pyMonteCarlo Documentation

Release 1.1.0+8.g0bda5a5

Philippe T. Pinard

Jan 16, 2022

Contents

pyMonteCarlo is a programming interface to run identical simulations using different Monte Carlo programs. The interface was designed to have common input and output that are independent of any Monte Carlo code. This allows users to combine the advantages of different codes and to compare the effect of different physical models without manually creating and running new simulations for each Monte Carlo program. The analysis of the results is also simplified by the common output format where results are expressed in the same units.

pyMonteCarlo is mainly designed and developed to fulfill simulation needs and solve problems faced by the electron microscopy and microanalysis community. Adapting **pyMonteCarlo** to other scientific fields is not completely excluded, but it is a more long-term objective.

CHAPTER 1

Goals

pyMonteCarlo has the following goals:

- Provide a common interface to setup Monte Carlo simulations for the electron microscopy and microanalysis community.
- Provide a common interface to analyze and report results from simulations.
- Easy way to create several simulations by varying one or more parameters.
- Store results in a open, easy accessible file format.
- Support several Monte Carlo programs.
- Be extensible to new Monte Carlo programs.
- Run on multiple operating systems, including on computer clusters.

Reversely, **pyMonteCarlo** does *not* attempt to:

- Support all the finer details and particularities of each Monte Carlo program. **pyMonteCarlo** tries to be as general as possible. This allows the same simulation options to be simulated on several programs.
- Provide new features to existing Monte Carlo program. Each program is taken as is. The development of new features is left to the original authors of the program.
- Provide a complete suite of analysis tools. **pyMonteCarlo** provides general tools to tabulate and plot results. This cannot however fulfill the needs of all users. Results can be exported to *csv* and *Excel* file. Users can also write their own Python scripts to analyze their results.

CHAPTER 2

License

pyMonteCarlo and the packages providing interfaces to Monte Carlo programs are licensed under Apache Software License 2.0.

CHAPTER 3

Contributors

- Philippe T. Pinard (High Wycombe, United Kingdom)
- Hendrix Demers (Montreal, Canada)
- Raynald Gauvin (McGill University, Montreal, Canada)
- Silvia Richter (RWTH Aachen University, Aachen, Germany)

3.1 Installation

3.1.1 Stable release

There are two ways to install **pyMonteCarlo**. The simplest one is the *stand-alone version*, which comes as a *zip* file containing an application executable under Windows. The more advanced one is to install **pyMonteCarlo** as a *Python package* from the Python Package Index (PyPI). Instructions for both methods are given below as well as how to install **pyMonteCarlo** for *developers*.

Stand-alone, graphical user interface

Note: Stand-alone version for MacOS and Linux are under development. Users of these operating systems can install **pyMonteCarlo** as a Python package. See instructions below.

Windows

For Windows, **pyMonteCarlo** is packaged as a stand-alone distribution. The latest release can be downloaded here. To install, simply extract the content of the *zip* file and run *pymontecarlo.exe*. This distribution is bundled with all *supported Monte Carlo programs*.

Python package

pyMonteCarlo only supports Python 3.7+. It is recommended to install the latest Python version. You can download Python from the official release. Using other Python distribution like Anaconda, minconda, etc. should also work.

To install **pyMonteCarlo**, run this command in your command prompt/terminal:

```
$ pip install pymontecarlo pymontecarlo-gui
```

This is the preferred method to install **pyMonteCarlo**, as it will always install the most recent stable release.

pyMonteCarlo package provides the core, common functionalities. In other words, it does not contain the interface to Monte Carlo programs. Each interface has its own package. The supported Monte Carlo programs are listed *here*.

As a starting point, it is recommended to install *pymontecarlo-casino2*. Most of the examples in this documentation are based on *pymontecarlo-casino2*. This Monte Carlo program works on Windows and after installation *Wine* on MacOS and Linux. Installation instructions are *below*.

```
$ pip install pymontecarlo-casino2
```

If your *PYTHONPATH* is properly configured, you should be able to run **pyMonteCarlo** graphical user interface by simply typing **pyMonteCarlo** in a command prompt/terminal:

```
$ pymontecarlo
```

If not, another way to start **pyMonteCarlo** is the following:

```
$ python -m pymontecarlo_gui
```

3.1.2 Development

Warning: Many projects in the **pyMonteCarlo** organization uses Git LFS. Please make sure it is installed before cloning any repository.

Clone the **pyMonteCarlo** Github repository, either directly or after forking:

```
$ git clone git://github.com/pymontecarlo/pymontecarlo
```

Install the project in editable mode:

```
$ cd pymontecarlo
$ pip install -e .[dev]
```

Run the unit tests to make sure everything works properly:

```
$ pytest
```

Repeat the same procedure for any other **pyMonteCarlo** projects in the Github **pyMonteCarlo** organization.

3.1.3 Wine

Wine is a Windows emulator for MacOS and Linux. Since some Monte Carlo programs are only available on Windows, *Wine* is a way to run them on other operating systems. Please refer to the *Wine* website to download the latest version and the platform-specific installation instructions. **pyMonteCarlo** assumes that *Wine* is properly installed and that the *wine* executable is in the *PATH*.

3.2 Tutorials

3.2.1 First simulation

A (Monte Carlo) Simulation consists in (1) *options*, defining all the necessary parameters to setup the simulation, and (2) *results*, containing all the outputs of a simulation. One or more simulations form a `Project`. A **pyMonteCarlo** project stored on disk has the extension `.mcsim`. It consists of a HDF5 file and can be opened in the HDFViewer or using any HDF5 library.

Setting up simulation options

The options are defined by the class `Options`. It contains all the parameters necessary to run **one** simulation. The parameters are grouped into four categories:

- program
- beam
- sample
- analyses

The beam, sample and analyses are independent of Monte Carlo programs. In other words, the same sample definition can be used for different Monte Carlo programs. For a given `Options` instance, only the program needs to change to run the same simulation with different Monte Carlo programs. That being said not all beam, sample and analyses are supported by all Monte Carlo programs. Supported parameters for each Monte Carlo program are listed in the *supported options* page.

Program

The program is specific to a particular Monte Carlo program. Each program follows the contract specified by the base program class `Program`. One implementation is for Casino 2 as part of the package **pymontecarlo-casino2**. The program can be imported as follow:

```
[ ]: from pymontecarlo_casino2.program import Casino2Program
```

The parameters associated with the program will depend on each Monte Carlo program. For Casino 2, the number of trajectories and the models used for the simulation can be specified. Here is an example with the default models and 5000 trajectories:

```
[ ]: program = Casino2Program(5000)
```

Throughout **pyMonteCarlo**, a parameter can also be set/modified using its attribute inside the class:

```
[ ]: program.number_trajectories = 6000
```

All parameters are completely mutable and are only validated before a simulation starts.

Beam

The second category of parameters is the beam. At the moment, three types of beam are implemented/supported:

- a pencil beam: beam with no diameter,
- a Gaussian beam: beam where the electrons are randomly distributed following a two dimensional Gaussian distribution, where the diameter is defined as full width at half maximum (FWHM),
- a cylindrical beam: beam where the electrons are randomly distributed within a cylinder.

All beam implementations must define the energy and type of the incident particles as defined by the base `Beam` class. The type of incident particle is defined for future expansions, since all currently supported Monte Carlo programs only accept `ELECTRON`. Unless otherwise stated, all beams assume that the incident particles travel downwards along the z-axis, i.e. following the vector `(0, 0, -1)`.

The pencil beam is the most supported by the different Monte Carlo programs as no diameter is defined. Here is an example of a pencil beam with a beam energy of 15keV:

```
[ ]: from pymontecarlo.options.beam import PencilBeam
     beam = PencilBeam(15e3)
```

Other parameters of the beam are the beam center position. By default, the beam is centered at `x = 0m` and `y = 0m`. The position can be changed using either attribute:

```
[ ]: beam.x_m = 100e-9
     beam.y_m = 200e-9
```

Sample

The sample parameter defines the geometry and the materials of the sample being bombarded by the incident particles. There are currently 5 types of sample implemented:

- substrate (`SubstrateSample`): An infinitely thick sample.
- inclusion (`InclusionSample`): An half-sphere inclusion in a substrate.
- horizontal layered (`HorizontalLayerSample`): Creates a multi-layers geometry. The layers are assumed to be in the x-y plane (normal parallel to z) at tilt of 0.0°.
- vertical layered (`VericalLayerSample`): Creates a grain boundaries sample. It consists of 0 or many layers in the y-z plane (normal parallel to x) simulating interfaces between different materials. If no layer is defined, the geometry is a couple.
- sphere (`SphereSample`): A sphere in vacuum.

For all types of sample, the sample is entirely located below the `z = 0` plane. While some Monte Carlo programs support custom and complex sample definitions, it was chosen for simplicity and compatibility to constrain the available types of sample. If you would like to suggest/contribute another type of sample, please open an enhancement issue or submit a pull request.

Before creating a sample, material(s) must be defined. A material defines the composition and density in a part of the sample (e.g. layer or substrate). After importing the `Material` class,

```
[ ]: from pymontecarlo.options.material import Material
```

There are three ways to create a material:

1. Pure, single element material:

```
[ ]: material = Material.pure(14) # pure silicon
```

2. A chemical formula:

```
[ ]: material = Material.from_formula('SiO2')
```

3. Composition in mass fraction. The composition is expressed as a *dict* where keys are atomic numbers and values, mass fractions:

```
[ ]: composition = {29: 0.4, 30: 0.6}
     material = Material('Brass', composition)
```

In all three cases the mass density (in kg/m3) can be specified as an argument or set from its attribute:

```
[ ]: material.density_kg_per_m3 = 8400
     material.density_g_per_cm3 = 8.4
```

If the density is not specified, it is calculated using this following formula:

$$\frac{1}{\rho} = \sum \frac{m_i}{\rho_i}$$

where ρ_i and m_i are respectively the elemental mass density and mass fraction of element i.

Each sample has different methods and variables to setup the materials. Here is an example for the substrate sample:

```
[ ]: from pymontecarlo.options.sample import SubstrateSample
     from pymontecarlo.options.material import Material

     copper = Material.pure(29)
     substrate = SubstrateSample(copper)
```

and here is an example for the horizontal layered sample. The substrate is set to copper and two layers are added on top, forming from top to bottom: 100nm of SiO2, 50nm of brass and then copper:

```
[ ]: from pymontecarlo.options.sample import HorizontalLayerSample
     from pymontecarlo.options.material import Material

     copper = Material.pure(29)
     sio2 = Material.from_formula('SiO2')
     brass = Material('Brass', {29: 0.4, 30: 0.6})

     sample = HorizontalLayerSample(copper)
     sample.add_layer(sio2, 100e-9)
     sample.add_layer(brass, 50e-9)
```

One trick to make sure the sample is properly setup is to draw it. **pyMonteCarlo** uses matplotlib to draw the sample in 2D along the XZ, YZ or XY perspective. Here is an example:

```
[ ]: import matplotlib.pyplot as plt
     from pymontecarlo.figures.sample import SampleFigure, Perspective

     fig, axes = plt.subplots(1, 3, figsize=(10, 3))

     samplefig = SampleFigure(sample, beams=[beam])

     for ax, perspective in zip(axes, Perspective):
```

(continues on next page)

(continued from previous page)

```
    samplefig.perspective = perspective
    samplefig.draw(ax)

plt.show()
```

Analyses

The analyses define which results from the Monte Carlo simulation will be processed and stored by **pyMonteCarlo**. To see a list of the supported analyses, please refer to *supported options* page.

Here is how to store the X-ray intensityies emitted from the sample. First we need to define a photon detector. Each detector requires a name, and the photon detector, an additional argument specifying its elevation, i.e. the angle between the detector and the XY plane.

[ ]:
```
from pymontecarlo.options.detector import PhotonDetector
import math
detector = PhotonDetector(name='detector1', elevation_rad=math.radians(40))
```

The photon detector is then used to create a new analysis.

[ ]:
```
from pymontecarlo.options.analysis import PhotonIntensityAnalysis
analysis = PhotonIntensityAnalysis(detector)
```

Options

The final step is to put together the program, beam, sample and analysis and create an `Options`. Note that the options can take several analyses, but in this example we only specified one.

[ ]:
```
from pymontecarlo.options import Options
options = Options(program, beam, sample, [analysis])
```

Running simulation(s)

We are now ready to run the simulation. **pyMonteCarlo** provides an helper function to run several simulation options. These options and their results are automatically stored in a `Project` object, which can be stored on disk and viewed in either the HDFViewer or the graphical interface of **pyMonteCarlo**. The results can also be processed programatically, as it will be demonstrated in this tutorial.

[ ]:
```
from pymontecarlo.runner.helper import run_async
project = await run_async([options])

import os
import tempfile
project.write(os.path.join(tempfile.gettempdir(), 'project1.h5'))
```

Interpreting simulation results

Let's now explore the results. Each simulation gets stored in the `simulations` attribute of the `project` object:

```
[ ]: project.simulations
```

Each simulation consists in the `options` used to setup the simulation and the `results`, which is a list of result objects.

```
[ ]: simulation = project.simulations[0]
     print('Simulation at {}keV contains {} result(s)'.format(simulation.options.beam.
     ↪energy_keV, len(simulation.results)))
```

The `PhotonIntensityAnalysis` returns an `EmittedPhotonIntensityResult`, which essentially consists of a dictionary, where the keys are the emitted X-ray lines and the values, their intensities. Here is a quick way to list all X-ray lines. The attribute `siegbahn` can be replaced with `iupac` if this notation is preferred. As shown below, the total X-ray intensity of a family of lines (e.g. K, L) is automatically calculated.

```
[ ]: result = simulation.results[0]
     print('Available X-ray intensities:')
     for xrayline in result:
         print(xrayline.siegbahn)
```

And here is how to retrieve the intensity of one line.

```
[ ]: import pyxray
     xrayline = pyxray.xray_line('Si', 'Ka')
     intensity = result[xrayline]
     print('X-ray intensity of {}: {} +/- {}'.format(xrayline.siegbahn, intensity.n,␣
     ↪intensity.s))
```

Another way to analyze the result is to convert them to a pandas dataframe. The project has two methods to create a data frame for the options using `create_options_dataframe(...)` and for the results `create_results_dataframe(...)`. Each row in these data frames corresponds to one simulation. Both have one required `settings` argument to specify the X-ray notation and units used.

```
[ ]: from pymontecarlo.settings import Settings, XrayNotation
     settings = Settings()
     settings.set_preferred_unit('eV')
     settings.set_preferred_unit('nm')
     settings.set_preferred_unit('deg')
     settings.preferred_xray_notation = XrayNotation.SIEGBAHN
```

```
[ ]: project.create_options_dataframe(settings)
```

```
[ ]: project.create_results_dataframe(settings, abbreviate_name=True)
```

This concludes the first tutorial on how to run a single simulation.

```
[ ]:
```

3.2.2 Run several simulations

One advantage of **pyMonteCarlo** is the possibility to run several simulations. In this example, we will try to study the influence of the beam energy and the thickness of the carbon coating on the k-ratios of O Kα and Al Kα in orthoclase.

We start by defining the beam energies and carbon thicknesses.

```
[ ]: beam_energies_keV = [5.0, 10.0, 20.0]
     carbon_thicknesses_nm = [10, 20, 50]
```

We import the required classes of **pyMonteCarlo** and Python's built-in module, *itertools*. For this example, we will run the simulations with **Casino 2**, but this could be replaced by any support programs.

```
[ ]: import math
     import itertools

     from pymontecarlo_casino2.program import Casino2Program
     from pymontecarlo.options.beam import PencilBeam
     from pymontecarlo.options.material import Material
     from pymontecarlo.options.sample import HorizontalLayerSample
     from pymontecarlo.options.detector import PhotonDetector
     from pymontecarlo.options.analysis import KRatioAnalysis
     from pymontecarlo.options import Options
     from pymontecarlo.runner.helper import run_async
```

We create the materials.

```
[ ]: material_orthoclase = Material.from_formula('KAlSi3O8')
     material_carbon = Material.pure(6)
```

Using `itertools.product(...)`, we create options for every combination of beam energy and carbon thickness and store them in a list.

```
[ ]: list_options = []
     for beam_energy_keV, carbon_thickness_nm in itertools.product(beam_energies_keV,␣
     ↪carbon_thicknesses_nm):
         program = Casino2Program(number_trajectories=5000)

         beam = PencilBeam(beam_energy_keV * 1e3) # Convert to eV

         sample = HorizontalLayerSample(material_orthoclase)
         sample.add_layer(material_carbon, carbon_thickness_nm * 1e-9) # Convert thickness␣
     ↪to meters

         detector = PhotonDetector(name='detector1', elevation_rad=math.radians(40))

         analysis = KRatioAnalysis(detector)

         options = Options(program, beam, sample, [analysis])
         list_options.append(options)
```

With 3 beam energies and 3 carbon thicknesses, we get 9 options.

```
[ ]: len(list_options)
```

We can now run these simulations in parallel. By default, the number of simulations that will be run concurrently depends on the number of CPUs. This can be also modified by the argument, `max_workers` of the `run_async(...)` function.

```
[ ]: project = await run_async(list_options)
```

We first checks the number of simulations. We should have 9 simulations from the 9 options provided, but also one simulation for each standard used to calculate the k-ratios. There are 4 elements in orthoclase and one element in the coating. The same standard can be reused for the different carbon coating thicknesses, but not for the different beam energies. So the expected number of simulations should be: `9 + 3 * (4 + 1) = 24`. This example shows that **pyMonteCarlo** is aware which additional simulations it needs to calculate the k-ratios and only simulate these once.

```
[ ]: len(project.simulations)
```

Now to the analysis of the results. There are several ways to extract results from the simulations, but perhaps the easiest one is to convert all the results into a **pandas** `DataFrame`. To reduce the number of columns of the `DataFrame`, it is useful to pass the result class, `KRatioReslt` in this case, and request only the columns with different information.

```
[ ]: from pymontecarlo.results import KRatioResult
     from pymontecarlo.settings import Settings, XrayNotation

     settings = Settings()
     settings.set_preferred_unit('keV')
     settings.set_preferred_unit('nm')
     settings.set_preferred_unit('deg')
     settings.set_preferred_unit('g/cm^3')
     settings.preferred_xray_notation = XrayNotation.SIEGBAHN

     df = project.create_dataframe(settings, result_classes=[KRatioResult], only_different_
     ↪columns=True)
     df
```

The **pandas** `DataFrame` above contains one row for each simulation, including the standards. Since we are only interested in the k-ratios of the *unknowns*, we can filter the `DataFrame` and drop some columns.

```
[ ]: # Keep only standard rows
     df = df[df['standard'] != True].dropna(axis=1)

     # Remove columns which all contain the same values
     df = df[[col for col in df if df[col].nunique() != 1]]
     df
```

We can then use this `DataFrame` to plot the results using **matplotlib**.

```
[ ]: import matplotlib.pyplot as plt

     xraylines = ['O Kα', 'Al Kα']

     fig, axes = plt.subplots(1, len(xraylines), figsize=(5 * len(xraylines), 4))
     fig.subplots_adjust(wspace=0.3)

     for ax, xrayline in zip(axes, xraylines):
         for beam_energy_keV, df_beam_energy in df.groupby('beam energy [keV]'):
             ax.plot(df_beam_energy['layer #0 thickness [nm]'], df_beam_energy[xrayline],
     ↪'o-', label=f'E0={beam_energy_keV:.0f} keV')

         ax.set_xlabel('Carbon coating thickness (nm)')
         ax.set_ylabel(f'{xrayline} k-ratio')

     axes[0].legend(loc='best')

     plt.show()
```

```
[ ]:
```

3.3 Supported Monte Carlo programs

Here are the currently supported Monte Carlo programs in **pyMonteCarlo**.

3.3.1 Casino 2

Authors Dominique Drouin, Alexandre Real Couture, Raynald Gauvin, Pierre Hovington, Paula Horny, Hendrix Demers, Dany Joly, Philippe Drouin and Nicolas Poirier-Demers

Version 2.5.1.0 (2017)

Website http://www.gel.usherbrooke.ca/casino

Supported platforms

- Windows
- MacOS (using *Wine*, see *installation*)
- Linux (using *Wine*, see *installation*)

Python package dependencies

- pycasinotools (code | documentation)
- pymontecarlo-casino2

Installation *Casino 2* is distributed inside *pymontecarlo-casino2*. No installation step is needed.

3.4 Supported options

Here are the options supported by each Monte Carlo program.

3.4.1 Beam

	Cylindrical	Gaussian	Pencil
Casino 2		x	x

3.4.2 Sample

	Horizontal layer	Inclusion	Sphere	Substrate	Vertical layer
Casino 2	x			x	x

3.4.3 Analysis

	K ratio	Photon intensity
Casino 2	x	x

3.5 Examples

Here are examples on how to use pyMonteCarlo.

3.6 Code API

CHAPTER 4

Indices and tables

- genindex
- modindex
- search