or Apache parquet for any other (binary, compressed) data. Parquet is optimal for columnar/tabular data as it is produced by CORSIKA 8.

One advantage of this format is that with normal filesystem utilties users can manage the libraries. On all systems there are tools available to directly read/process yaml as well as parquet files. If you, for example, don't need the particle data for space reasons, this is very simple to remove from a library. Individual output stream (modules) can be easily separated with no extra effort.

For example, the output of the "vertical_EAS" example program looks like this:

```
vertical_EAS_outputs/
    config.yaml
    summary.yaml
    particles/
        config.yaml
        summary.yaml
        particles.parquet
```

The "vertical_EAS_outputs" and the "particles" are user-defined names and can be arranged/changed. But the type of data is well defined, e.g. in "particles" the data from an ObservationPlane object is stored. This is relevant, since it allows python to access this data in a controlled way.

The top level "config.yaml" contains top-level library information:

```
name: vertical_EAS_outputs
creator: CORSIKA8
version: 8.0.0-prealpha
```

and the "summary.yaml" is written in the very end (thus, the presence of the summary also indicates that a run is finished):

```
showers: 2
start time: 06/02/2021 23:46:18 HST
end time: 06/02/2021 23:46:42 HST
runtime: 00:00:24.260
```

Each module has its own "config.yaml" and "summary.yaml" file, too. To handle thus output for analysis any tool of your preference is feasible. We recommend python. There is a python library accompanied with CORSIKA~8 to facilitate analysis and output handling (>>> is python prompt):

```
>>> import corsika
>>> lib = corsika.Library("vertical_EAS_outputs")
>>> lib.config  # this gets the library configuration as a Python dictionary
{'name': 'vertical_EAS_outputs',
'creator': 'CORSIKA8',
'version': '8.0.0-prealpha'}
>>> lib.names  # get a list of all registered processes in the library
['particles']
>>> lib.summary  # you can also load the summary information
{'showers': 1,
'start time': '06/02/2021 23:46:18 HST',
'end time': '06/02/2021 23:46:30 HST',
'runtime': 11.13}
>>> lib.get("particles")  # you can then get the process by its registered name.
ObservationPlane('particles')
>>> lib.get("particles").config  # and you can also get its config as well
{'type': 'ObservationPlane',
'plane': {'center': [0, 0, 6371000],
'center.units': 'm',
'normal': [0, 0, 1]},
'x-axis': [1, 0, 0],
'y-axis': [0, 1, 0],
'delete_on_hit': True,
'name': 'particles'}
>>> lib.get("particles").data  # this returns the data as a Pandas data frame
   shower  pdg        energy         x         y    radius
0         0  211  9.066702e+10  2.449931 -5.913341  7.093710
1         0   22  2.403024e+11 -1.561504 -1.276160  2.024900
2         0  211  1.306354e+11 -4.626045 -3.237780  6.009696
3         0  211  1.773324e+11 -1.566567  4.172961  4.461556
4         0  211  7.835374e+10  3.152863 -1.049201  3.330416
..      ...  ...           ...       ...       ...       ...
>>> lib.get("particles").astype("arrow")  # you can also request the data in a different␣
↪format
pyarrow.Table
shower: int32 not null
pdg: int32 not null
energy: double not null
x: double not null
y: double not null
radius: double not null
>>>lib.get("particles").astype("pandas") # or astype("arrow"), or astype("pandas").to_
↪numpy()
```

You can locally install the corsika python analysis library from within your corsika source code directory by *pip3 install –user -e python pyarrow==0.17.0*. Note, the pyarrow version fix has shown to be needed on some older systems. You may not need this, or you may need additional packages, too.

CHAPTER

THREE

PARTICLE STORAGE AND STACK

Particles in memory are stored in a Stack. The Stack handles the memory management and access to particle data properties. The stack can be extended with additional fields that then become part of the particle object.

The standard Stack object in CORSIKA 8 is the NuclearStackExtension, further extended with internal fields like *geometry node* and *weight* or *history*. But the important part is the *NuclearStackExtension*.

There are several ways to create particles: * novel particles on the stack:

- stack::addParticle(particle_data_type const&)
- stack::addParticle(nuclear_particle_data_type const&)
- stack::addParticle(particle_data_momentum_type const&)
- stack::addParticle(nuclear_particle_data_momentum_type const&)
- secondary particles: * stack::addSecondary(parent const&, particle_data_type const&) * stack::addSecondary(parent const&, nuclear_particle_data_type const&) * stack::addSecondary(parent const&, particle_data_momentum_type const&) * stack::addSecondary(parent const&, nuclear_particle_data_momentum_type const&) or directly: * particle::addSecondary(particle_data_type const&) * particle::addSecondary(nuclear_particle_data_type const&) * particle::addSecondary(particle_data_momentum_type const&) * particle::addSecondary(nuclear_particle_data_momentum_type const&)

The content of these method parameters are:

- particle_data_type: {PID [corsika::Code], kinetic energy [HEPEnergyType], direction [DirectionVector], position [Point], time[TimeType]}
- particle_data_momentum_type: {PID [corsika::Code], momentum [MomentumVector], position [Point], time[TimeType]}
- nuclear_particle_data_type: {PID [corsika::Code], kinetic energy [HEPEnergyType], direction [DirectionVector], position [Point], time[TimeType], A [unsigned int], Z [unsigned int]}
- nuclear_particle_data_momentum_type: {PID [corsika::Code], momentum [MomentumVector], position [Point], time[TimeType], A [unsigned int], Z [unsigned int]}

CHAPTER FOUR

PARTICLE PROPERTIES

4.1 Particle Classes

Warning: doxygengroup: Cannot find group "ParticleClasses" in doxygen xml output for project "CORSIKA8" from directory: _build/workdir/doxygen/xml

group **Particles**

The properties of all particles are saved in static and flat arrays.

There is a enum corsika::Code to identify each particle, and each individual particle has its own static class, which can be used to retrieve its physical properties.

The properties of all elementary particles are accessible here. The data are taken from the Pythia ParticleData.xml file.

Particle data can be accessed via global function in namespace corsika, or via static classes for each particle type. These classes all have the interface (example for the class corsika::Electron):

```
static constexpr Code code{Code::Electron};
static constexpr Code anti_code{Code::Positron};
static constexpr HEPMassType mass{corsika::get_mass(code)};
static constexpr ElectricChargeType charge{corsika::get_charge(code)};
static constexpr int charge_number{corsika::get_charge_number(code)};
static constexpr std::string_view name{corsika::get_name(code)};
static constexpr bool is_nucleus{corsika::is_nucleus(code)};
```

The names, relations and properties of all particles known to CORSIKA 8 are listed below.

Note on energy threshold on particle production as well as particle propagation. The functions:

```
HEPEnergyType constexpr get_energy_production_threshold(Code const);
void constexpr set_energy_production_threshold(Code const, HEPEnergyType const);
```

can be used to tune the transition where explicit production of new particles, e.g. in Bremsstrahlung, is simulated versus a continuous handling of low-energy particles as generic energy losses. The default value for all particle types is 1 MeV.

Furthermore, the functions:

```
HEPEnergyType constexpr get_kinetic_energy_propagation_threshold(Code const);
void constexpr set_kinetic_energy_propagation_threshold(Code const, HEPEnergyType
                                                        const);
```

are used to discard low energy particle during tracking. The default value for all particle types is 1 GeV.

Functions

int16_t constexpr **`get_charge_number`**(Code const)

electric charge in units of e

ElectricChargeType constexpr **`get_charge`**(Code const)

electric charge

HEPMassType constexpr **`get_mass`**(Code const)

mass

HEPEnergyType **`get_kinetic_energy_propagation_threshold`**(Code const)

Get the kinetic energy propagation threshold.

Particles are tracked only above the kinetic energy propagation threshold. Below this, they are discarded and removed. Sensible default values must be configured for a simulation.

void **`set_kinetic_energy_propagation_threshold`**(Code const, HEPEnergyType const)

Set the kinetic energy propagation threshold object.

HEPEnergyType **`get_energy_production_threshold`**(Code const)

Get the particle production energy threshold.

The (total) energy below which a particle is only handled stoachastically (no production below this energy). This is for example important for stochastic discrete Bremsstrahlung versus low-energy Bremsstrahlung as part of continuous energy losses.

void **`set_energy_production_threshold`**(Code const, HEPEnergyType const)

Set the particle production energy threshold in total energies.

PDGCode constexpr **`get_PDG`**(Code const)

Particle code according to PDG, "Monte Carlo Particle Numbering Scheme".

PDGCode constexpr **`get_PDG`**(unsigned int const A, unsigned int const Z)

std::string_view constexpr **`get_name`**(Code const)

name of the particle as string

std::string **`get_name`**(Code, full_name)

get name of particle, including (A,Z) for nuclei

TimeType constexpr **`get_lifetime`**(Code const)

lifetime

bool constexpr **`is_hadron`**(Code const)

true if particle is hadron

bool constexpr **`is_em`**(Code const)

true if particle is electron, positron or photon

bool constexpr `is_muon`(Code const)
: true if particle is mu+ or mu-

bool constexpr `is_neutrino`(Code const)
: true if particle is (anti-) neutrino

bool constexpr `is_charged`(Code const)
: true if particle is charged

Code constexpr `get_nucleus_code`(size_t const A, size_t const Z)
: Creates the Code for a nucleus of type 10LZZZAAAI.

  Returns
  internal nucleus Code

bool constexpr `is_nucleus`(Code const)
: Checks if Code corresponds to a nucleus.

  Returns
  true if nucleus.

  Returns
  false if not nucleus.

size_t constexpr `get_nucleus_A`(Code const)
: Get the mass number A for nucleus.

  Returns
  int size of nucleus.returns A for hard-coded nucleus, otherwise 0

size_t constexpr `get_nucleus_Z`(Code const)
: Get the charge number Z for nucleus.

  Returns
  int charge of nucleus.returns Z for hard-coded nucleus, otherwise 0

HEPMassType constexpr `get_nucleus_mass`(Code const code)
: Calculates the mass of nucleus.

  Returns
  HEPMassType the mass of (A,Z) nucleus, disregarding binding energy.

HEPMassType constexpr `get_nucleus_mass`(unsigned int const A, unsigned int const Z)
: Calculates the mass of nucleus.

  Returns
  HEPMassType the mass of (A,Z) nucleus, disregarding binding energy.

inline std::string `get_nucleus_name`(Code const code)
: Get the nucleus name.

  Parameters
  `code` –

  Returns
  std::string_view

Code `convert_from_PDG`(PDGCode const)
: convert PDG code to CORSIKA 8 internal code.

  Returns
  Code internal code.

std::initializer_list<Code> constexpr **get_all_particles**()

Returns list of all non-nuclei particles.

Returns
std::initializer_list<Code> constexpr

std::ostream &**operator<<**(std::ostream&, corsika::Code)

Code output operator.

The output stream operator for human-readable particle codes.

Returns
std::ostream&

struct **full_name**

#include <ParticleProperties.hpp> tag class for *get_name()*

CHAPTER

FIVE

MEDIUM PROPERTIES

5.1 Media Classes

Warning: doxygengroup: Cannot find group "MediaPropertiesClasses" in doxygen xml output for project "CORSIKA8" from directory: _build/workdir/doxygen/xml

group **MediaProperties**

Medium types are useful most importantly for effective models like energy losses.

a particular medium (mixture of components) may have specif properties not reflected by its mixture of components.

The data provided here is automatically parsed from the file properties8.dat from NIST.

The data of each known medium can be access via the global functions in namespace corsika, or via a static class object with the following interface (here at the example of the class HydrogenGas):

```
static constexpr Medium medium() { return Medium::HydrogenGas; }

static std::string const getName() { return data_.getName(); }
static std::string const getPrettyName() { return data_.getPrettyName(); }
static double getWeight() { return data_.getWeight (); }
static int weight_significant_figure() { return data_.weight_significant_figure ();␣
↪}
static int weight_error_last_digit() { return data_.weight_error_last_digit(); }
static double Z_over_A() { return data_.Z_over_A(); }
static double getSternheimerDensity() { return data_.getSternheimerDensity(); }
static double getCorrectedDensity() { return data_.getCorrectedDensity(); }
static StateOfMatter getStateOfMatter() { return data_.getStateOfMatter(); }
static MediumType getType() { return data_.getType(); }
static std::string const getSymbol() { return data_.getSymbol(); }

static double getIeff() { return data_.getIeff(); }
static double getCbar() { return data_.getCbar(); }
static double getX0() { return data_.getX0(); }
static double getX1() { return data_.getX1(); }
static double getAA() { return data_.getAA(); }
static double getSK() { return data_.getSK(); }
static double getDlt0() { return data_.getDlt0(); }
```

(continues on next page)

(continued from previous page)

```
inline static const MediumData data_ { "hydrogen_gas", "hydrogen gas (H%2#)", 1.008,
3, 7, 0.99212,
8.3748e-05, 8.3755e-05, StateOfMatter::DiatomicGas,
MediumType::Element, "H", 19.2, 9.5835, 1.8639, 3.2718, 0.14092, 5.7273, 0.0 };
```

The numeric data known to CORSIKA 8 (and obtained from NIST) can be browsed in MediaPropertiesClasses.

Enums

enum **MediumType**

General type of medium.

Values:

enumerator **Unknown**

enumerator **Element**

enumerator **RadioactiveElement**

enumerator **InorganicCompound**

enumerator **OrganicCompound**

enumerator **Polymer**

enumerator **Mixture**

enumerator **BiologicalDosimetry**

enum **StateOfMatter**

Physical state of medium.

Values:

enumerator **Unknown**

enumerator **Solid**

enumerator **Liquid**

enumerator **Gas**

enumerator **DiatomicGas**

struct **MediumData**

#include <MediumProperties.hpp> Enum for all known Media types.

The internal integer type of a medium enum.

Simple object to group together the properties of a medium.

MediumDataInterface Interface methods

Interface functions for *MediumData*

inline std::string **getPrettyName**() const

returns name

inline double **getWeight**() const

returns pretty name

inline const int &**weight_significant_figure**() const

return weight

inline const int &**weight_error_last_digit**() const

return significnat figures of weight

inline const double &**Z_over_A**() const

return error of weight

inline double **getSternheimerDensity**() const

Z_over_A_.

inline double **getCorrectedDensity**() const

Sternheimer density.

inline *StateOfMatter* **getStateOfMatter**() const

corrected density

inline *MediumType* **getType**() const

state

inline std::string **getSymbol**() const

type

inline double **getIeff**() const

symbol

inline double **getCbar**() const

Ieff.

inline double **getX0**() const

Cbar.

inline double **getX1**() const

X0.

inline double **getAA**() const

X1.

inline double **`getSK()`** const

AA.

inline double **`getDlt0()`** const

Sk.

CHAPTER

SIX

PHYSICS UNITS

Not yet documented in sphinx. Check doxygen, examples, tests.

CHAPTER

SEVEN

GEOMETRY AND ENVIRONMENT

Not yet documented in sphinx. Check doxygen, examples, tests.

CHAPTER

EIGHT

PARTICLE STORAGE IN MEMORY

Not yet documented in sphinx. Check doxygen, examples, tests.

CHAPTER

NINE

UTILITIES

group **`Utilities`**

Collection of classes and methods to perform recurring tasks.

class **`COMBoost`**

#include <COMBoost.hpp> This utility class handles Lorentz boost (in one spatial direction) between different reference frames, using FourVector.

The class is initialized with projectile and optionally target energy/momentum data. During initialization, a rotation matrix is calculated to represent the projectile movement (and thus the boost) along the z-axis. Also the inverse of this rotation is calculated. The Lorentz boost matrix and its inverse are determined as 2x2 matrices considering the energy and pz-momentum.

Different constructors are offered with different specialization for the cases of collisions (projectile-target) or just decays (projectile only).

Public Functions

`COMBoost``(`FourMomentum const &P4projectile, HEPEnergyType const massTarget`)`

Construct a *COMBoost* given four-vector of projectile and mass of target (target at rest).

The FourMomentum and mass define the lab system.

`COMBoost``(`MomentumVector const &momentum, HEPEnergyType const mass`)`

Construct a *COMBoost* to boost into the rest frame of a particle given its 3-momentum and mass.

`COMBoost``(`FourMomentum const &P4projectile, FourMomentum const &P4target`)`

Construct a *COMBoost* given two four-vectors of projectile target.

The two FourMomentum can define an arbitrary system.

template<typename **`FourVector`**>
FourVector **`toCoM`**`(`*FourVector* const &p4`)` const

transforms a 4-momentum from lab frame to the center-of-mass frame

template<typename **`FourVector`**>
FourVector **`fromCoM`**`(`*FourVector* const &p4`)` const

transforms a 4-momentum from the center-of-mass frame back to lab frame

CoordinateSystemPtr const &**`getRotatedCS`**`()` const

returns the rotated coordinate system: +z is projectile direction

CoordinateSystemPtr const &**getOriginalCS**() const

returns the original coordinate system of the projectile (lab)

namespace **andre**

Functions

std::vector<double> **solveP3**(long double a, long double b, long double c, long double d, double const epsilon = 1e-12)

Solve a x^3 + b x^2 + c x + d = 0.

std::vector<double> **solve_quartic_real**(long double a, long double b, long double c, long double d, long double e, double const epsilon = 1e-12)

solve quartic equation a*x^4 + b*x^3 + c*x^2 + d*x + e Attention - this function returns dynamically allocated array.

It has to be released afterwards.

CHAPTER

TEN

PHYSICS MODULES AND PROCESSES

group **Processes**

Physics processes in CORSIKA 8 are clustered in *ProcessSequence* and *SwitchProcessSequence* containers.

The former is a mere (ordered) collection, while the latter has the option to switch between two alternative ProcessSequences.

Depending on the type of data to act on and on the allowed actions of processes there are several interface options:

- *InteractionProcess*
- *DecayProcess*
- *ContinuousProcess*
- *StackProcess*
- *SecondariesProcess*
- *BoundaryCrossingProcess*

And all processes (including *ProcessSequence* and *SwitchProcessSequence*) are derived from *BaseProcess*.

Processes of any type (e.g. p1, p2, p3,. . .) can be assembled into a *ProcessSequence* using the `make_sequence` factory function.

```
auto sequence1 = make_sequence(p1, p2, p3);
auto sequence2 = make_sequence(p4, p5, p6, p7);
auto sequence3 = make_sequence(sequence1, sequemce2, p8, p9);
```

Note, if the order of processes matters, the order of occurence in the *ProcessSequence* determines the executiion order.

SecondariesProcess alyways act on new secondaries produced (i.e. in *InteractionProcess* and *DecayProcess*) in the scope of their *ProcessSequence*. For example if i1 and i2 are InteractionProcesses and s1 is a *SecondariesProcess*, then:

```
auto sequence = make_sequence(i1, make_sequence(i2, s1))
```

will result in s1 acting only on the particles produced by i2 and not by i1. This can be very useful, e.g. to fine tune thinning.

A special type of *ProcessSequence* is *SwitchProcessSequence*, which has two branches and a functor that can select between these two branches.

```
auto sequence = make_switch(sequence1, sequence2, selector);
```

where the only requirement to `selector` is that it provides a `SwitchResult operator()(Particle const& particle) const` method. Thus, based on the dynamic properties of `particle` the functor can make its decision. This is clearly important for switching between low-energy and high-energy models, but not limited to this. The selection can even be done with a lambda function.

template<typename **TDerived**>

struct **BaseProcess**

#include <BaseProcess.hpp> Each process in C8 must derive from *BaseProcess*.

The structural base type of a process object in a *ProcessSequence*. Both, the *ProcessSequence* and all its elements are of type *BaseProcess*.

Todo:

rename *BaseProcess* into just Process

Subclassed by *corsika::BoundaryCrossingProcess< TDerived >*, *corsika::CascadeEquationsProcess< TDerived >*, *corsika::ContinuousProcess< TDerived >*, *corsika::DecayProcess< TDerived >*, *corsika::SecondariesProcess< TDerived >*, *corsika::StackProcess< TDerived >*

Public Types

using **process_type** = *TDerived*

Base processor type for use in other template classes.

Public Static Functions

static inline size_t constexpr **getNumberOfProcesses**()

Default number of processes is just one, obviously.

template<typename **TDerived**>

class **BoundaryCrossingProcess** : public corsika::*BaseProcess*<*TDerived*>

#include <BoundaryCrossingProcess.hpp> Processes acting on the particles traversion from one volume into another volume.

Create a new *BoundaryCrossingProcess*, e.g. for XYModel, via

```
class XYModel : public BoundaryCrossingProcess<XYModel> {};
```

and provide the necessary interface method:

```
template <typename TParticle>
ProcessReturn XYModel::doBoundaryCrossing(TParticle& Particle,
                                    typename TParticle::node_type const& from,
                                    typename TParticle::node_type const& to);
```

where Particle is the object to read particle data from a Stack. The volume the particle is originating from is `from`, the volume where it goes to is `to`.

template<typename **TDerived**>

class **CascadeEquationsProcess** : public corsika::*BaseProcess*<*TDerived*>

#include <CascadeEquationsProcess.hpp> Processes executing cascade-equations calculations.

Create a new *CascadeEquationsProcess*, e.g. for XYModel, via:

```
class XYModel : public CascadeEquationsProcess<XYModel> {};
```

and provide the necessary interface method:

```
template <typename TStack>
void doCascadeEquations(TStack& stack);
```

Cascade equation processes may generate new particles on the stack. They also typically generate output.

template<typename **TDerived**>

class **ContinuousProcess** : public corsika::*BaseProcess*<*TDerived*>

#include <ContinuousProcess.hpp> Processes with continuous effects along a particle Trajectory.

Create a new *ContinuousProcess*, e.g. for XYModel, via:

```
class XYModel : public ContinuousProcess<XYModel> {};
```

and provide two necessary interface methods:

```
template <typename TParticle, typename TTrack>
LengthType getMaxStepLength(TParticle const& p, TTrack const& track) const;
```

which allows any *ContinuousProcess* to tell to CORSIKA a maximum allowed step length. Such step-length limitation, if it turns out to be smaller/sooner than any other limit (decay length, interaction length, other continuous processes, geometry, etc.) will lead to a limited step length.

```
template <typename TParticle, typename TTrack>
ProcessReturn doContinuous(TParticle& p, TTrack const& t, bool const stepLimit)
const;
```

which applied any continuous effects on Particle p along Trajectory t. The particle in all typical scenarios will be altered by a doContinuous. The flag stepLimit will be true if the preious evaluation of getMaxStepLength resulted in this particular *ContinuousProcess* to be responsible for the step length limit on the current track t. This information can be expoited and avoid e.g. any uncessary calculations.

Particle and Track are the valid classes to access particles and track (Trajectory) data on the Stack. Those two methods do not need to be templated, they could use the types e.g. corsika::setup::Stack::particle_type &#8212; but by the cost of loosing all flexibility otherwise provided.

class **ContinuousProcessIndex**

#include <ContinuousProcessIndex.hpp> To index individual processes (continuous processes) inside a *ProcessSequence*.

class **ContinuousProcessStepLength**

#include <ContinuousProcessStepLength.hpp> To store step length in LengthType and unique index in *ProcessSequence* of shortest step *ContinuousProcess*.

template<typename **TDerived**>

struct **DecayProcess** : public corsika::*BaseProcess*<*TDerived*>

#include <DecayProcess.hpp> Process decribing the decay of particles.

Create a new *DecayProcess*, e.g. for XYModel, via

```
class XYModel : public DecayProcess<XYModel> {};
```

and provide the two necessary interface methods

```
template <typename TSecondaryView>
void XYModel::doDecay(TSecondaryView&);

template <typename TParticle>
TimeType getLifetime(TParticle const&)
```

Where, of course, SecondaryView and Particle are the valid classes to access particles on the Stack. In user code those two methods do not need to be templated, they could use the types e.g. corsika::setup::Stack::particle_type &#8212; but by the cost of loosing all flexibility otherwise provided.

SecondaryView allows to retrieve the properties of the projectile particles, AND to store new particles (secondaries) which then subsequently can be processes by *SecondariesProcess*. This is how the output of decays can be studied right away.

template<class **TCountedProcess**>

class **InteractionCounter** : public corsika::*InteractionProcess*<*InteractionCounter*<*TCountedProcess*>>

#include <InteractionCounter.hpp> Wrapper around an *InteractionProcess* that fills histograms of the number of calls to `doInteraction()` binned in projectile energy (both in lab and center-of-mass frame) and species.

Use by wrapping a normal *InteractionProcess*:

```
InteractionProcess collision1;
InteractionClounter<collision1> counted_collision1;
```

Public Functions

template<typename **TSecondaryView**>
void **doInteraction**(*TSecondaryView* &view, Code const, Code const, FourMomentum const&, FourMomentum const&)

Wrapper around internal process doInteraction.

CrossSectionType **getCrossSection**(Code const, Code const, FourMomentum const&, FourMomentum const&) const

Wrapper around internal process getCrossSection.

InteractionHistogram const &**getHistogram**() const

returns the filles histograms.

Returns

InteractionHistogram, which contains the histogram data

class **InteractionHistogram**

#include <InteractionHistogram.hpp> Class that creates and stores histograms of collisions dN/dE_{lab}, $dN/d\sqrt{s}$ which is used by class *InteractionCounter*.

Histograms are of type boost::histogram

Public Functions

void **fill**(Code const projectile_id, HEPEnergyType const lab_energy, HEPEnergyType const mass_target)

fill both CMS and lab histograms at the same time

Parameters

- **projectile_id** – corsika::Code of particle
- **lab_energy** – Energy in lab. frame
- **mass_target** – Mass of target particle
- **A** – if projectile_id is Nucleus : Mass of nucleus
- **Z** – if projectile_id is Nucleus : Charge of nucleus

inline hist_type const &**CMSHist**() const

return histogram in c.m.s. frame

inline hist_type const &**labHist**() const

return histogram in laboratory frame

template<class **TUnderlyingProcess**>

class **InteractionLengthModifier** : public corsika::*InteractionProcess*<*InteractionLengthModifier*<*TUnderlyingProcess*>>

#include <InteractionLengthModifier.hpp> Wrapper around an *InteractionProcess* that allows modifying the interaction length returned by the underlying process in a user-defined way. *

Public Types

using **functor_signature** = GrammageType(GrammageType, corsika::Code, HEPEnergyType)

The signature of the modifying functor. Arguments are original int. length, PID, energy.

Public Functions

InteractionLengthModifier(*TUnderlyingProcess* &&process, std::function<*functor_signature*> modifier)

Create wrapper around *InteractionProcess*.

Note that the passed process object itself may no longer be used, only through this class.

template<typename **TSecondaryView**>
void **doInteraction**(*TSecondaryView* &view)

wrapper around internal process doInteraction

template<typename **TParticle**>
GrammageType **getInteractionLength**(*TParticle* const &particle)

! returns underlying process getInteractionLength modified

TUnderlyingProcess const &**getProcess**() const

! obtain reference to wrapped process

template<typename **TModel**>

class **InteractionProcess** : public corsika::*BaseProcess*<*TModel*>

#include <InteractionProcess.hpp> Process describing the interaction of particles.

Create a new *InteractionProcess*, e.g. for XYModel, via:

```
class XYModel : public InteractionProcess<XYModel> {};
```

and provide the two necessary interface methods:

```
template <typename TSecondaryView>
void XYModel::doInteraction(TSecondaryView&);

template <typename TParticle>
GrammageType XYModel::getInteractionLength(TParticle const&)
```

Where, of course, SecondaryView and Particle are the valid classes to access particles on the Stack. In user code, those two methods do not need to be templated, they could use the types e.g. corsika::setup::Stack::particle_type &#8212; but by the cost of loosing all flexibility otherwise provided.

SecondaryView allows to retrieve the properties of the projectile particles, AND to store new particles (secondaries) which then subsequently can be processes by *SecondariesProcess*. This is how the output of interactions can be studied right away.

class **NullModel**

#include <NullModel.hpp> Process that does nothing.

It is not even derived from *BaseProcess*. But it can be added to a *ProcessSequence*.

Public Static Functions

static inline size_t constexpr **getNumberOfProcesses**()

Default number of processes is zero, obviously.

template<typename **TDerived**>

class **SecondariesProcess** : public corsika::*BaseProcess*<*TDerived*>

#include <SecondariesProcess.hpp> Processes acting on the secondaries produced by other processes.

Create a new *SecondariesProcess*, e.g. for XYModel, via

```
class XYModel : public SecondariesProcess<XYModel> {};
```

and provide the necessary interface method:

```
template <typename TStackView>
void doSecondaries(TStackView& StackView);
```

where StackView is an object that can store secondaries on a stack and also iterate over these secondaries.

template<typename **TDerived**>

class **StackProcess** : public corsika::*BaseProcess*<*TDerived*>

#include <StackProcess.hpp> Process to act on the entire particle stack.

Create a new *StackProcess*, e.g. for XYModel, via:

```
class XYModel : public StackProcess<XYModel> {};
```

and provide the necessary interface method:

```
template <typename TStack>
void XYModel::doStack(TStack&);
```

Where, of course, Stack is the valid class to access particles on the Stack. This methods does not need to be templated, they could use the types e.g. `corsika::setup::Stack` directly &#8212; but by the cost of loosing all flexibility otherwise provided.

A *StackProcess* has only one constructor `StackProcess::StackProcess(unsigned int const nStep)` where nStep (n_{step}) is the number of steps of the cascade stepping after which the stack process should be run. Good values are on the order of 1000, which will not compromise run time in the end, but provide all the benefits of the *StackProcess*.

The number of *steps* during the cascade processing after which a *StackProcess* is going to be executed is determined from `iStep_` and `nStep_` using the modulo: $!(iStep\_ \% nStep\_)$. And `iStep_` is increased for each evaluation (step).

Public Functions

inline int **getStep**() const

return the current Cascade step counter

inline bool **checkStep**()

check if current step is where *StackProcess* should be executed, this also increases the internal step counter implicitly

template<typename **TCondition**, typename **TSequence**, typename **USequence**, int **IndexFirstProcess** = 0, int **IndexOfProcess1** = count_processes<*TSequence*, *IndexFirstProcess*>::count, int **IndexOfProcess2** = count_processes<*USequence*, *IndexOfProcess1*>::count>
class **SwitchProcessSequence** : public corsika::*BaseProcess*<*SwitchProcessSequence*<*TCondition*, *TSequence*, *USequence*>>

#include <SwitchProcessSequence.hpp> Class to switch between two process branches.

A compile-time static list of processes that uses an internal TCondition class to switch between different versions of processes (or process sequence).

TSequence and USequence must be derived from *BaseProcess* and are both references if possible (lvalue), otherwise (rvalue) they are just classes. This allows us to handle both, rvalue as well as lvalue Processes in the *SwitchProcessSequence*. Please use the `corsika::make_select(condition, sequence, alt_sequence)` factory function for best results.

TCondition has to implement a `bool operator()(Particle const&)`. Note: TCondition may absolutely also use random numbers to sample between its results. This can be used to achieve arbitrarily smooth transition or mixtures of processes.

Warning: do not put *StackProcess* into a *SwitchProcessSequence* since this makes no sense. The *StackProcess* acts on an entire particle stack and not on indiviidual particles.

Template parameters:

See also class *ProcessSequence*.

Template Parameters

- **TCondition** – selector functor/function
- **TSequence** – is of type *BaseProcess*, either a dedicatd process, or a *ProcessSequence*
- **USequence** – is of type *BaseProcess*, either a dedicatd process, or a *ProcessSequence*
- **IndexFirstProcess** – to count and index each Process in the entire process-chain
- **IndexOfProcess1** – index of TSequence (counting of Process)
- **IndexOfProcess2** – index of USequence (counting of Process)

Public Functions

inline **SwitchProcessSequence**(*TCondition* sel, *TSequence* in_A, *USequence* in_B)

Only valid user constructor will create fully initialized object.

SwitchProcessSequence supports and encourages move semantics. You can use object, l-value references or r-value references to construct sequences.

Parameters

- **sel** – functor to switch between branch A and B
- **in_A** – process branch A
- **in_B** – process branch B

Public Static Functions

static inline size_t constexpr **getNumberOfProcesses**()

static counter to uniquely index (count) all *ContinuousProcess* in switch sequence.

template<typename **TProcess1**, typename **TProcess2** = *NullModel*, int **ProcessIndexOffset** = 0, int **IndexOfProcess1** = corsika::count_processes<*TProcess1*, corsika::count_processes<*TProcess2*, *ProcessIndexOffset*>::count>::count, int **IndexOfProcess2** = corsika::count_processes<*TProcess2*, *ProcessIndexOffset*>::count>
class **ProcessSequence** : public corsika::*BaseProcess*<*ProcessSequence*<*TProcess1*, *TProcess2*>>

#include <ProcessSequence.hpp> Definition of a static process list/sequence.

A compile time static list of processes. The compiler will generate a new type based on template logic containing all the elements provided by the user.

TProcess1 and TProcess2 must both be derived from *BaseProcess*, and are both references if possible (lvalue), otherwise (rvalue) they are just classes. This allows us to handle both, rvalue as well as lvalue Processes in the *ProcessSequence*.

(For your potential interest, the static version of the *ProcessSequence* and all Process types are based on the CRTP C++ design pattern).

Template parameters:

Template Parameters

- **TProcess1** – is of type *BaseProcess*, either a dedicatd process, or a *ProcessSequence*
- **TProcess2** – is of type *BaseProcess*, either a dedicatd process, or a *ProcessSequence*
- **IndexFirstProcess** – to count and index each Process in the entire process-chain. The offset is the starting value for this *ProcessSequence*
- **IndexOfProcess1** – index of TProcess1 (counting of Process)
- **IndexOfProcess2** – index of TProcess2 (counting of Process)

Public Functions

`ProcessSequence`(*TProcess1* in_A, *TProcess2* in_B)

Only valid user constructor will create fully initialized object.

ProcessSequence supports and encourages move semantics. You can use object, l-value references or r-value references to construct sequences.

Parameters
- **`in_A`** – *BaseProcess* or switch/process list
- **`in_B`** – *BaseProcess* or switch/process list

template<typename **`TParticle`**>
ProcessReturn **`doBoundaryCrossing`**(*TParticle* &particle, typename *TParticle*::node_type const &from, typename *TParticle*::node_type const &to)

List of all BoundaryProcess.

Template Parameters
`TParticle` –

Parameters
- **`particle`** – The particle.
- **`from`** – Volume the particle is exiting.
- **`to`** – Volume the particle is entering.

Returns
ProcessReturn

template<typename **`TSecondaries`**>
void **`doSecondaries`**(*TSecondaries* &vS)

Process all secondaries in TSecondaries.

The seondaries produced by other processes and accessible via TSecondaries are processed by all SecondariesProcesse via a call here.

Template Parameters
`TSecondaries` –

Parameters
`vS` –

bool **`checkStep`**()

The processes of type *StackProcess* do have an internal counter, so they can be exectuted only each N steps.

Often these are "maintenacne processes" that do not need to run after each single step of the simulations. In the CheckStep function it is tested if either A_ or B_ are *StackProcess* and if they are due for execution.

template<typename **`TStack`**>
void **`doStack`**(*TStack* &stack)

Execute the StackProcess-es in the *ProcessSequence*.

template<typename **`TStack`**>
void **`doCascadeEquations`**(*TStack* &stack)

Execute the CascadeEquationsProcess-es in the *ProcessSequence*.

void **`initCascadeEquations`**()

Init the CascadeEquationsProcess-es in the *ProcessSequence*.

template<typename **`TParticle`**, typename **`TTrack`**>

ContinuousProcessStepLength **getMaxStepLength**(*TParticle* &&particle, *TTrack* &&vTrack)

Calculate the maximum allowed length of the next tracking step, based on all ContinuousProcess-es.

The maximum allowed step length is the minimum of the allowed track lenght over all ContinuousProcess-es in the *ProcessSequence*.

Template Parameters

- **TParticle** – particle type.
- **TTrack** – the trajectory type.

Parameters

- **particle** – The particle data object.
- **track** – The track data object.

Returns

ContinuousProcessStepLength which contains the step length itself in LengthType, and a unique identifier of the related *ContinuousProcess*.

template<typename **TParticle**>
CrossSectionType **getCrossSection**(*TParticle* const &projectile, Code const targetId, FourMomentum const &targetP4) const

Calculates the cross section of a projectile with a target.

Template Parameters

TParticle –

Parameters

- **projectile** –
- **targetId** –
- **targetP4** –

Returns

CrossSectionType

template<typename **TSecondaryView**, typename **TRNG**>
inline ProcessReturn **selectInteraction**(*TSecondaryView* &&view, FourMomentum const &projectileP4, NuclearComposition const &composition, *TRNG* &&rng, CrossSectionType const cx_select, CrossSectionType cx_sum = CrossSectionType::zero())

Selects one concrete *InteractionProcess* and samples a target nucleus from the material.

The selectInteraction method statically loops over all active *InteractionProcess* and calculates the material-weighted cross section for all of them. In an iterative way those cross sections are summed up. The random number cx_select, uniformely drawn from the cross section before energy losses, is used to discriminate the selected sub-process here. If the cross section after the step smaller than it was before, there is a non-zero probability that the particle survives and no interaction takes place. This method becomes imprecise when cross section rise with falling energies.

If a sub-process was selected, the target nucleus is selected from the material (weighted with cross section). The interaction is then executed.

Template Parameters

- **TSecondaryView** – Object type as storage for new secondary particles.
- **TRNG** – Object type to produce random numbers.

Parameters

- **view** – Object to store new secondary particles.
- **projectileP4** – The four momentum of the projectile.
- **composition** – The environment/material composition.
- **rng** – Random number object.
- **cx_select** – Drawn random numer, uniform between [0, cx_initial]
- **cx_sum** – For interal use, to sum up cross section contributions.

Returns

ProcessReturn

Public Static Functions

static inline size_t constexpr **`getNumberOfProcesses`**`()`

static counter to uniquely index (count) all *ContinuousProcess* in switch sequence.

CHAPTER

ELEVEN

HOWTO CREATE NEW PHYSICS MODULES

There are different types of physics modules, which you can add to the CORSIKA 8 physics process sequence. Modules can act on particles, secondaries or the entire stack. Modules can create new particles or they can modify or delete particles. They can also produce output.

Types of different modules are explained in ::modules

When creating new modules, we suggest to stick as close as possible to the default CORSIKA 8 coding guidelines and code structure. This makes code review and sharing with others not more complicated than needed.

When your modules creates output, use the available CORSIKA 8 output machinery. This is not explained here. Also learn how to use units, and use the loggers from the very beginning. Furthermore, get aquinted with C++17.

Let's consider the case of an "InteractionProcess" which will remove the projectile particle and create secondary particles on the stack instead. It also has a cross section in order to evaulate the probability with respect to other InteractionProcesses. Create a header file *SimpleProcess.hpp*, which is conceptually based to resemble (roughly) a Matthew-Heitler model:

```
#pragma once

#include <corsika/framework/core/ParticleProperties.hpp>
#include <corsika/framework/core/PhysicalConstants.hpp>
#include <corsika/framework/core/PhysicalUnits.hpp>
#include <corsika/framework/core/Logging.hpp>
#include <corsika/framework/process/InteractionProcess.hpp>
#include <corsika/framework/random/RNGManager.hpp>

namespace corsika::simple_process {

class SimpleProcess : public corsika::InteractionProcess<SimpleProcess> {

public:
   SimpleProcess();

   template <typename TParticle>
   GrammageType getInteractionLength(TParticle const&) const;

   template <typename TSecondaryView>
   void doInteraction(TSecondaryView& view) const;

private:
   // the random number stream
   corsika::default_prng_type& rng_ =
```

(continues on next page)

(continued from previous page)

```
      corsika::RNGManager<>::getInstance().getRandomStream("simple_process");

   // the logger
   std::shared_ptr<spdlog::logger> logger_ = get_logger("corsika_SimpleProcess");

};

}

#include <SimpleProcess.inl>
```

And the corresponding *SimpleProcess.inl* as:

```
Namespace corsika::simple_process
{

inline SimpleProcess::SimpleProcess() {}

template <typename TParticle>
inline GrammageType SimpleProcess::getInteractionLength(
      TParticle const &particle) const
{
   // this process only applies to hadrons above Ekin=100GeV
  if (is_hadron(particle.getPID()) && particle.getKineticEnergy()>100_GeV) {
     return 2_g/square(1_cm);
   }
   // otherwise its cross section is 0
     return std::numeric_limits<double>::infinity() * 1_g/square(1_cm);
}

template <typename TSecondaryView>
inline void SimpleProcess::doInteraction(TSecondaryView &view) const
{
   // the actual projectile particle, which will disappear after the call to␣
↪doInteraction finished.
   auto projectile = view.getProjectile();
   int const nMult = 15;
   auto pionEnergy = projectile.getEnergy() / nMult;

   auto const pOrig = projectile.getPosition();
   auto const tOrig = projectile.getTime();
   auto const projectileMomentum = projectile.getMomentum();
   auto const &cs = projectileMomentum.getCoordinateSystem();

   for (int iMult=0; iMult<nMult; ++iMult) {

      projectile.addSecondary(std::make_tuple(Code::PiPlus,
         projectileMomentum.normalized() * sqrt(square(pionEnergy) +␣
↪square(PiPlus::mass)),
         pOrig,
         tOrig));
   }
```

(continues on next page)

(continued from previous page)

```
  CORSIKA_LOGGER_INFO(logger_, "Created {} new secondaries of energy {}.", nMult,␣
↪pionEnergy);
}

}
```

In this example, *TParticle* as used here in the SimpleModule is one particle on the CORSIKA8 particle stack. It has all methods you expect as *getPosition()*, *getEnergy()*, *getKineticEnergy()*, etc.

The *TSecondaryView* provides special access to the CORSIKA8 particle stack as needed for a particle interaction with a projectile and secondaries. For example, *getProjectil()* will return the actual projectile particle, *addSecondary(…)* will produce a secondary of the projectil. The advantage of addSecondary wrt. addParticle is that there exists a direct reference to the projectile allowing to automatically copy weights, keep the cascade history, etc. Furthermore, you can define subsequent *SeoncariesProcesses* which can further process all the newly created secondaries. This could perform energy threshold cuts, or also calculate new weights, etc.

The SimpleProcess is not necessarily useful for anything and its sole purpuse is to illustrate the mechanism to create your own processes.

You can then include such a process in your programm (e.g. in vertical_EAS) with

- add: *#include <SimpleProcess.hpp>*
- register the new random stream: *RNGManager<>::getInstance().registerRandomStream("simple_process");*
- initialize it with `simple_process::SimpleProcess simple;`
- add it to the physiscs sequence i.e. via *auto extended_sequence = make_sequence(old_sequence, simple);*

Please follow the style and guidelines for programming CORSIKA 8 code even for private project. Your code will be GPLv3, too, and thus should be made public to the community. Any discussion or eventual bug-fixing is much more complicated if there are deviations from the guideline.

CHAPTER

TWELVE

REFERENCE DOCUMENTATION

Find the latest full reference manual at Doxygen, or the components on:

- Classes
- Functions
- Files

INDEX

C

convert_from_PDG (*C++ function*), 13
corsika::andre (*C++ type*), 26
corsika::andre::solve_quartic_real (*C++ function*), 26
corsika::andre::solveP3 (*C++ function*), 26
corsika::BaseProcess (*C++ struct*), 28
corsika::BaseProcess::getNumberOfProcesses (*C++ function*), 28
corsika::BaseProcess::process_type (*C++ type*), 28
corsika::BoundaryCrossingProcess (*C++ class*), 28
corsika::CascadeEquationsProcess (*C++ class*), 28
corsika::COMBoost (*C++ class*), 25
corsika::COMBoost::COMBoost (*C++ function*), 25
corsika::COMBoost::fromCoM (*C++ function*), 25
corsika::COMBoost::getOriginalCS (*C++ function*), 25
corsika::COMBoost::getRotatedCS (*C++ function*), 25
corsika::COMBoost::toCoM (*C++ function*), 25
corsika::ContinuousProcess (*C++ class*), 29
corsika::ContinuousProcessIndex (*C++ class*), 29
corsika::ContinuousProcessStepLength (*C++ class*), 29
corsika::DecayProcess (*C++ struct*), 29
corsika::full_name (*C++ struct*), 14
corsika::InteractionCounter (*C++ class*), 30
corsika::InteractionCounter::doInteraction (*C++ function*), 30
corsika::InteractionCounter::getCrossSection (*C++ function*), 30
corsika::InteractionCounter::getHistogram (*C++ function*), 30
corsika::InteractionHistogram (*C++ class*), 30
corsika::InteractionHistogram::CMSHist (*C++ function*), 31
corsika::InteractionHistogram::fill (*C++ function*), 31
corsika::InteractionHistogram::labHist (*C++ function*), 31
corsika::InteractionLengthModifier (*C++ class*), 31
corsika::InteractionLengthModifier::doInteraction (*C++ function*), 31
corsika::InteractionLengthModifier::functor_signature (*C++ type*), 31
corsika::InteractionLengthModifier::getInteractionLength (*C++ function*), 31
corsika::InteractionLengthModifier::getProcess (*C++ function*), 31
corsika::InteractionLengthModifier::InteractionLengthModif
(*C++ function*), 31
corsika::InteractionProcess (*C++ class*), 31
corsika::MediumData (*C++ struct*), 16
corsika::MediumData::getAA (*C++ function*), 17
corsika::MediumData::getCbar (*C++ function*), 17
corsika::MediumData::getCorrectedDensity (*C++ function*), 17
corsika::MediumData::getDlt0 (*C++ function*), 18
corsika::MediumData::getIeff (*C++ function*), 17
corsika::MediumData::getPrettyName (*C++ function*), 17
corsika::MediumData::getSK (*C++ function*), 17
corsika::MediumData::getStateOfMatter (*C++ function*), 17
corsika::MediumData::getSternheimerDensity (*C++ function*), 17
corsika::MediumData::getSymbol (*C++ function*), 17
corsika::MediumData::getType (*C++ function*), 17
corsika::MediumData::getWeight (*C++ function*), 17
corsika::MediumData::getX0 (*C++ function*), 17
corsika::MediumData::getX1 (*C++ function*), 17
corsika::MediumData::weight_error_last_digit (*C++ function*), 17
corsika::MediumData::weight_significant_figure (*C++ function*), 17
corsika::MediumData::Z_over_A (*C++ function*), 17
corsika::NullModel (*C++ class*), 32
corsika::NullModel::getNumberOfProcesses

(*C++ function*), 32
corsika::ProcessSequence (*C++ class*), 34
corsika::ProcessSequence::checkStep (*C++ function*), 35
corsika::ProcessSequence::doBoundaryCrossing (*C++ function*), 35
corsika::ProcessSequence::doCascadeEquations (*C++ function*), 35
corsika::ProcessSequence::doSecondaries (*C++ function*), 35
corsika::ProcessSequence::doStack (*C++ function*), 35
corsika::ProcessSequence::getCrossSection (*C++ function*), 36
corsika::ProcessSequence::getMaxStepLength (*C++ function*), 35
corsika::ProcessSequence::getNumberOfProcesses (*C++ function*), 37
corsika::ProcessSequence::initCascadeEquations (*C++ function*), 35
corsika::ProcessSequence::ProcessSequence (*C++ function*), 35
corsika::ProcessSequence::selectInteraction (*C++ function*), 36
corsika::SecondariesProcess (*C++ class*), 32
corsika::StackProcess (*C++ class*), 32
corsika::StackProcess::checkStep (*C++ function*), 33
corsika::StackProcess::getStep (*C++ function*), 33
corsika::SwitchProcessSequence (*C++ class*), 33
corsika::SwitchProcessSequence::getNumberOfProcesses (*C++ function*), 34
corsika::SwitchProcessSequence::SwitchProcessSequence (*C++ function*), 34

G

get_all_particles (*C++ function*), 13
get_charge (*C++ function*), 12
get_charge_number (*C++ function*), 12
get_energy_production_threshold (*C++ function*), 12
get_kinetic_energy_propagation_threshold (*C++ function*), 12
get_lifetime (*C++ function*), 12
get_mass (*C++ function*), 12
get_name (*C++ function*), 12
get_nucleus_A (*C++ function*), 13
get_nucleus_code (*C++ function*), 13
get_nucleus_mass (*C++ function*), 13
get_nucleus_name (*C++ function*), 13
get_nucleus_Z (*C++ function*), 13
get_PDG (*C++ function*), 12

I

is_charged (*C++ function*), 13
is_em (*C++ function*), 12
is_hadron (*C++ function*), 12
is_muon (*C++ function*), 12
is_neutrino (*C++ function*), 13
is_nucleus (*C++ function*), 13

M

MediumType (*C++ enum*), 16
MediumType::BiologicalDosimetry (*C++ enumerator*), 16
MediumType::Element (*C++ enumerator*), 16
MediumType::InorganicCompound (*C++ enumerator*), 16
MediumType::Mixture (*C++ enumerator*), 16
MediumType::OrganicCompound (*C++ enumerator*), 16
MediumType::Polymer (*C++ enumerator*), 16
MediumType::RadioactiveElement (*C++ enumerator*), 16
MediumType::Unknown (*C++ enumerator*), 16

O

operator<< (*C++ function*), 14

S

set_energy_production_threshold (*C++ function*), 12
set_kinetic_energy_propagation_threshold (*C++ function*), 12
StateOfMatter (*C++ enum*), 16
StateOfMatter::DiatomicGas (*C++ enumerator*), 16
StateOfMatter::Gas (*C++ enumerator*), 16
StateOfMatter::Liquid (*C++ enumerator*), 16
StateOfMatter::Solid (*C++ enumerator*), 16
StateOfMatter::Unknown (*C++ enumerator*), 16