PyPartMC: A Pythonic interface enhancing Fortran-based simulation package

Gracjan Adamus & the PyPartMC team*,**
* Rafał Oszajca, Sylwester Arabas (AGH University in Kraków)
** Jeff Curtis, Zach D'Aquino, Matthew West, Nicole Riemer (University of Illinois Urbana Champaign)
Faculty of Physics and Applied Computer Science, AGH University of Kraków

Uniform API across 4 Languages

Python

```python
import numpy as np

import PyPartMC as ppmc
from PyPartMC import si

aero_data = ppmc.AeroData((
# [density, ions in solution, molecular weight, kappa, abifm_m,
abifm_c]
    {"OC": [1000 *si.kg/si.m**3, 0, 1e-3 *si.kg/si.mol, 0.001, 0, 0]},
    {"BC": [1800 *si.kg/si.m**3, 0, 1e-3 *si.kg/si.mol, 0, 0 , 0]},
))

aero_dist = ppmc.AeroDist(
    aero_data,
    [{
        "cooking": {
            "mass_frac": [{"OC": [1]}],
            "diam_type": "geometric",
            "mode_type": "log_normal",
            "num_conc": 3200 / si.cm**3,
            "geom_mean_diam": 8.64 * si.nm,
            "log10_geom_std_dev": 0.28,
        },
        "diesel": {
            "mass_frac": [{"OC": [0.3]}, {"BC": [0.7]}],
            "diam_type": "geometric",
            "mode_type": "log_normal",
            "num_conc": 2900 / si.cm**3,
            "geom_mean_diam": 50 * si.nm,
            "log10_geom_std_dev": 0.24,
        }
    }],
)

n_part = 100
aero_state = ppmc.AeroState(aero_data, n_part, "nummass_source")
aero_state.dist_sample(aero_dist)
print(np.dot(aero_state.masses(), aero_state.num_concs), "# kg/m3")
```

Julia

```julia
using Pkg
Pkg.add("PyCall")

using PyCall
ppmc = pyimport("PyPartMC")
si = ppmc["si"]

aero_data = ppmc.AeroData((
# (density, ions in solution, molecular weight, kappa, abifm_m,
abifm_c)
  Dict("OC"=>(1000 * si.kg/si.m^3, 0, 1e-3 * si.kg/si.mol, 0.001, 0, 0)),
  Dict("BC"=>(1800 * si.kg/si.m^3, 0, 1e-3 * si.kg/si.mol, 0, 0, 0))
))

aero_dist = ppmc.AeroDist(aero_data, (
  Dict(
    "cooking" => Dict(
      "mass_frac" => (Dict("OC" => (1,)),),
      "diam_type" => "geometric",
      "mode_type" => "log_normal",
      "num_conc" => 3200 / si.cm^3,
      "geom_mean_diam" => 8.64 * si.nm,
      "log10_geom_std_dev" => .28,
    ),
    "diesel" => Dict(
      "mass_frac" => (Dict("OC" => (.3,)), Dict("BC" => (.7,))),
      "diam_type" => "geometric",
      "mode_type" => "log_normal",
      "num_conc" => 2900 / si.cm^3,
      "geom_mean_diam" => 50 * si.nm,
      "log10_geom_std_dev" => .24,
    )
  ),
))

n_part = 100
aero_state = ppmc.AeroState(aero_data, n_part, "nummass_source")
aero_state.dist_sample(aero_dist)
print(aero_state.masses()'aero_state.num_concs, "# kg/m3")
```

Matlab

```matlab
ppmc = py.importlib.import_module('PyPartMC');
si = py.importlib.import_module('PyPartMC').si;

aero_data = ppmc.AeroData(py.tuple({ ...
  py.dict(pyargs("OC", py.tuple({1000 * si.kg/si.m^3, 0, 1e-3 *
  si.kg/si.mol, 0.001, 0, 0}))), ...
  py.dict(pyargs("BC", py.tuple({1800 * si.kg/si.m^3, 0, 1e-3 *
  si.kg/si.mol, 0, 0, 0}))) ...
}));

aero_dist = ppmc.AeroDist(aero_data, py.tuple({ ...
  py.dict(pyargs( ...
    "cooking", py.dict(pyargs( ...
      "mass_frac", py.tuple({py.dict(pyargs("OC", py.tuple({1})))}), ...
      "diam_type", "geometric", ...
      "mode_type", "log_normal", ...
      "num_conc", 3200 / si.cm^3, ...
      "geom_mean_diam", 8.64 * si.nm, ...
      "log10_geom_std_dev", .28 ...
    )), ...
    "diesel", py.dict(pyargs( ...
      "mass_frac", py.tuple({ ...
        py.dict(pyargs("OC", py.tuple({.3}))), ...
        py.dict(pyargs("BC", py.tuple({.7}))), ...
      }), ...
      "diam_type", "geometric", ...
      "mode_type", "log_normal", ...
      "num_conc", 2900 / si.cm^3, ...
      "geom_mean_diam", 50 * si.nm, ...
      "log10_geom_std_dev", .24 ...
    )) ...
  )) ...
}));

n_part = 100;
aero_state = ppmc.AeroState(aero_data, n_part, "nummass_source");
aero_state.dist_sample(aero_dist);
masses = cell(aero_state.masses());
num_concs = cell(aero_state.num_concs);
fprintf('%g # kg/m3\n', dot([masses{:}], [num_concs{:}]))
```

C++

```cpp
#include "PyPartMC.hpp"
#include "PyPartMC/si.hpp"

auto aero_data = std::make_shared<AeroData>(json({
    { {"OC", {1000.0 * si::kg / (si::m * si::m * si::m), 0.0, 1e-3 *
    si::kg / si::mol, 0.001, 0.0, 0.0}} },
    { {"BC", {1800.0 * si::kg / (si::m * si::m * si::m), 0.0, 1e-3 *
    si::kg / si::mol, 0.0,   0.0, 0.0}} }
}));

auto aero_dist = AeroDist(
    aero_data,
    ordered_json::array({
        ordered_json::object({
            {"cooking", {
                {"mass_frac", ordered_json::array({
                ordered_json::object({{"OC", {1.0}}}) }) },
                {"diam_type", "geometric"},
                {"mode_type", "log_normal"},
                {"num_conc", 3200.0 / (si::cm * si::cm * si::cm)},
                {"geom_mean_diam", 8.64 * si::nm},
                {"log10_geom_std_dev", 0.28}
            }},
            {"diesel", {
                {"mass_frac", ordered_json::array({
                ordered_json::object({{"OC", {0.3}}}),
                ordered_json::object({{"BC", {0.7}}}) }) },
                {"diam_type", "geometric"},
                {"mode_type", "log_normal"},
                {"num_conc", 2900.0 / (si::cm * si::cm * si::cm)},
                {"geom_mean_diam", 50.0 * si::nm},
                {"log10_geom_std_dev", 0.24}
            }}
        })
    })
);

int n_part = 100;
auto aero_s = AeroState(aero_data, n_part, "nummass_source");
AeroState::dist_sample(aero_s, aero_dist, 1.0, 0.0, true, true);
auto masses = AeroState::masses(aero_s, {}, {});
auto num_concs = AeroState::num_concs(aero_s);
double total_mass = std::inner_product(std::begin(num_concs),
std::end(num_concs), std::begin(masses), 0.0);
std::cout << std::scientific << total_mass << " # kg/m3\n";
```

What are PyPartMC and PartMC?

PartMC is a particle-resolved Monte Carlo model for atmospheric aerosol dynamics, providing highly detailed simulations of aerosol processes. It is being developed at **University of Illinois Urbana-Champaign** for more than 20 years now. PartMC works as an executable binary, completely controlled by outside configuration text files. https://doi.org/10.1080/02786826.2017.1311988

PyPartMC is a modern, Pythonic interface to PartMC. It enables researchers to leverage the scientific Python ecosystem (NumPy, SciPy, pandas, Jupyter) for setup, execution, and analysis of PartMC simulations, drastically improving accessibility and workflow efficiency. The package is being developed by teams at **University of Illinois Urbana-Champaign** and **AGH University of Kraków**. Contrary to PartMC, it serves as a library for building your own software. It is callable from **Python**, **C++**, and also **Julia** and **Matlab** through their respective Python bridges.

Fortran ISO C Bindings

```fortran
function sum_array(arr, n) result(total)
bind(C, name="sum_array")
    use iso_c_binding
    implicit none

    integer(c_int), intent(in), value :: n
    real(c_double), intent(in) :: arr(*)

    real(c_double) :: total
    integer(c_int) :: i

    total = 0.0_c_double
    do i = 1, n
        total = total + arr(i)
    end do
end function sum_array
```

Listing 1. Fortran function exposed to C

```cpp
extern "C"
double sum_array(const double *arr, int n);

int main() {
    std::vector<double> vec = {1.5, 2.5, 3.0,
    4.0, 5.0};

    double result = sum_array(vec.data(),
    vec.size());

    return 0;
}
```

Listing 2. Calling the Fortran function from C++

High-performance computing frequently demands coupling modern applications with highly optimized Fortran numerical libraries. The **Fortran 2003 ISO C Binding** standardizes this integration, replacing fragile, compiler-specific name-mangling hacks with robust, portable interfaces.

By leveraging the `bind(C)` attribute and intrinsic `C_types`, developers can achieve zero-overhead cross-language function calls and direct memory sharing.

Additionally, with the simple use of the `extern` keyword, we can extend the function's usability to C++.

nanobind

```cpp
#include <nanobind/nanobind.h>
namespace nb = nanobind;

int add(int a, int b) {
    return a + b;
}

struct Dog {
    std::string name;

    std::string bark() const {
        return name + ": woof!";
    }
};

NB_MODULE(ExampleModule, m) {
    m.doc() = "This is a \"Hello World\"
    example with nanobind";

    m.def("add", add, "Add two integers");

    nb::class_<Dog>(m, "Dog", "Barking
    canine")
        .def(nb::init<const std::string &>())
        .def_ro("name", &Dog::name)
        .def("bark", &Dog::bark);
}
```

Listing 3. "Hello World" example in `nanobind`

nanobind is a library used to create C++ to Python bindings. It is a successor to **pybind11**, created by the same author.

The library works as a very convenient and efficient wrapper over the standard Python C API (*Python.h*), with a very minimal boilerplate.

nanobind also enables you to write your own **type casters**. We have used that to create casters for `std::valarray` and `tl::optional` to avoid rewriting significant portions of the codebase.

nanobind_json

```cpp
#include <nlohmann/json.hpp>
#include <nanobind_json/nanobind_json.h>

void take_json(const nlohmann::json &json) {
    std::cout << "This function took an
    nlohmann::json instance as argument: " << json
    << std::endl;
}

nlohmann::json return_json() {
    nlohmann::json j = {{"value", 1}};
    std::cout << "This function returns an
    nlohmann::json instance: "  << j << std::endl;
    return j;
}

NB_MODULE(my_module, m) {
    m.doc() = "My awesome module";

    m.def("take_json", &take_json, "Pass a Python
    object to a C++ function that takes an
    nlohmann::json");
    m.def("return_json", &return_json, "Return a
    nb::object from a C++ function that returns a
    nlohmann::json");
}
```

Listing 4. Example usage of the `nanobind_json` library. After including the header the casting is performed automatically

nanobind FAQ

Q: Is there a way to pass JSON objects between Python and C++?

A: Yes, an unofficial, currently maintained, package supporting that can be found here. It is based on a similar package for pybind11 called pybind11_json.

`PyPartMC` uses `JSONs` internally. Thanks to them we are able to provide a flexible and uniform interface, without any auxillary files (like "`spec files`" in `PartMC`).

`JSON` operability is not supported in `nanobind`. We found the `nanobind_json` library, but it was unmaintained and broken. We decided to fix it and become its current maintainers. The library is now mentioned in the official `nanobind` documentation!

Additionally, for the purposes of our package, support for `ordered_json` was also added.

Project Architecture

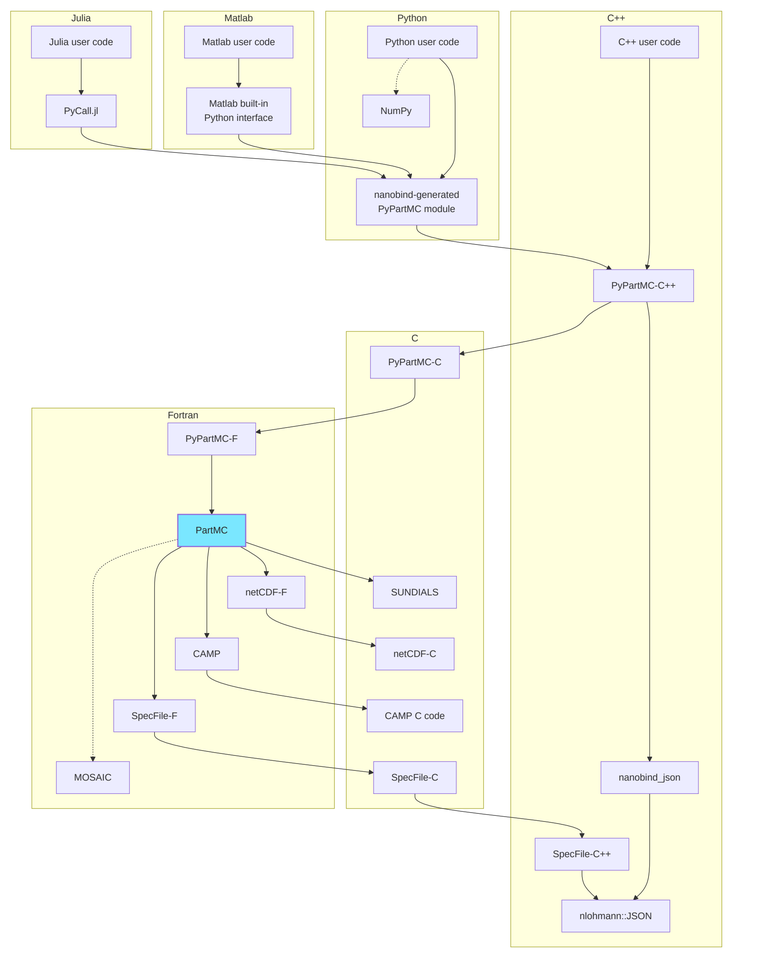

Figure 1. High-level overview of the library architecture showing Python, C++ bindings, and Fortran core

cibuildwheel

```yaml
- name: Install cibuildwheel
  run: python -m pip install cibuildwheel

- name: Build and test wheels
  env:
    # skip 32-bit, musllinux, and
    free-threaded builds
    CIBW_SKIP: "*-win32 *-manylinux_i686
    *musllinux* cp3??t-*"
    CIBW_BEFORE_BUILD_MACOS: brew reinstall
    gcc; pip install --only-binary=llvmlite
    llvmlite
    CIBW_TEST_REQUIRES: pytest pytest-order
    CIBW_TEST_COMMAND: pytest -v -s -We -p
    no:unraisableexception {package}/tests
  run: python -m cibuildwheel --output-dir
  dist
```

Listing 5. Snippet of the `PyPartMC` CI workflow for building and uploading the Python wheels

To use the library, you only need to do one `pip install` call. All dependencies are statically linked in the package. Python wheels are then built with the use of **`cibuildwheel`** tool from **PyPA**.

The whole process is performed on the `Github Actions` CI. The library is compiled into a wheel and tested across a matrix of Python versions and OSes all with the help of `cibuildwheel`. The package is then automatically uploaded to `PyPI`.

Garbage Collection with RAII

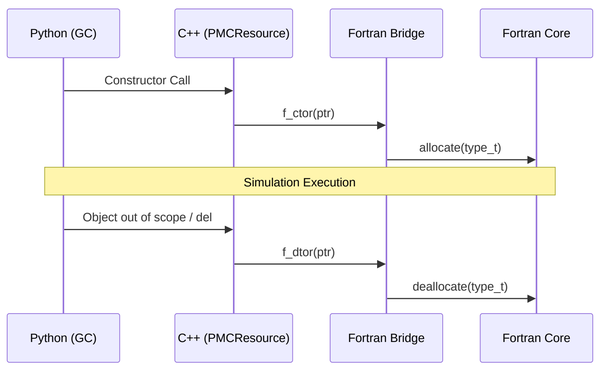

Figure 2: C++ RAII wrappers safely manage the lifecycle of Fortran heap objects.

`PyPartMC` objects live across different languages. C++ and Fortran do not have a garbage collector like Python. To maintain memory safety, we use a thin RAII wrapper on the C++ side (`PMCResource`) to maintain a garbage-collector-like behavior.

`PMCResource` wraps every Fortran object pointer. It also holds pointers to a "constructor" and "destructor" functions of said object. When Python GC deletes an object, nanobind will automatically call the underlying C++ object's destructor, which in turn will call the saved dtor function of the Fortran object.

Maintaining Submodules

Figure 3. Dependabot logo

`PyPartMC` makes use of a lot dependencies, all of which are part of the repository through the `Git submodule` mechanism. To ensure maintainability, we use Dependabot to automatically track and update submodules, guaranteeing that our Fortran, C and C++ dependencies stay secure and up-to-date without manual intervention. In case of a new version of a submodule, Dependabot will create a PR bumping the version.