Matrix Cascade Equation (MCEq) Documentation

Release 1.0rc1

Anatoli Fedynitch

Aug 09, 2017

Contents

Contents:

CHAPTER 1

Introduction

The program Matrix Cascade Equation is a numerical solver of the discrete form of the cascade equation

$$\frac{\mathrm{d}}{\mathrm{d}X}\mathbf{\Phi} = \left[(-\mathbf{1} + \boldsymbol{C})\mathbf{\Lambda}_{int} + \frac{1}{\rho(X)}(-\mathbf{1} + \boldsymbol{D})\mathbf{\Lambda}_{dec}\right]\mathbf{\Phi}.$$

...

CHAPTER 2

Citations (Updated)

The first version of the code is feature complete and enters a final stage, where release candidates are circulated. We had several conference papers using `MCEq`. If you use it in your scientific publication, please cite the code **AND** the physical model publications properly.

Before the main paper is published please continue citing the proceedings below, which describes a bit of functionality, possibilities and technical details:

> *Calculation of conventional and prompt lepton fluxes at very high energy*
> A. Fedynitch, R. Engel, T. K. Gaisser, F. Riehn, T. Stanev,
> EPJ Web Conf. 99 (2015) 08001
> arXiv:1503.00544

You might want to consider citing the paper which for us was the basis for this type of calculations

> *Influence of hadronic interaction models and the cosmic ray spectrum on the high energy atmospheric muon and neutrino flux*
> A. Fedynitch, J. B. Tjus, P. Desiati
> Phys.Rev. D86 (2012) 114024
> arXiv:1206.6710

Cosmic-ray flux parametrizations

The parametrizations of the flux of cosmic rays are provided via the module CRFluxModels. The references are given in the module's documentation.

Atmosphere

- **CORSIKA parametrizations**

  CORSIKA: A Monte Carlo Code to Simulate Extensive Air Showers

D. Heck, J. Knapp, J. Capdevielle, G. Schatz, T. Thouw
Tech. Rep. FZKA 6019, Karsruhe (1998), link

- **NRLMSISE-00**

  NRLMSISE-00 empirical model of the atmosphere: Statistical comparisons and scientific issues
  J.M. Picone, A.E. Hedin, D.P. Drob, and A.C. Aikin
  J. Geophys. Res., 107(A12), 1468, doi:10.1029/2002JA009430, (2002)

Hadronic interaction models

- **SIBYLL-2.3c:**

  The hadronic interaction model Sibyll 2.3c and Feynman scaling
  F. Riehn, R. Engel, A. Fedynitch, T. K. Gaisser, T. Stanev
  PoS(ICRC2017)301
  arXiv link soon

- **SIBYLL-2.3:**

  Charm production in SIBYLL
  F. Riehn, R. Engel, A. Fedynitch, T. K. Gaisser, T. Stanev
  EPJ Web Conf. 99 (2015) 12001
  arXiv:1502.06353

- **SIBYLL-2.1:**

  Cosmic ray interaction event generator SIBYLL 2.1
  E.-J. Ahn, R. Engel, T. K. Gaisser, P. Lipari, T.Stanev
  Phys.Rev. D80 (2009) 094003, arXiv:0906.4113

- **EPOS-LHC:**

  EPOS LHC : test of collective hadronization with LHC data
  T. Pierog, I. Karpenko, J. M. Katzy, E. Yatsenko, and K. Werner
  Phys. Rev. C92 (2015) 034906
  arXiv:1306.0121

- **QGSJET-II-04:**

  Monte Carlo treatment of hadronic interactions in enhanced Pomeron scheme: I. QGSJET-II model
  Sergey Ostapchenko
  Phys.Rev. D83 (2011) 014018,
  arXiv:1010.1869

- **DPMJET-III:**

  The Monte Carlo event generator DPMJET-III
  S. Roesler, R. Engel, J. Ranft
  Advanced Monte Carlo for radiation physics, particle transport simulation and applications. Proceedings, Conference, MC2000, Lisbon, Portugal, October 23-26, 2000
  arXiv:hep-ph/0012252

- **DPMJET-III-17.1:**

  Revision of the high energy hadronic interaction models PHOJET/DPMJET-III
  A. Fedynitch, R. Engel

14th International Conference on Nuclear Reaction Mechanisms, Villa Monastero, Varenna, Italy, 15 - 19 Jun 2015, pp.291
on CERN Document Server

Models of charm production

For prompt fluxes from SIBYLL-2.3, refer to the corresponding proceedings above.

- **MRS - Martin-Ryskin-Stasto**

  Prompt neutrinos from atmospheric $c\bar{c}$ and $b\bar{b}$ production and the gluon at very small x
  A. D. Martin, M. G. Ryskin, A. M. Stasto
  Acta Physica Polonica B 34, 3273 (2003)

CHAPTER 3

Module documetation

Physical models

The cascade equation can be applied whenever particles interact and decay. The main purpose of the current program version is to calculate inclusive fluxes of leptons (μ, ν_μ, ν_e and ν_τ) in the Earth's atmosphere. The calculation requires a spherical model of the Earth's atmosphere which on hand is based on its *MCEq.geometry* and on the other hand parameterizations or numerical models of the *MCEq.density_profiles*.

Alternatiely, this code could be used for calculations of the high energy hadron and lepton flux in astrophysical environments, where the gas density and the (re-)interaction probability are very low. Prediction of detailed photon spectra is possible but additional extensions.

At very high energies, i.e. beyond 100 TeV, the decays of very short-lived particles become an important contribution to the flux. However, heavy-flavor production at these energies is not well known and it can not be consitently predicted within theoretical frameworks. Some of the default interaction models (`SIBYLL-2.3` or `SIBYLL-2.3c`) contain models of charmed particle production but they represent only hypotheses based on experimental data and phenomenology. Other *MCEq.charm_models* exist, such as the *MCEq.charm_models.MRS_charm*, which can be coupled with any other interaction model for normal hadron production. In practice any kind of model which predicts a x distribution can be employed in this code as extension of the *MCEq.charm_models* module.

`MCEq.density_profiles` - models of the Earth's atmosphere

This module includes classes and functions modeling the Earth's atmosphere. Currently, two different types models are supported:

- Linsley-type/CORSIKA-style parameterization
- Numerical atmosphere via external routine (NRLMSISE-00)

Both implementations have to inherit from the abstract class *EarthAtmosphere*, which provides the functions for other parts of the program. In particular the function *EarthAtmosphere.get_density()*

Typical interaction:

```
$ atm_object = CorsikaAtmosphere("BK_USStd")
$ atm_object.set_theta(90)
$ print 'density at X=100', atm_object.X2rho(100.)
```

The class `MCEqRun` will only the following routines::

- *EarthAtmosphere.set_theta()*,
- *EarthAtmosphere.r_X2rho()*.

If you are extending this module make sure to provide these functions without breaking compatibility.

Example

An example can be run by executing the module:

```
$ python MCEq/atmospheres.py
```

class MCEq.density_profiles.**AIRSAtmosphere**(*location*, *season*, *extrapolate=True*, **args*, ***kwargs*)

Interpolation class for tabulated atmospheres.

This class is intended to read preprocessed AIRS Satellite data.

Parameters

- **location** (*str*) – see *init_parameters()*
- **season** (*str, optional*) – see *init_parameters()*

get_density(*h_cm*)

Returns the density of air in g/cm**3.

Wraps around ctypes calls to the NRLMSISE-00 C library.

Parameters **h_cm** (*float*) – height in cm

Returns density $\rho(h_{cm})$ in g/cm**3

Return type float

init_parameters(*location*, ***kwargs*)

Loads tables and prepares interpolation.

Parameters

- **location** (*str*) – supported is only "SouthPole"
- **doy** (*int*) – Day Of Year

class MCEq.density_profiles.**CorsikaAtmosphere**(*location*, *season=None*)

Class, holding the parameters of a Linsley type parameterization similar to the Air-Shower Monte Carlo CORSIKA.

The parameters pre-defined parameters are taken from the CORSIKA manual. If new sets of parameters are added to *init_parameters()*, the array _thickl can be calculated using *calc_thickl()* .

_atm_param

numpy.array – (5x5) Stores 5 atmospheric parameters _aatm, _batm, _catm, _thickl, _hlay for each of the 5 layers

Parameters

- **location** (*str*) – see *init_parameters()*
- **season** (*str, optional*) – see *init_parameters()*

calc_thickl()
Calculates thickness layers for *depth2height()*

The analytical inversion of the CORSIKA parameterization relies on the knowledge about the depth X, where trasitions between layers/exponentials occur.

Example

Create a new set of parameters in *init_parameters()* inserting arbitrary values in the _thikl array:

```
$ cor_atm = CorsikaAtmosphere(new_location, new_season)
$ cor_atm.calc_thickl()
```

Replace _thickl values with printout.

depth2height(*x_v*)
Converts column/vertical depth to height.

Parameters **x_v** (*float*) – column depth X_v in g/cm**2

Returns height in cm

Return type float

get_density(*h_cm*)
Returns the density of air in g/cm**3.

Uses the optimized module function `corsika_get_density_jit()`.

Parameters **h_cm** (*float*) – height in cm

Returns density $\rho(h_{cm})$ in g/cm**3

Return type float

get_mass_overburden(*h_cm*)
Returns the mass overburden in atmosphere in g/cm**2.

Uses the optimized module function `corsika_get_m_overburden_jit()`.

Parameters **h_cm** (*float*) – height in cm

Returns column depth $T(h_{cm})$ in g/cm**2

Return type float

init_parameters(*location*, *season*)
Initializes *_atm_param*.

location	CORSIKA Table	Description/season
"USStd"	1	US Standard atmosphere
"BK_USStd"	31	Bianca Keilhauer's USStd
"Karlsruhe"	18	AT115 / Karlsruhe
"SouthPole"	26 and 28	MSIS-90-E for Dec and June
"PL_SouthPole"	29 and 30	16.Lipari's Jan and Aug

Parameters

- **location** (*str*) – see table
- **season** (*str, optional*) – choice of season for supported locations

Raises Exception – if parameter set not available

rho_inv(*X*, *cos_theta*)
Returns reciprocal density in cm**3/g using planar approximation.

This function uses the optimized function `planar_rho_inv_jit()`

Parameters **h_cm** (*float*) – height in cm

Returns $\frac{1}{\rho}(X, \cos\theta)$ cm**3/g

Return type float

class MCEq.density_profiles.**EarthAtmosphere**(**args*, ***kwargs*)
Abstract class containing common methods on atmosphere. You have to inherit from this class and implement the virtual method *get_density()*.

Note: Do not instantiate this class directly.

---

thrad
float – current zenith angle θ in radiants

theta_deg
float – current zenith angle θ in degrees

max_X
float – Slant depth at the surface according to the geometry defined in the *MCEq.geometry*

X2rho(*X*)
Returns the density $\rho(X)$.

The spline *s_X2rho* is used, which was calculated or retrieved from cache during the *set_theta()* call.

Parameters **X** (*float*) – slant depth in g/cm**2

Returns ρ in cm**3/g

Return type float

calculate_density_spline(*n_steps=2000*)
Calculates and stores a spline of $\rho(X)$.

Parameters **n_steps** (*int, optional*) – number of X values to use for interpolation

Raises Exception – if *set_theta()* was not called before.

gamma_cherenkov_air(*h_cm*)
Returns the Lorentz factor gamma of Cherenkov threshold in air (MeV).

get_density(*h_cm*)
Abstract method which implementation should return the density in g/cm**3.

Parameters **h_cm** (*float*) – height in cm

Returns density in g/cm**3

Return type float

Raises NotImplementedError

h2X(*h*)
Returns the depth along path as function of height above surface.

The spline *s_X2rho* is used, which was calculated or retrieved from cache during the *set_theta()* call.

Parameters **h** (*float*) – vertical height above surface in cm

Returns X slant depth in g/cm**2

Return type float

moliere_air(*h_cm*)
Returns the Moliere unit of air for US standard atmosphere.

nref_rel_air(*h_cm*)
Returns the refractive index - 1 in air (density parametrization as in CORSIKA).

r_X2rho(*X*)
Returns the inverse density $\frac{1}{\rho}(X)$.

The spline *s_X2rho* is used, which was calculated or retrieved from cache during the *set_theta()* call.

Parameters **X** (*float*) – slant depth in g/cm**2

Returns $1/\rho$ in cm**3/g

Return type float

set_theta(*theta_deg*, *force_spline_calc=False*)
Configures geometry and initiates spline calculation for $\rho(X)$.

If the option 'use_atm_cache' is enabled in the config, the function will check, if a corresponding spline is available in the cache and use it. Otherwise it will call *calculate_density_spline()*, make the function *r_X2rho()* available to the core code and store the spline in the cache.

Parameters

- **theta_deg** (*float*) – zenith angle θ at detector
- **force_spline_calc** (*bool*) – forces (re-)calculation of the spline for each call

theta_cherenkov_air(*h_cm*)
Returns the Cherenkov angle in air (degrees).

class MCEq.density_profiles.**IsothermalAtmosphere**(*location*, *season*, *hiso_km=6.3*, *X0=1300.0*)
Isothermal model of the atmosphere.

This model is widely used in semi-analytical calculations. The isothermal approximation is valid in a certain range of altitudes and usually one adjust the parameters to match a more realistic density profile at altitudes between 10 - 30 km, where the high energy muon production rate peaks. Such parametrizations are given in the book "Cosmic Rays and Particle Physics", Gaisser, Engel and Resconi (2016). The default values are from M. Thunman, G. Ingelman, and P. Gondolo, Astropart. Physics 5, 309 (1996).

Parameters

- **location** (*str*) – no effect
- **season** (*str*) – no effect
- **hiso_km** (*float*) – isothermal scale height in km
- **X0** (*float*) – Ground level overburden

get_density(*h_cm*)
Returns the density of air in g/cm**3.

Parameters **h_cm** (*float*) – height in cm

Returns density $\rho(h_{cm})$ in g/cm**3

Return type float

get_mass_overburden(*h_cm*)
Returns the mass overburden in atmosphere in g/cm**2.

Parameters **h_cm** (*float*) – height in cm

Returns column depth $T(h_{cm})$ in g/cm**2

Return type float

class MCEq.density_profiles.**MSIS00Atmosphere**(*location*, *season*)
Wrapper class for a python interface to the NRLMSISE-00 model.

NRLMSISE-00 is an empirical model of the Earth's atmosphere. It is available as a FORTRAN 77 code or as a verson traslated into C by Dominik Borodowski. Here a PYTHON wrapper has been used.

_msis
NRLMSISE-00 python wrapper object handler

Parameters

- **location** (*str*) – see *init_parameters()*
- **season** (*str, optional*) – see *init_parameters()*

get_density(*h_cm*)
Returns the density of air in g/cm**3.

Wraps around ctypes calls to the NRLMSISE-00 C library.

Parameters **h_cm** (*float*) – height in cm

Returns density $\rho(h_{cm})$ in g/cm**3

Return type float

init_parameters(*location*, *season*)
Sets location and season in NRLMSISE-00.

Translates location and season into day of year and geo coordinates.

Parameters

- **location** (*str*) – Supported are "SouthPole" and "Karlsruhe"
- **season** (*str*) – months of the year: January, February, etc.

class MCEq.density_profiles.**MSIS00IceCubeCentered**(*location*, *season*)
Extension of *MSIS00Atmosphere* which couples the latitude setting with the zenith angle of the detector.

Parameters

- **location** (*str*) – see init_parameters()
- **season** (*str, optional*) – see init_parameters()

latitude(*det_zenith_deg*)
Returns the geographic latitude of the shower impact point.

Assumes a spherical earth. The detector is 1948m under the surface.

Credits: geometry fomulae by Jakob van Santen, DESY Zeuthen.

Parameters **det_zenith_deg** (*float*) – zenith angle at detector in degrees

Returns latitude of the impact point in degrees

Return type float

MCEq.geometry — Extensive-Air-Shower geometry

This module includes classes and functions modeling the geometry of the cascade trajectory.

class MCEq.geometry.**EarthGeometry**

A model of the Earth's geometry, approximating it by a sphere. The figure below illustrates the meaning of the parameters.

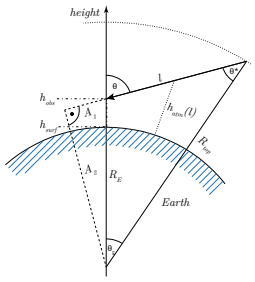
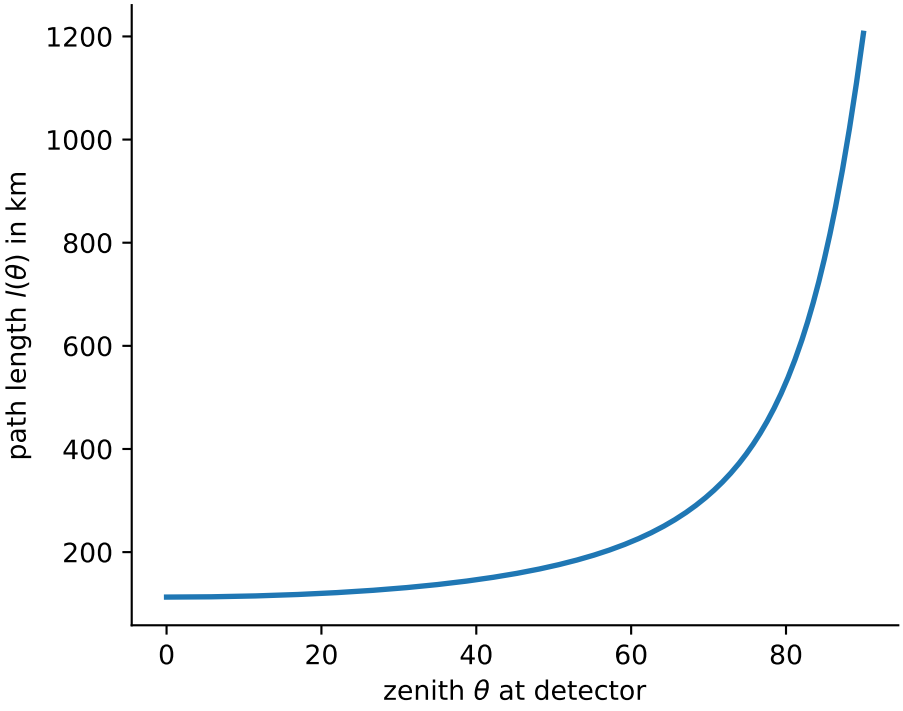


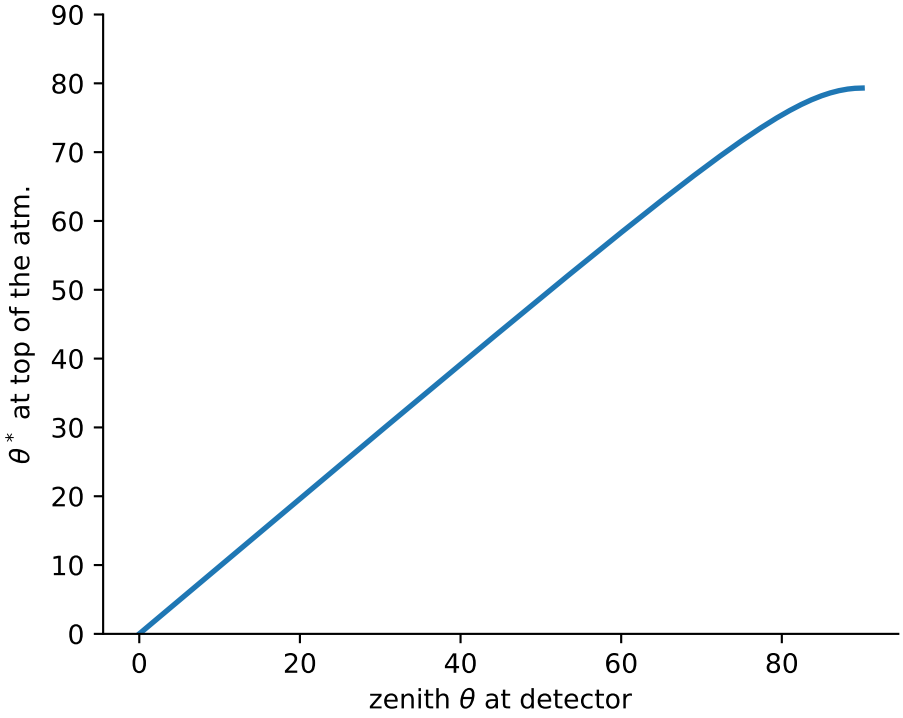
Fig. 3.1: Curved geometry as it is used in the code (not to scale!).

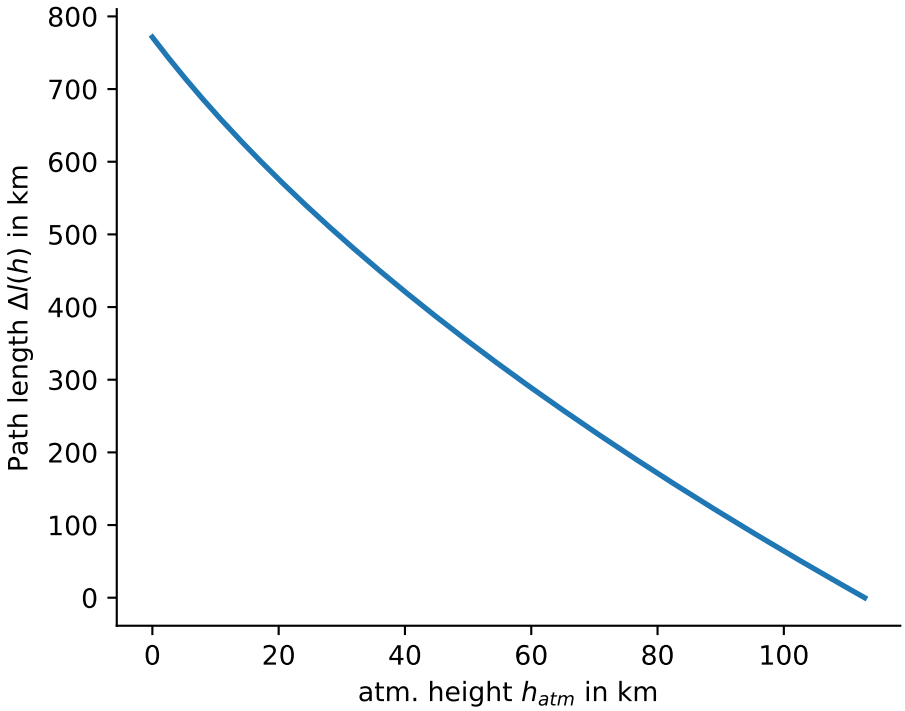
Example

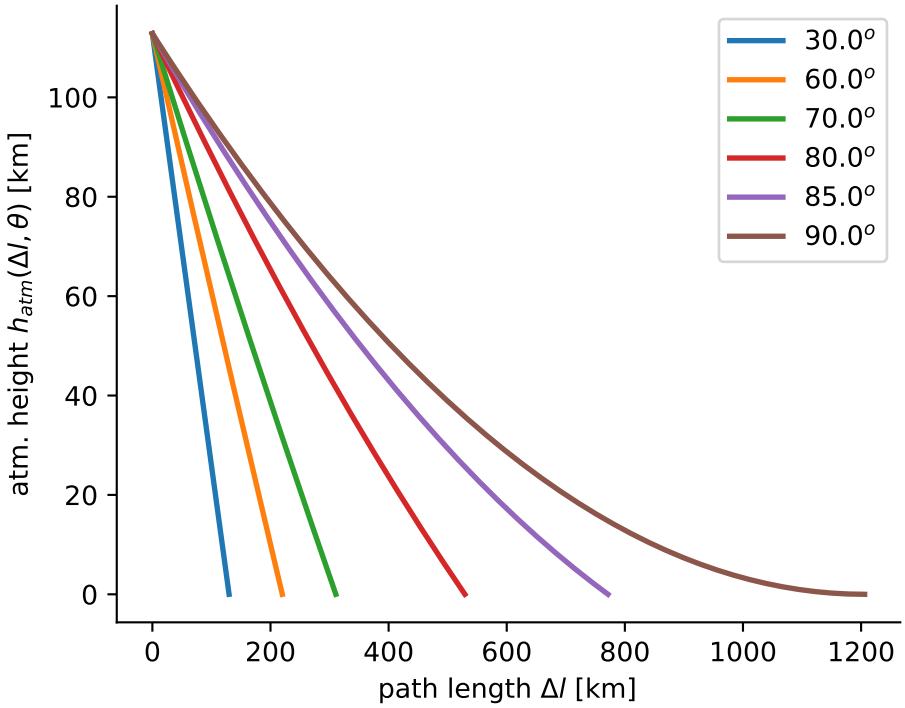
The plots below will be produced by executing the module:

```
$ python geometry.py
```


1200
1000
800
600
400
200
path length $l(\theta)$ in km
0
20
40
60
80
zenith θ at detector


90
80
70
60
50
40
30
20
10
0
θ^* at top of the atm.
0
20
40
60
80
zenith θ at detector





The module currently contains only module level functions. To implement a different geometry, e.g. in an astrophysical content, you can create a new module providing similar functions or place the current content into a class.

h_obs

float – observation level height [cm]

h_atm
float – top of the atmosphere [cm]

r_E
float – radius Earth [cm]

r_top
float – radius at top of the atmosphere [cm]

r_obs
float – radius at observation level [cm]

cos_th_star(*theta*)
Returns the zenith angle at atmospheric boarder $\cos(\theta^*)$ in [rad] as a function of zenith at detector.

delta_l(*h*, *theta*)
Distance dl covered along path $l(\theta)$ as a function of current height. Inverse to `h()`.

h(*dl*, *theta*)
Height above surface at distance dl counted from the beginning of path $l(\theta)$ in cm.

l(*theta*)
Returns path length in [cm] for given zenith angle θ [rad].

MCEq.geometry.**chirkin_cos_theta_star**(*costheta*)
$\cos(\theta^*)$ parameterization.

This function returns the equivalent zenith angle for for very inclined showers. It is based on a CORSIKA study by D. Chirkin, hep-ph/0407078v1, 2004.

Parameters **costheta** (*float*) – $\cos(\theta)$ in [rad]

Returns $\cos(\theta*)$ in [rad]

Return type float

MCEq.charm_models — charmed particle production

This module includes classes for custom charmed particle production. Currently only the MRS model is implemented as the class *MRS_charm*. The abstract class *CharmModel* guides the implementation of custom classes.

The `Yields` instantiates derived classes of *CharmModel* and calls *CharmModel.get_yield_matrix()* when overwriting a model yield file in `Yields.set_custom_charm_model()`.

class MCEq.charm_models.**CharmModel**
Abstract class, from which implemeted charm models can inherit.

Note: Do not instantiate this class directly.

get_yield_matrix(*proj*, *sec*)
The implementation of this abstract method returns the yield matrix spanned over the energy grid of the calculation.

Parameters

- **proj** (*int*) – PDG ID of the interacting particle (projectile)

- **sec** (*int*) – PDG ID of the final state charmed meson (secondary)

Returns yield matrix

Return type np.array

Raises NotImplementedError

class MCEq.charm_models.**MRS_charm**(*e_grid*, *csm*)

Martin-Ryskin-Stasto charm model.

The model is described in A. D. Martin, M. G. Ryskin, and A. M. Stasto, Acta Physica Polonica B 34, 3273 (2003). The parameterization of the inclusive $c\bar{c}$ cross-section is given in the appendix of the paper. This formula provides the behavior of the cross-section, while fragmentation functions and certain scales are needed to obtain meson and baryon fluxes as a function of the kinematic variable x_F. At high energies and $x_F > 0.05$, where this model is valid, $x_F \approx x = E_c/E_{proj}$. Here, these fragmentation functions are used:

- D-mesons $\frac{4}{3}x$
- Λ-baryons $\frac{1}{1.47}x$

The production ratios between the different types of D-mesons are stored in the attribute *cs_scales* and *D0_scale*, where *D0_scale* is the $c\bar{c}$ to D^0 ratio and *cs_scales* stores the production ratios of $D^{\pm}/D^0$, D_s/D^0 and Λ_c/D^0.

Since the model employs only perturbartive production of charm, the charge conjugates are symmetric, i.e. $\sigma_{D^+} = \sigma_{D^-}$ etc.

Parameters

- **e_grid** (*np.array*) – energy grid as it is defined in MCEqRun.
- **csm** (*np.array*) – inelastic cross-sections as used in MCEqRun.

D_dist(*x*, *E*, *mes*)

Returns the Feynman-x_F distribution of $\sigma_{D-mesons}$ in mb

Parameters

- **x** (*float or np.array*) – x_F
- **E** (*float*) – center-of-mass energy in GeV
- **mes** (*int*) – PDG ID of D-meson: $\pm 421, \pm 431, \pm 411$

Returns $\sigma_{D-mesons}$ in mb

Return type float

LambdaC_dist(*x*, *E*)

Returns the Feynman-x_F distribution of σ_{Λ_C} in mb

Parameters

- **x** (*float or np.array*) – x_F
- **E** (*float*) – center-of-mass energy in GeV
- **mes** (*int*) – PDG ID of D-meson: $\pm 421, \pm 431, \pm 411$

Returns $\sigma_{D-mesons}$ in mb

Return type float

dsig_dx(*x*, *E*)

Returns the Feynman-x_F distribution of $\sigma_{c\bar{c}}$ in mb

Parameters

- **x** (*float or np.array*) – x_F
- **E** (*float*) – center-of-mass energy in GeV

Returns $\sigma_{c\bar{c}}$ in mb

Return type float

get_yield_matrix(*proj*, *sec*)

Returns the yield matrix in proper format for `MCEqRun`.

Parameters

- **proj** (*int*) – projectile PDG ID ± [2212, 211, 321]
- **sec** (*int*) – charmed particle PDG ID ± [411, 421, 431, 4122]

Returns

yield matrix if (proj,sec) combination allowed, else zero matrix

Return type np.array

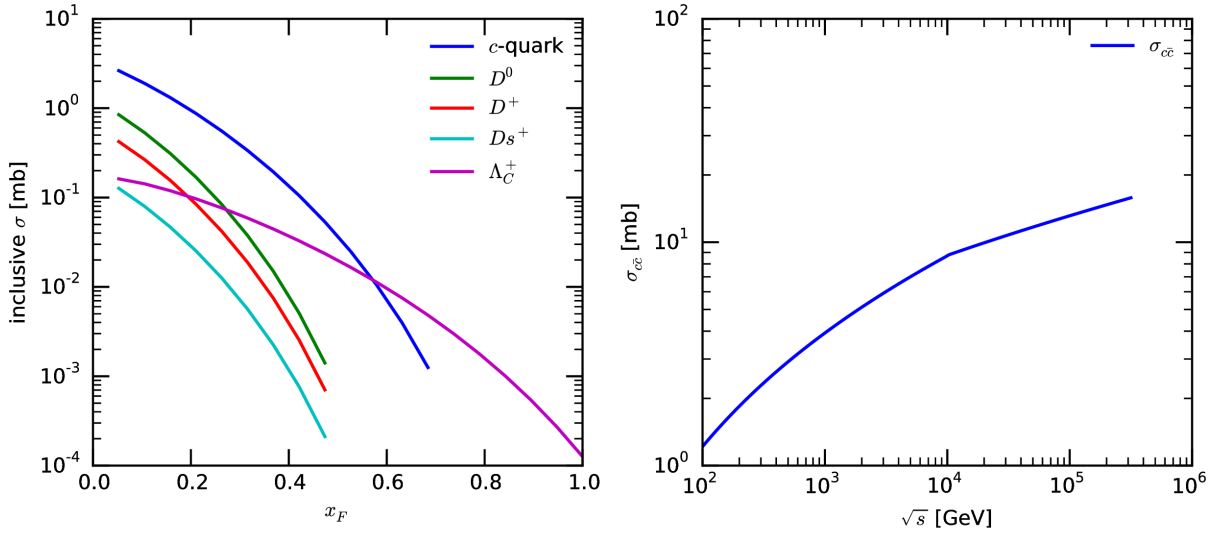
sigma_cc(*E*)

Returns the integrated ccbar cross-section in mb.

Note: Integration is not going over complete phase-space due to limitations of the parameterization.

test()

Plots the meson, baryon and charm quark distribution as shown in the plot below.



D0_scale = 0.47619047619047616

D0 cross-section wrt to the ccbar cross-section

allowed_proj = [2212, -2212, 2112, -2112, 211, -211, 321, -321]

hadron projectiles, which are allowed to produce charm

allowed_sec = [411, 421, 431, 4122]

charm secondaries, which are predicted by this model

cs_scales = {4122: 0.45, 411: 0.5, 421: 1.0, 431: 0.15}
fractions of cross-section wrt to D0 cross-section

class MCEq.charm_models.**WHR_charm**(*e_grid*, *csm*)
Logan Wille, Francis Halzen, Hall Reno.

The approach is the same as in A. D. Martin, M. G. Ryskin, and A. M. Stasto, Acta Physica Polonica B 34, 3273 (2003). The parameterization of the inclusive $c\bar{c}$ cross-section is replaced by interpolated tables from the calculation. Fragmentation functions and certain scales are needed to obtain meson and baryon fluxes as a function of the kinematic variable x_F. At high energies and $x_F > 0.05$, where this model is valid, $x_F \approx x = E_c/E_{proj}$. Here, these fragmentation functions are used:

- D-mesons $\frac{4}{3}x$
- Λ-baryons $\frac{1}{1.47}x$

The production ratios between the different types of D-mesons are stored in the attribute `cs_scales` and `D0_scale`, where `D0_scale` is the $c\bar{c}$ to D^0 ratio and `cs_scales` stores the production ratios of $D^{\pm}/D^0$, D_s/D^0 and Λ_c/D^0.

Since the model employs only perturbative production of charm, the charge conjugates are symmetric, i.e. $\sigma_{D^+} = \sigma_{D^-}$ etc.

Parameters

- **e_grid** (*np.array*) – energy grid as it is defined in MCEqRun.
- **csm** (*np.array*) – inelastic cross-sections as used in MCEqRun.

dsig_dx(*x*, *E*)
Returns the Feynman-x_F distribution of $\sigma_{c\bar{c}}$ in mb

Parameters

- **x** (*float or np.array*) – x_F
- **E** (*float*) – center-of-mass energy in GeV

Returns $\sigma_{c\bar{c}}$ in mb

Return type float

Core functionality

MCEq.core - core module

This module contains the main program features. Instantiating *MCEq.core.MCEqRun* will initialize the data structures and particle tables, create and fill the interaction and decay matrix and check if all information for the calculation of inclusive fluxes in the atmosphere is available.

The preferred way to instantiate *MCEq.core.MCEqRun* is:

```
from mceq_config import config
from MCEq.core import MCEqRun
import CRFluxModels as pm

mceq_run = MCEqRun(interaction_model='SIBYLL2.3c',
                   primary_model=(pm.HillasGaisser2012, "H3a"),
                   **config)
```

```
mceq_run.set_theta_deg(60.)
mceq_run.solve()
```

class MCEq.core.**MCEqRun**(*interaction_model*, *density_model*, *primary_model*, *theta_deg*, *adv_set*, *obs_ids*, **args*, ***kwargs*)

Main class for handling the calculation.

This class is the main user interface for the caclulation. It will handle initialization and various error/configuration checks. The setup has to be accomplished before invoking the integration routine is `MCeqRun.solve()`. Changes of configuration, such as:

- interaction model in *MCEqRun.set_interaction_model()*,
- primary flux in *MCEqRun.set_primary_model()*,
- zenith angle in *MCEqRun.set_theta_deg()*,
- density profile in *MCEqRun.set_density_model()*,
- member particles of the special `obs_` group in *MCEqRun.set_obs_particles()*,

can be made on an active instance of this class, while calling *MCEqRun.solve()* subsequently to calculate the solution corresponding to the settings.

The result can be retrieved by calling *MCEqRun.get_solution()*.

Parameters

- **interaction_model** (*string*) – PDG ID of the particle
- **density_model** (*string*,*sting*,*string*) – model type, location, season
- **primary_model** (*class, param_tuple*) – classes derived from `CRFluxModels.PrimaryFlux` and its parameters as tuple
- **theta_deg** (*float*) – zenith angle θ in degrees, measured positively from vertical direction
- **adv_set** (*dict*) – advanced settings, see `mceq_config`
- **obs_ids** (*list*) – list of particle name strings. Those lepton decay products will be scored in the special `obs_` categories

get_solution(*particle_name*, *mag=0.0*, *grid_idx=None*, *integrate=False*)

Retrieves solution of the calculation on the energy grid.

Some special prefixes are accepted for lepton names:

- the total flux of muons, muon neutrinos etc. from all sources/mothers can be retrieved by the prefix `total_`, i.e. `total_numu`
- the conventional flux of muons, muon neutrinos etc. from all sources can be retrieved by the prefix `conv_`, i.e. `conv_numu`
- correspondigly, the flux of leptons which originated from the decay of a charged pion carries the prefix `pi_` and from a kaon `k_`
- conventional leptons originating neither from pion nor from kaon decay are collected in a category without any prefix, e.g. `numu` or `mu+`

Parameters

- **particle_name** (*str*) – The name of the particle such, e.g. `total_mu+` for the total flux spectrum of positive muons or `pr_antinumu` for the flux spectrum of prompt anti muon neutrinos

- **mag** (*float, optional*) – 'magnification factor': the solution is multiplied by `sol` $= \Phi \cdot E^{mag}$
- **grid_idx** (*int, optional*) – if the integrator has been configured to save intermediate solutions on a depth grid, then `grid_idx` specifies the index of the depth grid for which the solution is retrieved. If not specified the flux at the surface is returned
- **integrate** (*bool, optional*) – return averge particle number instead of
- **flux** (*multiply by bin width*) –

Returns flux of particles on energy grid *e_grid*

Return type (numpy.array)

set_density_model(*density_config*)

Sets model of the atmosphere.

To choose, for example, a CORSIKA parametrization for the Southpole in January, do the following:

```
mceq_instance.set_density_model(('CORSIKA', 'PL_SouthPole', 'January'))
```

More details about the choices can be found in *MCEq.density_profiles*. Calling this method will issue a recalculation of the interpolation and the integration path.

Parameters **density_config** (*tuple of strings*) – (parametrization type, arguments)

set_interaction_model(*interaction_model*, *charm_model=None*, *force=False*)

Sets interaction model and/or an external charm model for calculation.

Decay and interaction matrix will be regenerated automatically after performing this call.

Parameters

- **interaction_model** (*str*) – name of interaction model
- **charm_model** (*str, optional*) – name of charm model
- **force** (*bool*) – force loading interaction model

set_mod_pprod(*prim_pdg*, *sec_pdg*, *x_func*, *x_func_args*, *delay_init=False*)

Sets combination of projectile/secondary for error propagation.

The production spectrum of `sec_pdg` in interactions of `prim_pdg` is modified according to the function passed to `InteractionYields.init_mod_matrix()`

Parameters

- **prim_pdg** (*int*) – interacting (primary) particle PDG ID
- **sec_pdg** (*int*) – secondary particle PDG ID
- **x_func** (*object*) – reference to function
- **x_func_args** (*tuple*) – arguments passed to `x_func`
- **delay_init** (*bool*) – Prevent init of mceq matrices if you are planning to add more modifications

set_obs_particles(*obs_ids*)

Adds a list of mother particle strings which decay products should be scored in the special `obs_` category.

Decay and interaction matrix will be regenerated automatically after performing this call.

Parameters **obs_ids** (*list of strings*) – mother particle names

set_primary_model(*mclass*, *tag*)
Sets primary flux model.

This functions is quick and does not require re-generation of matrices.

Parameters

- **interaction_model** (CRFluxModel.PrimaryFlux) – reference
- **primary model class** (*to*) –
- **tag** (*tuple*) – positional argument list for model class

set_single_primary_particle(*E*, *corsika_id*)
Set type and energy of a single primary nucleus to calculation of particle yields.

The functions uses the superposition theorem, where the flux of a nucleus with mass A and charge Z is modeled by using Z protons and A-Z neutrons at energy $E_{nucleon} = E_{nucleus}/A$ The nucleus type is defined via CORSIKA ID $= A * 100 + Z$. For example iron has the CORSIKA ID 5226.

A continuous input energy range is allowed between $50 * A$ GeV $< E_{\text{nucleus}} < 10^{10} * A$GeV.

Parameters

- **E** (*float*) – (total) energy of nucleus in GeV
- **corsika_id** (*int*) – ID of nucleus (see text)

set_theta_deg(*theta_deg*)
Sets zenith angle θ as seen from a detector.

Currently only 'down-going' angles (0-90 degrees) are supported.

Parameters **atm_config** (*tuple of strings*) – (parametrization type, location string, season string)

solve(***kwargs*)
Launches the solver.

The setting *integrator* in the config file decides which solver to launch, either the simple but accelerated explicit Euler solvers, MCEqRun._forward_euler() or, solvers from ODEPACK MCEqRun._odepack().

Parameters **kwargs** (*dict*) – Arguments are passed directly to the solver methods.

unset_mod_pprod()
Removes modifications from *MCEqRun.set_mod_pprod()*.

cs = None
handler for cross-section data of type *MCEq.data.HadAirCrossSections*

d = None
(int) dimension of energy grid

ds = None
handler for decay yield data of type *MCEq.data.DecayYields*

e_grid = None
(np.array) energy grid (bin centers)

modtab = None
instance of ParticleDataTool.SibyllParticleTable – access to properties lists of particles, index translation etc.

pd = None
instance of `ParticleDataTool.PYTHIAParticleData` – access to properties of particles, like mass and charge

y = None
handler for decay yield data of type *MCEq.data.InteractionYields*

MCEq.data — data management

This module includes code for bookkeeping, interfacing and validating data structures:

- *InteractionYields* manages particle interactions, obtained from sampling of various interaction models
- *DecayYields* manages particle decays, obtained from sampling PYTHIA8 Monte Carlo
- *HadAirCrossSections* keeps information about the inelastic, cross-section of hadrons with air. Typically obtained from Monte Carlo.
- *MCEqParticle* bundles different particle properties for simpler usage in `MCEqRun`
- `EdepZFactos` calculates energy-dependent spectrum weighted moments (Z-Factors)

class `MCEq.data.`**DecayYields**(*mother_list=None*, *fname=None*)
Class for managing the dictionary of decay yield matrices.

The class un-pickles a dictionary, which contains x spectra of decay products/daughters, sampled from PYTHIA 8 Monte Carlo.

Parameters **mother_list** (*list, optional*) – list of particle mothers from interaction model

assign_d_idx(*mother*, *moidx*, *daughter*, *dtridx*, *dmat*)
Copies a subset, defined in tuples `moidx` and `dtridx` from the `decay_matrix(mother, daughter)` into `dmat`

Parameters

- **mother** (*int*) – PDG ID of mother particle
- **moidx** (*int, int*) – tuple containing index range relative to the mothers's energy grid
- **daughter** (*int*) – PDG ID of final state daughter/secondary particle
- **dtridx** (*int, int*) – tuple containing index range relative to the daughters's energy grid
- **dmat** (*numpy.array*) – array reference to the decay matrix

daughters(*mother*)
Checks if `mother` decays and returns the list of daughter particles.

Parameters **mother** (*int*) – PDG ID of projectile particle

Returns PDG IDs of daughter particles

Return type list

get_d_matrix(*mother*, *daughter*)
Returns a `DIM x DIM` decay matrix.

Parameters

- **mother** (*int*) – PDG ID of mother particle

- **daughter** (*int*) – PDG ID of final state daughter particle

Returns decay matrix

Return type numpy.array

Note: In the current version, the matrices have to be multiplied by the bin widths. In later versions they will be stored with the multiplication carried out.

---

is_daughter(*mother*, *daughter*)
Checks if `daughter` is a decay daughter of `mother`.

Parameters

- **mother** (*int*) – PDG ID of projectile particle
- **daughter** (*int*) – PDG ID of daughter particle

Returns `True` if `daughter` is daughter of `mother`

Return type bool

particle_list = None
(list) List of particles in the decay matrices

class MCEq.data.**HadAirCrossSections**(*interaction_model*)
Class for managing the dictionary of hadron-air cross-sections.

The class unpickles a dictionary, which contains proton-air, pion-air and kaon-air cross-sections tabulated on the common energy grid.

Parameters **interaction_model** (*str*) – name of the interaction model

get_cs(*projectile*, *mbarn=False*)
Returns inelastic `projectile`-air cross-section $\sigma_{inel}^{proj-Air}(E)$ as vector spanned over the energy grid.

Parameters

- **projectile** (*int*) – PDG ID of projectile particle
- **mbarn** (*bool, optional*) – if `True`, the units of the cross-section will be $mbarn$, else cm^2

Returns cross-section in $mbarn$ or cm^2

Return type numpy.array

set_interaction_model(*interaction_model*)
Selects an interaction model and prepares all internal variables.

Parameters **interaction_model** (*str*) – interaction model name

Raises `Exception` – if invalid name specified in argument `interaction_model`

GeV2mbarn = 0.38937930376300284
unit - $\text{GeV}^2 \cdot$ mbarn

GeVcm = 1.9732696312541852e-14
unit - GeV · cm

GeVfm = 0.19732696312541853
unit - GeV · fm

egrid = None
current energy grid

iam = None
current interaction model name

mbarn2cm2 = 9.999999999999999e-28
unit conversion - mbarn $\rightarrow$ cm^2

class MCEq.data.**InteractionYields**(*interaction_model*, *charm_model=None*)
Class for managing the dictionary of interaction yield matrices.

The class unpickles a dictionary, which contains the energy grid and x spectra, sampled from hadronic interaction models.

A list of available interaction model keys can be printed by:

```
$ print yield_obj
```

Parameters

- **interaction_model** (*str*) – name of the interaction model
- **charm_model** (*str, optional*) – name of the charm model

assign_yield_idx(*projectile*, *projidx*, *daughter*, *dtridx*, *cmat*)
Copies a subset, defined in tuples `projidx` and `dtridx` from the `yield_matrix(projectile, daughter)` into `cmat`

Parameters

- **projectile** (*int*) – PDG ID of projectile particle
- **projidx** (*int, int*) – tuple containing index range relative to the projectile's energy grid
- **daughter** (*int*) – PDG ID of final state daughter/secondary particle
- **dtridx** (*int, int*) – tuple containing index range relative to the daughters's energy grid
- **cmat** (*numpy.array*) – array reference to the interaction matrix

get_y_matrix(*projectile*, *daughter*)
Returns a `DIM x DIM` yield matrix.

Parameters

- **projectile** (*int*) – PDG ID of projectile particle
- **daughter** (*int*) – PDG ID of final state daughter/secondary particle

Returns yield matrix

Return type numpy.array

Note: In the current version, the matrices have to be multiplied by the bin widths. In later versions they will be stored with the multiplication carried out.

---

is_yield(*projectile*, *daughter*)
Checks if a non-zero yield matrix exist for `projectile- daughter` combination (deprecated)

Parameters

- **projectile** (*int*) – PDG ID of projectile particle

- **daughter** (*int*) – PDG ID of final state daughter/secondary particle

Returns `True` if non-zero interaction matrix exists else `False`

Return type bool

print_mod_pprod()
Prints the active particle production modification.

set_interaction_model(*interaction_model*, *force=False*)
Selects an interaction model and prepares all internal variables.

Parameters

- **interaction_model** (*str*) – interaction model name
- **force** (*bool*) – forces reloading of data from file

Raises `Exception` – if invalid name specified in argument `interaction_model`

set_xf_band(*xl_low_idx*, *xl_up_idx*)
Limits interactions to certain range in $x_{\rm lab}$.

Limit particle production to a range in $x_{\rm lab}$ given by lower index, below which no particles are produced and an upper index, respectively. (Needs more clarification).

Parameters

- **xl_low_idx** (*int*) – lower index of $x_{\rm lab}$ value
- **xl_up_idx** (*int*) – upper index of $x_{\rm lab}$ value

band = None
(tuple) selection of a band of coeffictients (in xf)

charm_model = None
(str) charm model name

dim = None
(int) dimension of grid

e_bins = None
(numpy.array) energy grid bin endges

e_grid = None
(numpy.array) energy grid bin centers

iam = None
(str) InterAction Model name

mod_pprod = None
(tuple) modified particle combination for error prop.

particle_list = None
(list) List of particles supported by interaction model

weights = None
(numpy.array) energy grid bin widths

xmat = None
(numpy.array) Matrix of x_lab values

class `MCEq.data.`**MCEqParticle**(*pdgid*, *particle_db*, *pythia_db*, *cs_db*, *d*, *max_density=0.00124*)
Bundles different particle properties for simplified availability of particle properties in *MCEq.core.MCEqRun*.

Parameters

- **pdgid** (*int*) – PDG ID of the particle
- **particle_db** (*object*) – handle to an instance of `ParticleDataTool.SibyllParticleTable`
- **pythia_db** (*object*) – handle to an instance of `ParticleDataTool.PYTHIAParticleData`
- **cs_db** (*object*) – handle to an instance of *InteractionYields*
- **d** (*int*) – dimension of the energy grid

calculate_mixing_energy(*e_grid*, *no_mix=False*, *max_density=0.00124*)

Calculates interaction/decay length in Air and decides if the particle has resonance and/or hadron behavior.

Class attributes *is_mixed*, *E_mix*, *mix_idx*, *is_resonance* are calculated here.

Parameters

- **e_grid** (*numpy.array*) – energy grid of size `d`
- **no_mix** (*bool*) – if True, mixing is disabled and all particles have either hadron or resonances behavior.
- **max_density** (*float*) – maximum density on the integration path (largest decay length)

critical_energy()

Returns critical energy where decay and interaction are balanced.

Approximate value in Air.

Returns $\frac{m\ 6.4\text{km}}{c\tau}$ in GeV

Return type (float)

hadridx()

Returns index range where particle behaves as hadron.

Returns range on energy grid

Return type `tuple()` (int,int)

inverse_decay_length(*E*, *cut=True*)

Returns inverse decay length (or infinity (np.inf), if particle is stable), where the air density ρ is factorized out.

Parameters

- **E** (*float*) – energy in laboratory system in GeV
- **cut** (*bool*) – set to zero in 'resonance' regime

Returns $\frac{\rho}{\lambda_{dec}}$ in 1/cm

Return type (float)

inverse_interaction_length(*cs=None*)

Returns inverse interaction length for A_target given by config.

Returns $\frac{1}{\lambda_{int}}$ in cm**2/g

Return type (float)

lidx()

Returns lower index of particle range in state vector.

Returns lower index in state vector `MCEqRun.phi`

Return type (int)

residx()
Returns index range where particle behaves as resonance.

Returns range on energy grid

Return type tuple() (int,int)

uidx()
Returns upper index of particle range in state vector.

Returns upper index in state vector MCEqRun.phi

Return type (int)

E_crit = None
(float) critical energy in air at the surface

E_mix = None
(float) mixing energy, transition between hadron and resonance behavior

is_alias = None
(bool) particle is an alias (PDG ID encodes special scoring behavior)

is_baryon = None
(bool) particle is a baryon

is_hadron = None
(bool) particle is a hadron (meson or baryon)

is_lepton = None
(bool) particle is a lepton

is_meson = None
(bool) particle is a meson

is_mixed = None
(bool) particle has both, hadron and resonance properties

is_projectile = None
(bool) particle is interacting projectile

is_resonance = None
(bool) if particle has just resonance behavior

mix_idx = None
(int) energy grid index, where transition between hadron and resonance occurs

nceidx = None
(int) MCEq ID

pdgid = None
(int) Particle Data Group Monte Carlo particle ID

MCEq.data_utils — file operations on MCEq databases

This module contains function to convert data files used by MCEq.

- *convert_to_compact()* converts an interaction model file into "compact" mode

- *extend_to_low_energies()* extends an interaction model file with an low energy interaction model using interpolation

MCEq.data_utils.**convert_to_compact**(*fname*)
Converts an interaction model dictionary to "compact" mode.

This function takes a compressed yield file, where all secondary particle types known to the particular model are expected to be final state particles (set stable in the MC), and converts it to a new compressed yield file which contains only production channels to the most important particles (for air-shower and inclusive lepton calculations).

The production of short lived particles and resonances is taken into account by executing the convolution with their decay distributions into more stable particles, until only final state particles are left. The list of "important" particles is defined in the standard_particles variable below. This results in a feed-down corretion, for example the process (chain) $p + A \to \rho + X \to \pi + \pi + X$ becomes simply $p + A \to \pi + \pi + X$. The new interaction yield file obtains the suffix *_compact* and it contains only those final state secondary particles:

$$\pi^+, K^+, K^0_{S,L}, p, n, \bar{p}, \bar{n}, \Lambda^0, \bar{\Lambda}^0, \eta, \phi, \omega, D^0, D^+, D^+_s + \text{c.c.} + \text{leptons}$$

The compact mode has the advantage, that the production spectra stored in this dictionary are directly comparable to what accelerators consider as stable particles, defined by a minimal life-time requirement. Using the compact mode is recommended for most applications, which use *MCEq.core.MCEqRun.set_mod_pprod()* to modify the spectrum of secondary hadrons.

For some interaction models, the performance advantage can be around 50%. The precision loss is negligible at energies below 100 TeV, but can increase up to a few % at higher energies where prompt leptons dominate. This is because also very short-lived charmed mesons and baryons with small branching ratios into leptons can interact with the atmosphere and lose energy before decay.

For *QGSJET*, compact and normal mode are identical, since the model does not produce resonances or rare mesons by design.

Parameters **fname** (*str*) – name of compressed yield (.bz2) file

MCEq.data_utils.**extend_to_low_energies**(*he_di=None*, *le_di=None*, *fname=None*)
Interpolates between a high-energy and a low-energy interaction model.

Theis function takes either two yield dictionaries or a file name of the high energy model and interpolates the matrices at the energy specified in *:mod:mceq_config* in the low_energy_extension section. The interpolation is linear in energy grid index.

In 'compact' mode all particles should be supported by the low energy model. However if you don't use compact mode, some rare or exotic secondaries might be not supported by the low energy model. In this case the config option "use_unknown_cs" decides if only the high energy part is used or if to raise an excption.

Parameters

- **he_di** (*dict, optional*) – yield dictionary of high-energy model
- **le_di** (*dict, optional*) – yield dictionary of low-energy model
- **fname** (*str, optional*) – file name of high-energy model yields

---

MCEq.kernels — calculation kernels for the forward-euler integrator

The module contains functions which are called by the forward-euler integration routine MCEq.core.MCEqRun.forward_euler().

The integration is part of these functions. The single step

$$\Phi_{i+1} = \left[\boldsymbol{M}_{int} + \frac{1}{\rho(X_i)}\boldsymbol{M}_{dec}\right] \cdot \Phi_i \cdot \Delta X_i$$

with

$$\boldsymbol{M}_{int} = (-\mathbf{1} + \boldsymbol{C})\boldsymbol{\Lambda}_{int} \tag{3.1}$$

and

$$\boldsymbol{M}_{dec} = (-\mathbf{1} + \boldsymbol{D})\boldsymbol{\Lambda}_{dec}. \tag{3.2}$$

The functions use different libraries for sparse and dense linear algebra (BLAS):

- The default for dense or sparse matrix representations is the function *kern_numpy()*. It uses the dot-product implementation of numpy. Depending on the details, your numpy installation can be already linked to some BLAS library like as ATLAS or MKL, what typically accelerates the calculation significantly.
- The fastest version, *kern_MKL_sparse()*, directly interfaces to the sparse BLAS routines from Intel MKL via ctypes. If you have the MKL runtime installed, this function is recommended for most purposes.
- The GPU accelerated versions *kern_CUDA_dense()* and *kern_CUDA_sparse()* are implemented using the cuBLAS or cuSPARSE libraries, respectively. They should be considered as experimental or implementation examples if you need extremely high performance. To keep Python as the main programming language, these interfaces are accessed via the module numbapro, which is part of the Anaconda Accelerate package. It is free for academic use.

MCEq.kernels.**kern_CUDA_dense**(*nsteps*, *dX*, *rho_inv*, *int_m*, *dec_m*, *phi*, *grid_idcs*, *mu_egrid=None*, *mu_dEdX=None*, *mu_lidx_nsp=None*, *prog_bar=None*)

NVIDIA CUDA cuBLAS implementation of forward-euler integration.

Function requires a working numbapro installation. It is typically slower compared to *kern_MKL_sparse()* but it depends on your hardware.

Parameters

- **nsteps** (*int*) – number of integration steps
- **dX** (*numpy.array[nsteps]*) – vector of step-sizes ΔX_i in g/cm**2
- **rho_inv** (*numpy.array[nsteps]*) – vector of density values $\frac{1}{\rho(X_i)}$
- **int_m** (*numpy.array*) – interaction matrix (3.1) in dense or sparse representation
- **dec_m** (*numpy.array*) – decay matrix (3.2) in dense or sparse representation
- **phi** (*numpy.array*) – initial state vector $\Phi(X_0)$
- **prog_bar** (*object,optional*) – handle to ProgressBar object

Returns state vector $\Phi(X_{nsteps})$ after integration

Return type numpy.array

MCEq.kernels.**kern_CUDA_sparse**(*nsteps*, *dX*, *rho_inv*, *context*, *phi*, *grid_idcs*, *mu_egrid=None*, *mu_dEdX=None*, *mu_lidx_nsp=None*, *prog_bar=None*)

NVIDIA CUDA cuSPARSE implementation of forward-euler integration.

Function requires a working accelerate installation.

Parameters

- **nsteps** (*int*) – number of integration steps

- **dX** (*numpy.array[nsteps]*) – vector of step-sizes ΔX_i in g/cm**2
- **rho_inv** (*numpy.array[nsteps]*) – vector of density values $\frac{1}{\rho(X_i)}$
- **int_m** (*numpy.array*) – interaction matrix (3.1) in dense or sparse representation
- **dec_m** (*numpy.array*) – decay matrix (3.2) in dense or sparse representation
- **phi** (*numpy.array*) – initial state vector $\Phi(X_0)$
- **prog_bar** (*object,optional*) – handle to `ProgressBar` object

Returns state vector $\Phi(X_{nsteps})$ after integration

Return type numpy.array

`MCEq.kernels.`**kern_MKL_sparse**(*nsteps*, *dX*, *rho_inv*, *int_m*, *dec_m*, *phi*, *grid_idcs*, *mu_egrid=None*, *mu_dEdX=None*, *mu_lidx_nsp=None*, *prog_bar=None*)

Intel MKL sparse BLAS implementation of forward-euler integration.

Function requires that the path to the MKL runtime library `libmkl_rt.[so/dylib]` defined in the config file.

Parameters

- **nsteps** (*int*) – number of integration steps
- **dX** (*numpy.array[nsteps]*) – vector of step-sizes ΔX_i in g/cm**2
- **rho_inv** (*numpy.array[nsteps]*) – vector of density values $\frac{1}{\rho(X_i)}$
- **int_m** (*numpy.array*) – interaction matrix (3.1) in dense or sparse representation
- **dec_m** (*numpy.array*) – decay matrix (3.2) in dense or sparse representation
- **phi** (*numpy.array*) – initial state vector $\Phi(X_0)$
- **grid_idcs** (*list*) – indices at which longitudinal solutions have to be saved.
- **prog_bar** (*object,optional*) – handle to `ProgressBar` object

Returns state vector $\Phi(X_{nsteps})$ after integration

Return type numpy.array

`MCEq.kernels.`**kern_XeonPHI_sparse**(*nsteps*, *dX*, *rho_inv*, *int_m*, *dec_m*, *phi*, *grid_idcs*, *mu_egrid=None*, *mu_dEdX=None*, *mu_lidx_nsp=None*, *prog_bar=None*)

Experimental Xeon Phi support using pyMIC library.

`MCEq.kernels.`**kern_numpy**(*nsteps*, *dX*, *rho_inv*, *int_m*, *dec_m*, *phi*, *grid_idcs*, *mu_egrid=None*, *mu_dEdX=None*, *mu_lidx_nsp=None*, *prog_bar=None*, *fa_vars=None*)

:mod;‘numpy‘ implementation of forward-euler integration.

Parameters

- **nsteps** (*int*) – number of integration steps
- **dX** (*numpy.array[nsteps]*) – vector of step-sizes ΔX_i in g/cm**2
- **rho_inv** (*numpy.array[nsteps]*) – vector of density values $\frac{1}{\rho(X_i)}$
- **int_m** (*numpy.array*) – interaction matrix (3.1) in dense or sparse representation
- **dec_m** (*numpy.array*) – decay matrix (3.2) in dense or sparse representation
- **phi** (*numpy.array*) – initial state vector $\Phi(X_0)$
- **prog_bar** (*object,optional*) – handle to `ProgressBar` object

- **fa_vars** (*dict, optional*) – contains variables for first interaction mode

Returns state vector $\Phi(X_{nsteps})$ after integration

Return type numpy.array

MCEq.misc - other useful things

Some helper functions and plotting features are collected in this module.

class MCEq.misc.**EdepZFactors**(*interaction_model*, *primary_flux_model*)
Handles calculation of energy dependent Z factors.

Was not recently checked and results could be wrong.

MCEq.misc.**cornertext**(*text*, *loc=2*, *color=None*, *frameon=False*, *axes=None*, ***kwargs*)
Conveniently places text in a corner of a plot.

Parameters

- **text** (*string or tuple of strings*) – Text to be placed in the plot. May be a tuple of strings to get several lines of text.
- **loc** (*integer or string*) – Location of text, same as in legend(...).
- **frameon** (*boolean (optional)*) – Whether to draw a border around the text. Default is False.
- **axes** (*Axes (optional, default: None)*) – Axes object which houses the text (defaults to the current axes).
- **fontproperties** (*matplotlib.font_manager.FontProperties object*) – Change the font style.
- **keyword arguments are forwarded to the text instance.** (*Other*) –
- **Authors** –
- **-------** –
- **Dembinski <hans.dembinski@kit.edu>** (*Hans*) –

MCEq.misc.**get_bins_and_width_from_centers**(*vector*)
Returns bins and bin widths given given bin centers.

MCEq.misc.**normalize_hadronic_model_name**(*name*)
Converts hadronic model name into standard form

MCEq.misc.**plot_hist**(*xedges*, *ws*, *axes=None*, *facecolor=None*, ***kwargs*)
Plots histogram data in ROOT style.

Parameters

- **xedge** (*lower bin boundaries + upper boundary of last bin*) –
- **w** (*content of the bins*) –

MCEq.misc.**print_in_rows**(*str_list*, *n_cols=8*)
Prints contents of a list in rows *n_cols* entries per row.

MCEq.misc.**set_ticks**(*which*, *n_divs=5*, *ax=None*)
Helps to control the number of ticks on x and y axes.

Parameters **which** (*str*) – can be *x* or *y* or *both*

MCEq.misc.**theta_deg**(*cos_theta*)
Converts $\cos\theta$ to θ in degrees.

MCEq.misc.**theta_rad**(*theta*)
Converts θ from rad to degrees.

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Python Module Index

m

Index

Symbols

A

B

C

D

E

G

H

I

K

L

M

N