Engineering Fortran-to-Python Bindings in C++ with nanobind[_json] and cibuildwheel

Gracjan Adamus @ EuroSciPy 2026

Jeff Curtis
Zach D'Aquino
Nicole Riemer
Matthew West

Gracjan Adamus
Rafał Oszajca
Sylwester Arabas

#whoami

- Third-year Applied Computer Science student @ AGH

#whoami

- Third-year Applied Computer Science student @ AGH
- President of KNI Kernel

#whoami

- Third-year Applied Computer Science student @ AGH
- President of KNI Kernel
- 2025 Summer Intern @ Paul Scherrer Institute

#whoami

- Third-year Applied Computer Science student @ AGH
- President of KNI Kernel
- 2025 Summer Intern @ Paul Scherrer Institute
- 2026 Summer Student @ CERN

#whoami

- Third-year Applied Computer Science student @ AGH
- President of KNI Kernel
- 2025 Summer Intern @ Paul Scherrer Institute
- 2026 Summer Student @ CERN
- Previously worked as C++ Developer

#whoami

- Third-year Applied Computer Science student @ AGH
- President of KNI Kernel
- 2025 Summer Intern @ Paul Scherrer Institute
- 2026 Summer Student @ CERN
- Previously worked as C++ Developer
- Certified Mole lover

#whoami

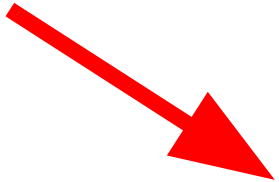
- Third-year Applied Computer Science student @ AGH
- President of KNI Kernel
- 2025 Summer Intern @ Paul Scherrer Institute
- 2026 Summer Student @ CERN
- Previously worked as C++ Developer
- Certified Mole lover
- Interested in HPC, GPUs and all the fun stuff

#whoami

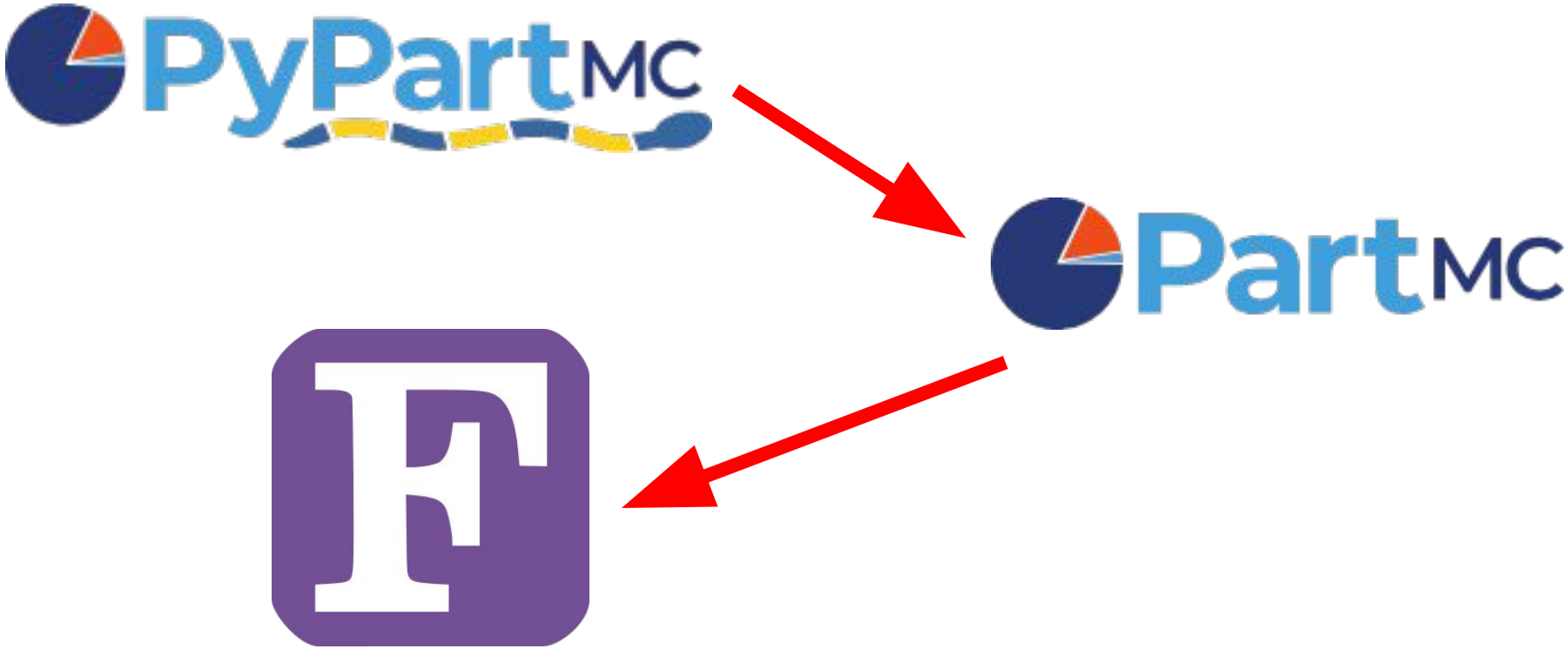
- Third-year Applied Computer Science student @ AGH
- President of KNI Kernel
- 2025 Summer Intern @ Paul Scherrer Institute
- 2026 Summer Student @ CERN
- Previously worked as C++ Developer
- Certified Mole lover
- Interested in HPC, GPUs and all the fun stuff
- Griger5 @ Github

Background

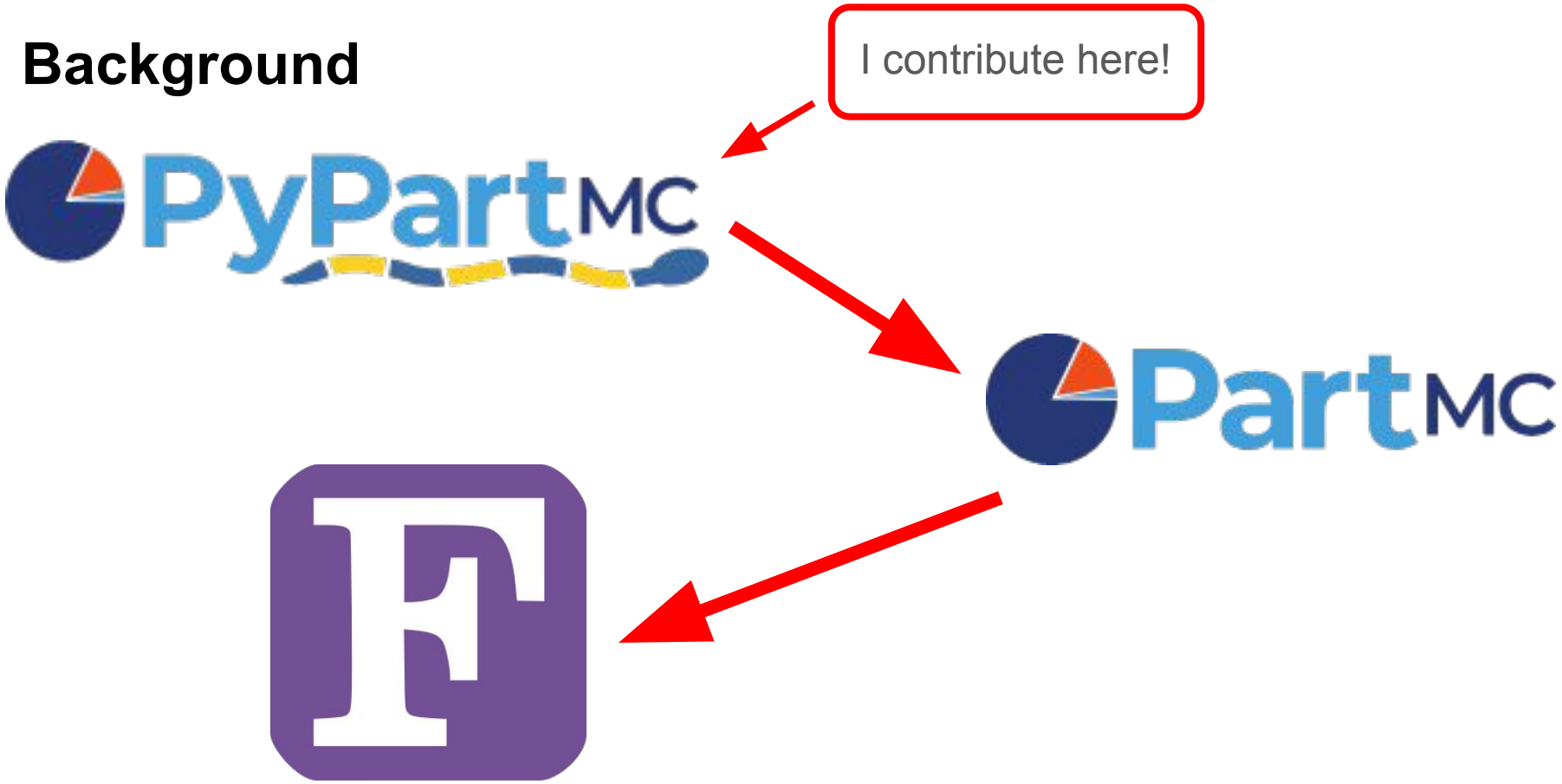
Background



Background



Background



PartMC

PartMC

- Particle-resolved Monte Carlo code for atmospheric aerosol simulation

PartMC

- Particle-resolved Monte Carlo code for atmospheric aerosol simulation
- Developed at University of Illinois Urbana-Champaign

PartMC

- Particle-resolved Monte Carlo code for atmospheric aerosol simulation
- Developed at University of Illinois Urbana-Champaign
- Developed for more than 20 years!

PartMC

- Particle-resolved Monte Carlo code for atmospheric aerosol simulation
- Developed at University of Illinois Urbana-Champaign
- Developed for more than 20 years!
- More than 4000 citations in research papers

PartMC

- Particle-resolved Monte Carlo code for atmospheric aerosol simulation
- Developed at University of Illinois Urbana-Champaign
- Developed for more than 20 years!
- More than 4000 citations in research papers
- Written in the object-oriented paradigm, in Fortran 90

PyPartMC: A maintainable Python interface

PyPartMC: A maintainable Python interface

- Developed at UIUC and AGH University of Kraków

PyPartMC: A maintainable Python interface

- Developed at UIUC and AGH University of Kraków
- We wanted a uniform interface across languages

PyPartMC: A maintainable Python interface

- Developed at UIUC and AGH University of Kraków
- We wanted a uniform interface across languages
- Using unmodified PartMC code and no shell

PyPartMC: A maintainable Python interface

- Developed at UIUC and AGH University of Kraków
- We wanted a uniform interface across languages
- Using unmodified PartMC code and no shell
- No need for auxiliary files

PyPartMC: A maintainable Python interface

- Developed at UIUC and AGH University of Kraków
- We wanted a uniform interface across languages
- Using unmodified PartMC code and no shell
- No need for auxiliary files
- Usable as a single Python script or a notebook

PyPartMC: A maintainable Python interface

- Developed at UIUC and AGH University of Kraków
- We wanted a uniform interface across languages
- Using unmodified PartMC code and no shell
- No need for auxiliary files
- Usable as a single Python script or a notebook
- **pip** installable

PyPartMC: A maintainable Python interface

- Developed at UIUC and AGH University of Kraków
- We wanted a uniform interface across languages
- Using unmodified PartMC code and no shell
- No need for auxiliary files
- Usable as a single Python script or a notebook
- **pip** installable
- Available on Linux, macOS, Windows, Linux-ARM

```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")
```

```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
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")
```

```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")
```

```javascript
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,
  py.dict(pyargs("BC", py.tuple({1800 * si.kg/si.m^3, 0, 1e-3 * si.kg/si.mol,
}));

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{:}]))
```

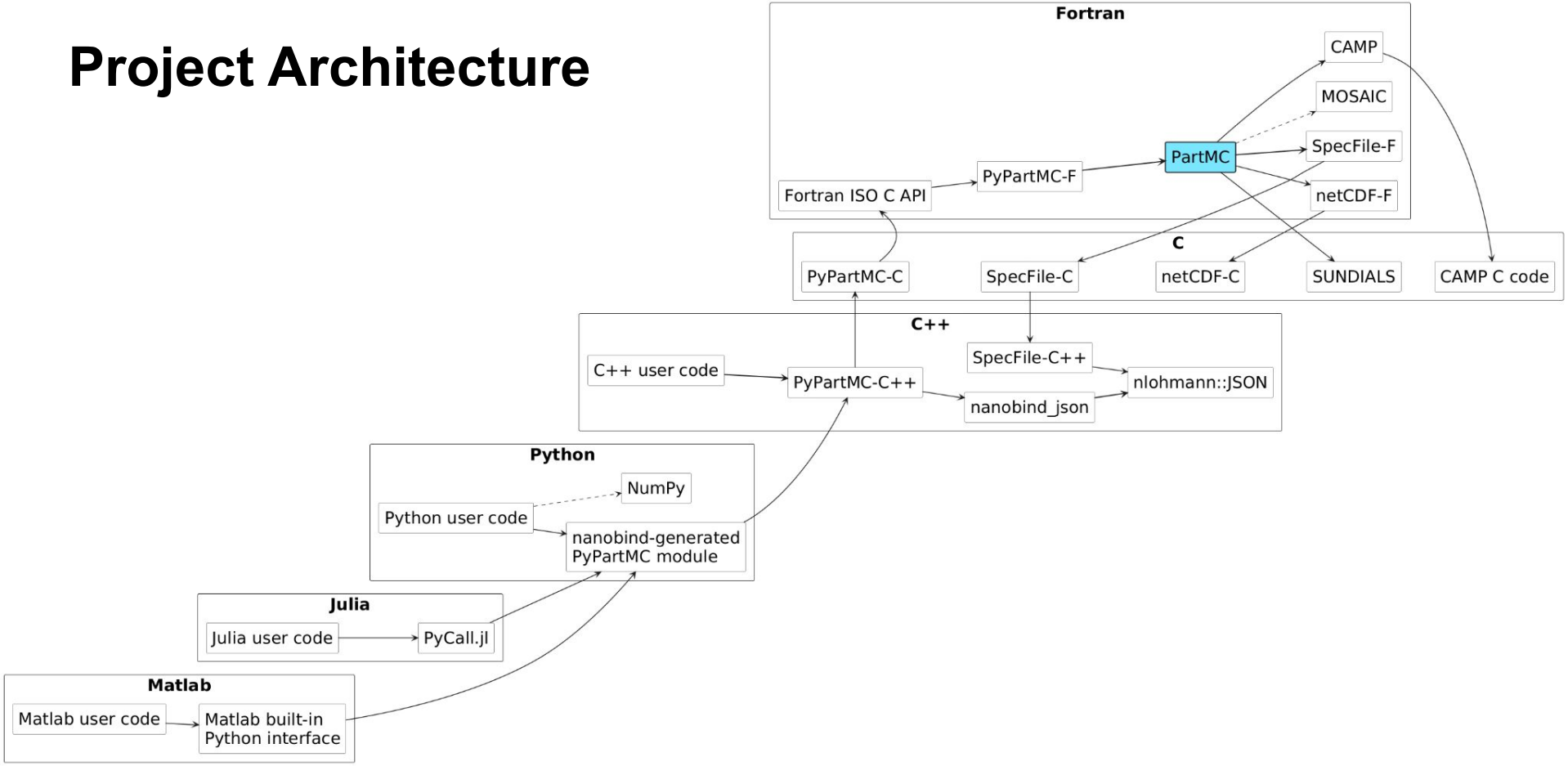
```python
import numpy as np

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")
```

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

auto aero_dist = AeroDist(
    aero_data,
    nlohmann::ordered_json::array({
        nlohmann::ordered_json::object({
            {"cooking", {
                {"mass_frac", nlohmann::ordered_json::array({ nlohmann::ordere
                {"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", nlohmann::ordered_json::array({ nlohmann::ordere
                {"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_state = AeroState(aero_data, n_part, "nummass_source");
AeroState::dist_sample(aero_state, aero_dist, 1.0, 0.0, true, true);

auto masses = AeroState::masses(aero_state, {}, {});
auto num_concs = AeroState::num_concs(aero_state);
double total_mass = std::inner_product(std::begin(num_concs), std::end(num_con

std::cout << std::scientific << total_mass << " # kg/m3\n";
```

```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")
```

```fortran
program main
  use pmc_spec_file
  use pmc_aero_data
  use pmc_aero_mode
  use pmc_aero_dist
  use pmc_aero_state

  implicit none

  type(spec_file_t) :: f_aero_data, f_aero_dist
  type(aero_data_t) :: aero_data
  type(aero_dist_t) :: aero_dist
  type(aero_state_t) :: aero_state
  integer, parameter :: n_part = 100
  integer :: n_part_add
  real(kind=dp), dimension(n_part) :: num_concs, masses

  call spec_file_open("aero_data.dat", f_aero_data)
  call spec_file_read_aero_data(f_aero_data, aero_data)
  call spec_file_close(f_aero_data)

  call spec_file_open("aero_dist.dat", f_aero_dist)
  call spec_file_read_aero_dist(f_aero_dist, aero_data, .false., aero_dist)
  call spec_file_close(f_aero_dist)

  call aero_state_zero(aero_state)
  call fractal_set_spherical(aero_data%fractal)
  call aero_state_set_weight(aero_state, aero_data, &
    AERO_STATE_WEIGHT_NUMMASS_SOURCE)
  call aero_state_set_n_part_ideal(aero_state, dble(n_part))
  call aero_state_add_aero_dist_sample(aero_state, aero_data, &
    aero_dist, 1d0, 1d0, 0d0, .true., .true., n_part_add)

  num_concs = aero_state_num_concs(aero_state, aero_data)
  masses = aero_state_masses(aero_state, aero_data)
  print *, dot_product(num_concs, masses), "# kg/m3"
end
```

Project Architecture




PartMC


PyPartMC

More than 36000 lines of code!

More than 12000 lines of code!

More than 36000 lines of code!

Over a 200 files

More than 12000 lines of code!

Over 120 files

More than 36000 lines of code!

Over a 200 files

Reported code coverage: 77%

More than 12000 lines of code!

Over 120 files

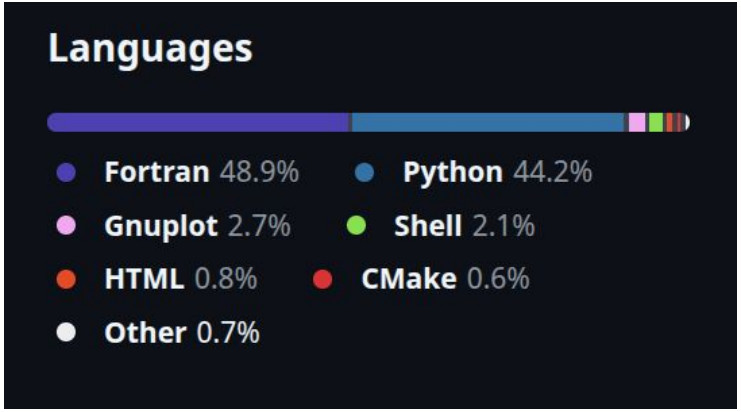
Reported code coverage: 92%

PartMC

More than 36000 lines of code!

Over a 200 files

Reported code coverage: 77%

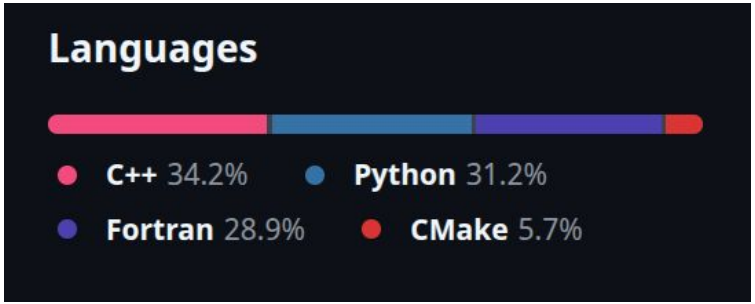


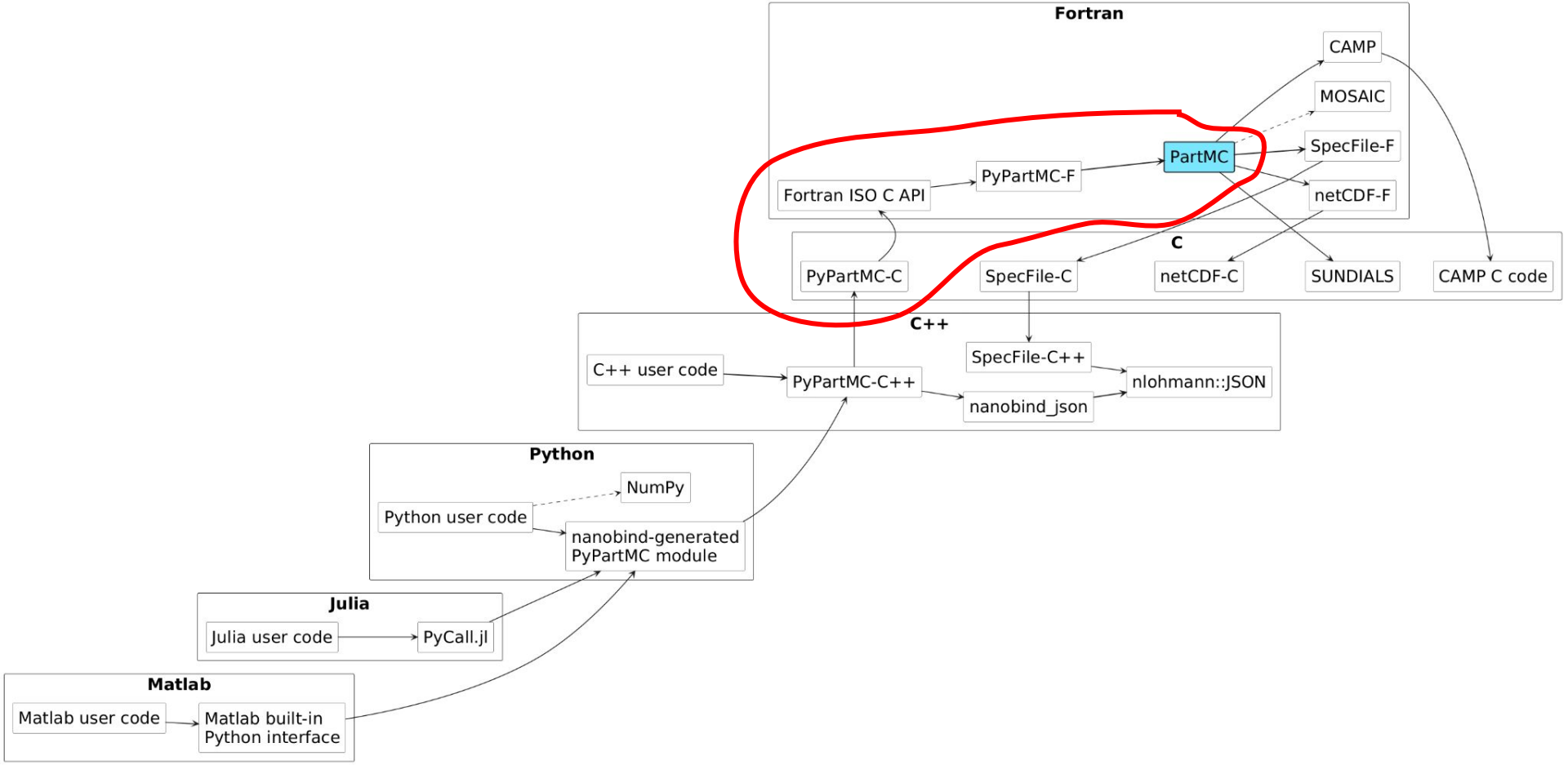
PyPartMC

More than 12000 lines of code!

Over 120 files

Reported code coverage: 92%




Fortran
CAMP
MOSAIC
SpecFile-F
PartMC
netCDF-F
Fortran ISO C API
PyPartMC-F
C
PyPartMC-C
SpecFile-C
netCDF-C
SUNDIALS
CAMP C code
C++
C++ user code
PyPartMC-C++
SpecFile-C++
nlohmann::JSON
nanobind_json
Python
NumPy
Python user code
nanobind-generated
PyPartMC module
Julia
Julia user code
PyCall.jl
Matlab
Matlab user code
Matlab built-in
Python interface

Fortran ISO C bindings

Fortran ISO C bindings

Unified compiler-agnostic interface

Fortran ISO C bindings

Unified compiler-agnostic interface

```
use iso_c_binding
```

Fortran ISO C bindings

Unified compiler-agnostic interface

```
use iso_c_binding

subroutine f_aero_data_len(ptr_c, len) bind(C)
  type(aero_data_t), pointer :: ptr_f => null()
  type(c_ptr), intent(in) :: ptr_c
  integer(c_int), intent(out) :: len

  call c_f_pointer(ptr_c, ptr_f)
  len = aero_data_n_spec(ptr_f)
end subroutine
```

Fortran ISO C bindings

Unified compiler-agnostic interface

```
use iso_c_binding

subroutine f_aero_data_len(ptr_c, len) bind(C)
  type(aero_data_t), pointer :: ptr_f => null()
  type(c_ptr), intent(in) :: ptr_c
  integer(c_int), intent(out) :: len

  call c_f_pointer(ptr_c, ptr_f)
  len = aero_data_n_spec(ptr_f)
end subroutine
```

PartMC function

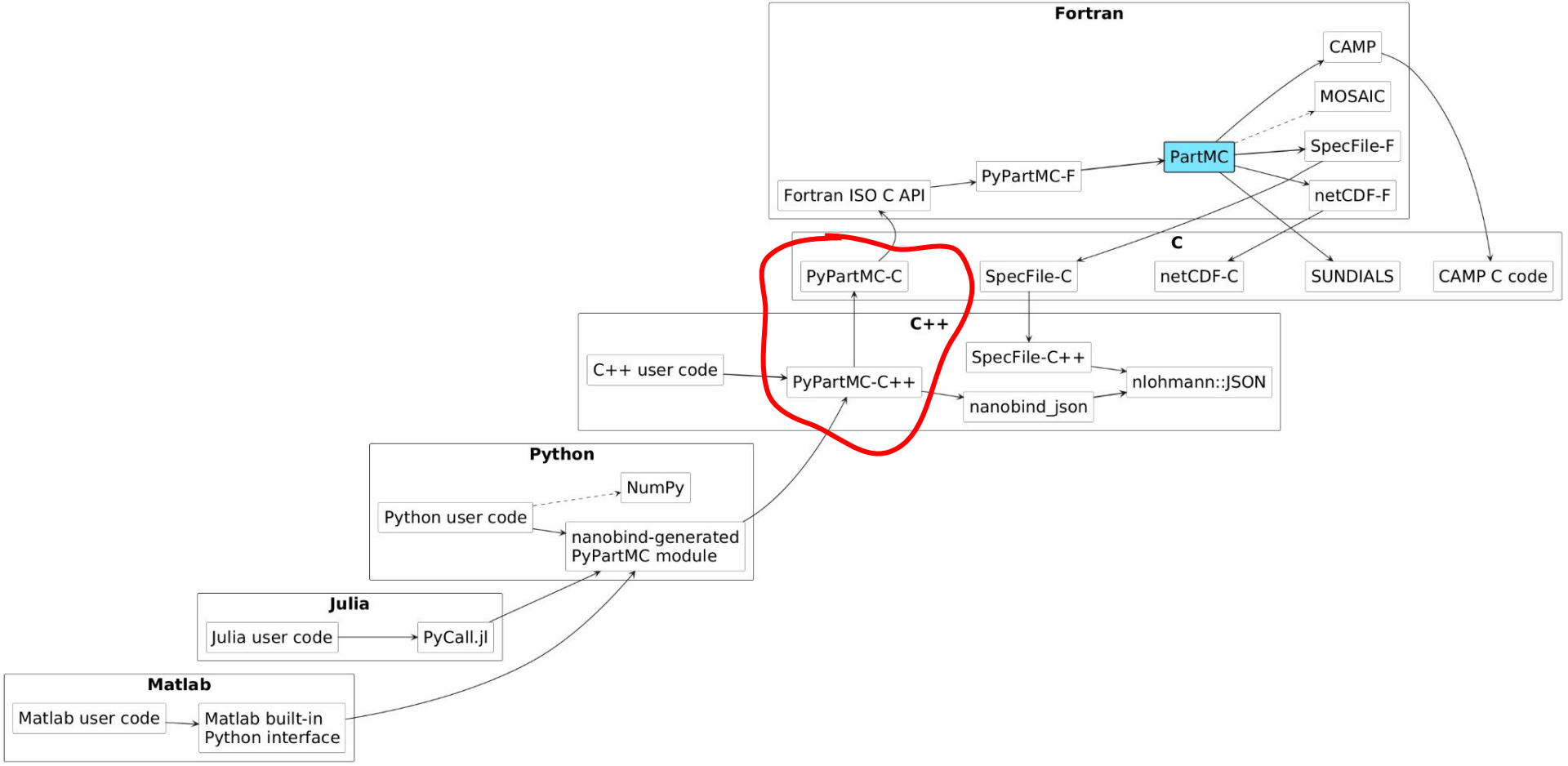
Fortran ISO C bindings

Unified compiler-agnostic interface

```
use iso_c_binding

subroutine f_aero_data_len(ptr_c, len) bind(C)
  type(aero_data_t), pointer :: ptr_f => null()
  type(c_ptr), intent(in) :: ptr_c
  integer(c_int), intent(out) :: len

  call c_f_pointer(ptr_c, ptr_f)
  len = aero_data_n_spec(ptr_f)
end subroutine
```

PartMC function

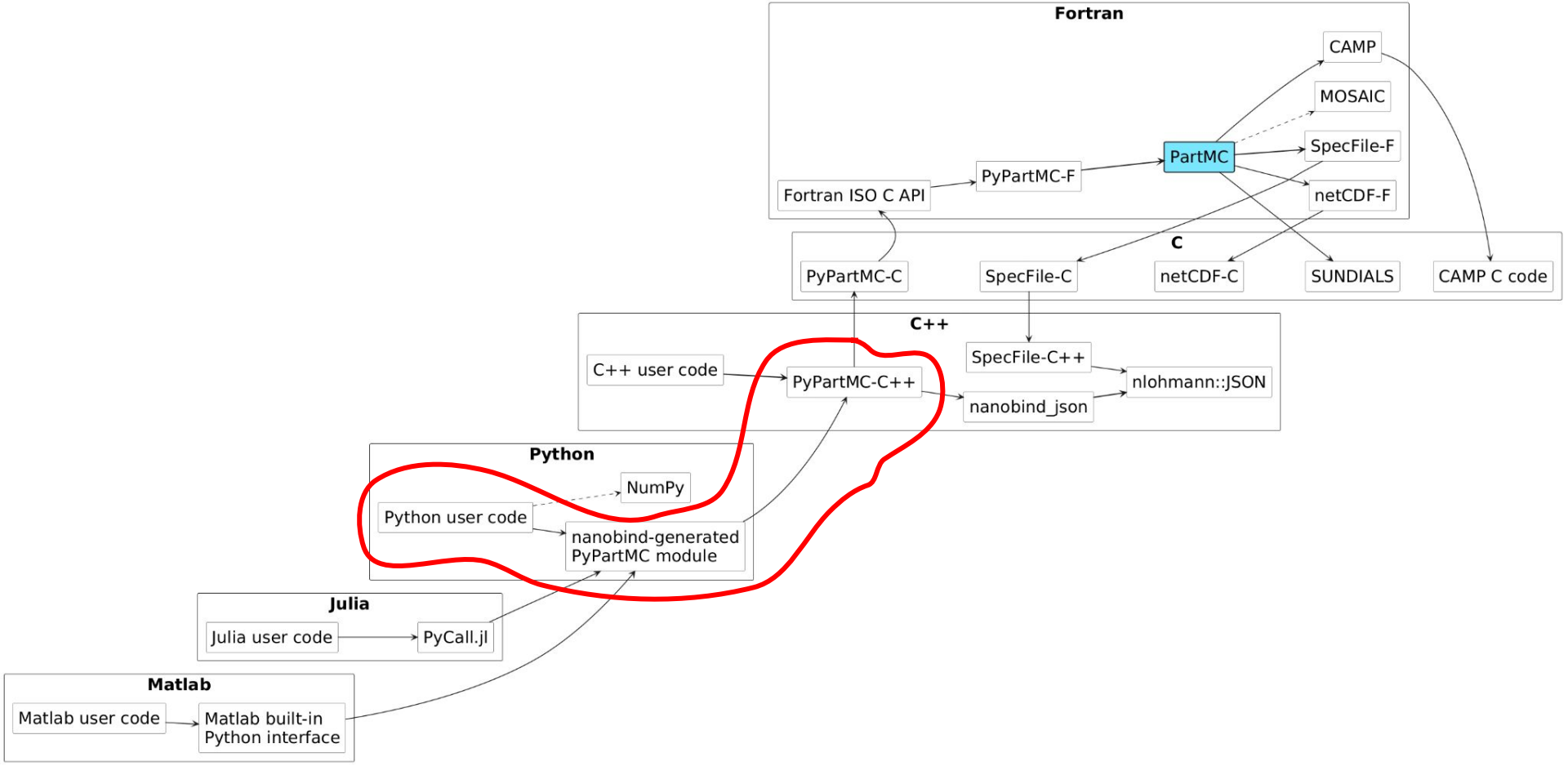
```
void f_aero_data_len(const void *ptr, int *len) noexcept;
```


Fortran
CAMP
MOSAIC
PartMC
SpecFile-F
netCDF-F
PyPartMC-F
Fortran ISO C API
C
PyPartMC-C
SpecFile-C
netCDF-C
SUNDIALS
CAMP C code
C++
C++ user code
PyPartMC-C++
SpecFile-C++
nlohmann::JSON
nanobind_json
Python
NumPy
Python user code
nanobind-generated
PyPartMC module
Julia
Julia user code
PyCall.jl
Matlab
Matlab user code
Matlab built-in
Python interface

Calling C from C++

Calling C from C++

```
extern "C"
void f_aero_data_len(
    const void *ptr,
    int *len) noexcept;
```


Fortran
CAMP
MOSAIC
SpecFile-F
PartMC
netCDF-F
PyPartMC-F
Fortran ISO C API
C
PyPartMC-C
SpecFile-C
netCDF-C
SUNDIALS
CAMP C code
C++
C++ user code
PyPartMC-C++
SpecFile-C++
nlohmann::JSON
nanobind_json
Python
NumPy
Python user code
nanobind-generated
PyPartMC module
Julia
Julia user code
PyCall.jl
Matlab
Matlab user code
Matlab built-in
Python interface

nanobind

nanobind

- Created by Wenzel Jakob, associate professor at EPFL

nanobind

- Created by Wenzel Jakob, associate professor at EPFL
- Successor to his previous project - **pybind11**

nanobind

- Created by Wenzel Jakob, associate professor at EPFL
- Successor to his previous project - **pybind11**
- Exposes C++ to Python and vice versa

nanobind

- Created by Wenzel Jakob, associate professor at EPFL
- Successor to his previous project - **pybind11**
- Exposes C++ to Python and vice versa
- A convenient wrapper over Python's C API (**Python.h**)

nanobind

- Created by Wenzel Jakob, associate professor at EPFL
- Successor to his previous project - **pybind11**
- Exposes C++ to Python and vice versa
- A convenient wrapper over Python's C API (**Python.h**)
- Minimal boilerplate

nanobind

```
NB_MODULE(_PyPartMC, m) {
    m.doc() = R"pbdoc(
        PyPartMC is a Python interface to PartMC.
    )pbdoc";

    m.def("run_part", &run_part, "Do a particle-resolved Monte Carlo simulation.");
    m.def("run_part_timestep", &run_part_timestep, "Do a single time step.");

    nb::class_<AeroDist>(m,"AeroDist")
        .def(nb::init<std::shared_ptr<AeroData>, const nlohmann::ordered_json&>())
        .def_prop_ro("n_mode", &AeroDist::get_n_mode,
            "Number of aerosol modes in the distribution.")
        .def_prop_ro("num_conc", &AeroDist::get_total_num_conc,
            "Total number concentration of a distribution (#/m^3).")
        .def("mode", AeroDist::get_mode,
            "Return the mode of a given index.")
    ;
```

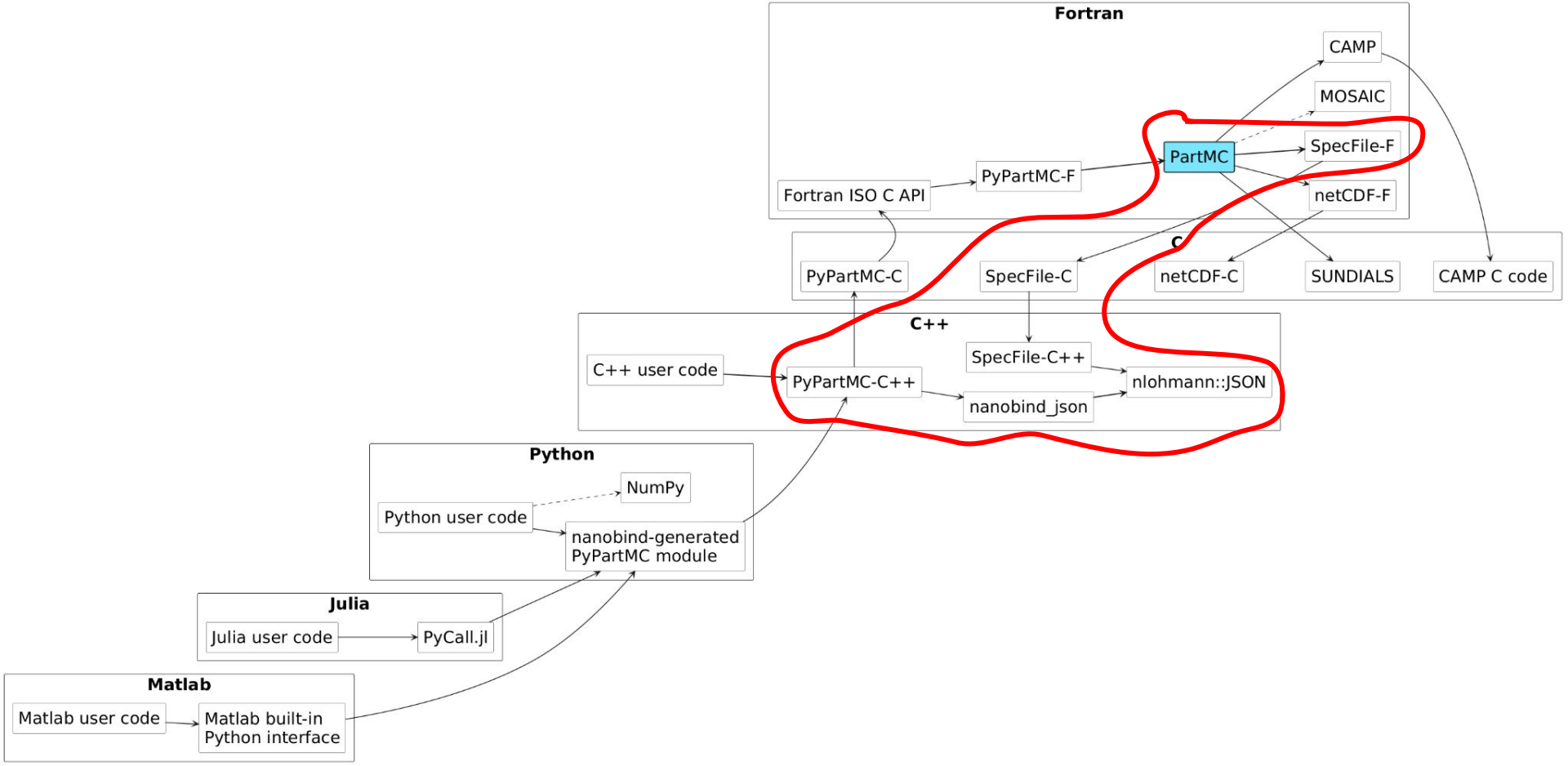
```python
aero_data = ppmc.AeroData((
# [density, ions in solution, molecular weight, kappa, abifm_m, abifm_c]
    {"OC": [1000 *si.kg/si.m**3, 0, 1e-3 *si.kg/si.mol, 0.001, 0, 0]},
    {"BC": [1800 *si.kg/si.m**3, 0, 1e-3 *si.kg/si.mol, 0, 0 , 0]},
))

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,
        }
    }],
)
```

nanobind: Custom Type Casters

nanobind: Custom Type Casters

```
template <typename Type>
struct type_caster<tl::optional<Type>> :
optional_caster<tl::optional<Type>> {};
```

```cpp
template <typename Type> struct type_caster<std::valarray<Type>> {
    NB_TYPE_CASTER(std::valarray<Type>, const_name("[") + const_name("std::valarray") + const_name("]"))

    using Caster = make_caster<Type>;

    bool from_python(handle src, uint8_t flags, cleanup_list *cleanup) noexcept {
        if (nb::isinstance<nb::list>(src)) {
            try {
                auto py_array = nb::cast<nb::list>(src);
                size_t size = py_array.size();

                value.resize(size);

                for (size_t i = 0; i < size; i++) {
                    value[i] = nb::cast<Type>(py_array[i]);
                }

                return true;
            }
            catch (...) {
                PyErr_Clear();
                return false;
            }
        }
        else if (nb::isinstance<nb::ndarray<Type, nb::ndim<1>>>(src)) {
```


Fortran
CAMP
MOSAIC
PartMC
SpecFile-F
netCDF-F
PyPartMC-F
Fortran ISO C API
C
PyPartMC-C
SpecFile-C
netCDF-C
SUNDIALS
CAMP C code
C++
C++ user code
PyPartMC-C++
SpecFile-C++
nlohmann::JSON
nanobind_json
Python
NumPy
Python user code
nanobind-generated
PyPartMC module
Julia
Julia user code
PyCall.jl
Matlab
Matlab user code
Matlab built-in
Python interface

Spec files

aero_emit_dist.dat

scenarios > 1_urban_plume > aero_emit_dist.dat

```
mode_name paved_road
mass_frac aero_emit_comp_pavedrd.dat # composition proportions of species
diam_type geometric             # type of diameter specified
mode_type log_normal            # type of distribution
num_conc 5.03E+03               # particle number concentration (#/m^2)
geom_mean_diam 1.42e-6          # geometric mean diameter (m)
log10_geom_std_dev 0.11         # log_10 of geometric std dev of diameter

mode_name cooking
mass_frac aero_emit_comp_cooking.dat # composition proportions of species
diam_type geometric             # type of diameter specified
mode_type log_normal            # type of distribution
num_conc 9.0E+06                # particle number concentration (#/m^2)
geom_mean_diam 8.64e-8          # geometric mean diameter (m)
log10_geom_std_dev 0.28         # log_10 of geometric std dev of diameter

mode_name diesel
mass_frac aero_emit_comp_diesel.dat # composition proportions of species
diam_type geometric             # type of diameter specified
mode_type log_normal            # type of distribution
num_conc 1.6E+08                # particle number concentration (#/m^2)
geom_mean_diam 5e-8             # geometric mean diameter (m)
log10_geom_std_dev 0.24         # log_10 of geometric std dev of diameter

mode_name gasoline
mass_frac aero_emit_comp_gasol.dat # composition proportions of species
diam_type geometric             # type of diameter specified
mode_type log_normal            # type of distribution
num_conc 5.00E+07               # particle number concentration (#/m^2)
geom_mean_diam 5e-8             # geometric mean diameter (m)
log10_geom_std_dev 0.24         # log_10 of geometric std dev of diameter
```

Why JSONs?

Why JSONs?

- Flexibility, the same object can operate with different mode types

Why JSONs?

- Flexibility, the same object can operate with different mode types
- Unified interface for different languages

Why JSONs?

- Flexibility, the same object can operate with different mode types
- Unified interface for different languages
- No auxiliary files - everything is included in the code

Why JSONs?

- Flexibility, the same object can operate with different mode types
- Unified interface for different languages
- No auxiliary files - everything is included in the code
- Invisible to the user - implementation detail

Why JSONs?

- Flexibility, the same object can operate with different mode types
- Unified interface for different languages
- No auxiliary files - everything is included in the code
- Invisible to the user - implementation detail

```
AERO_MODE_CTOR_LOG_NORMAL_FULL = {
    "test_mode": {
        "mass_frac": [{"SO4": [1]}],
        "diam_type": "geometric",
        "mode_type": "log_normal",
        "num_conc": 100 / si.m**3,
        "geom_mean_diam": 2 * si.um,
        "log10_geom_std_dev": np.log10(1.6),
    }
}
```

```
AERO_MODE_CTOR_EXP = {
    "test_mode": {
        "mass_frac": [{"H2O": [1]}],
        "diam_type": "geometric",
        "mode_type": "exp",
        "num_conc": 1e9 / si.m**3,
        "diam_at_mean_vol": 20 * si.um,
    }
}
```

```
AERO_MODE_CTOR_SAMPLED = {
    "test_mode": {
        "mass_frac": [{"H2O": [1]}],
        "diam_type": "geometric",
        "mode_type": "sampled",
        "size_dist": [
            {"diam": [1, 2, 3, 4]},
            {"num_conc": [100, 200, 300]},
        ],
    }
}
```

nanobind_json

nanobind_json

- We needed JSON operability for nanobind

nanobind_json

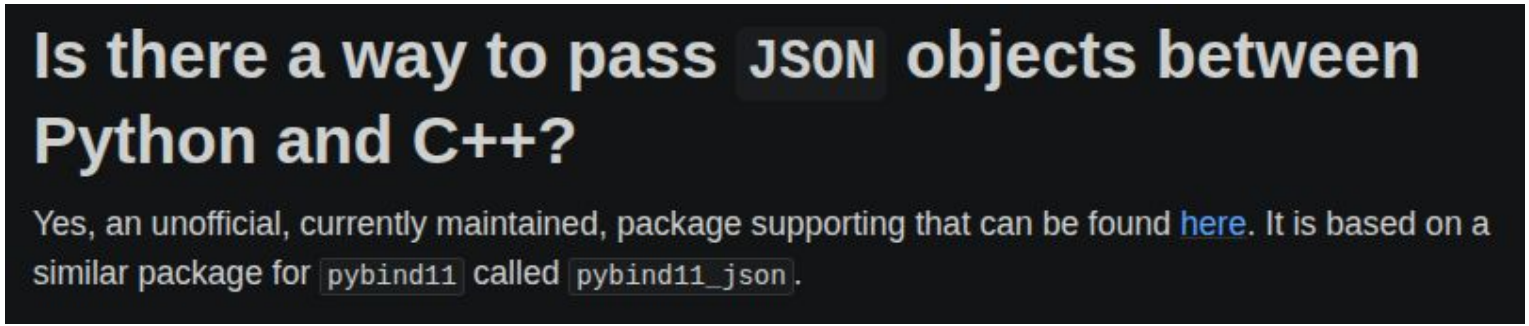
- We needed JSON operability for nanobind
- We found an unmaintained and broken nanobind_json package

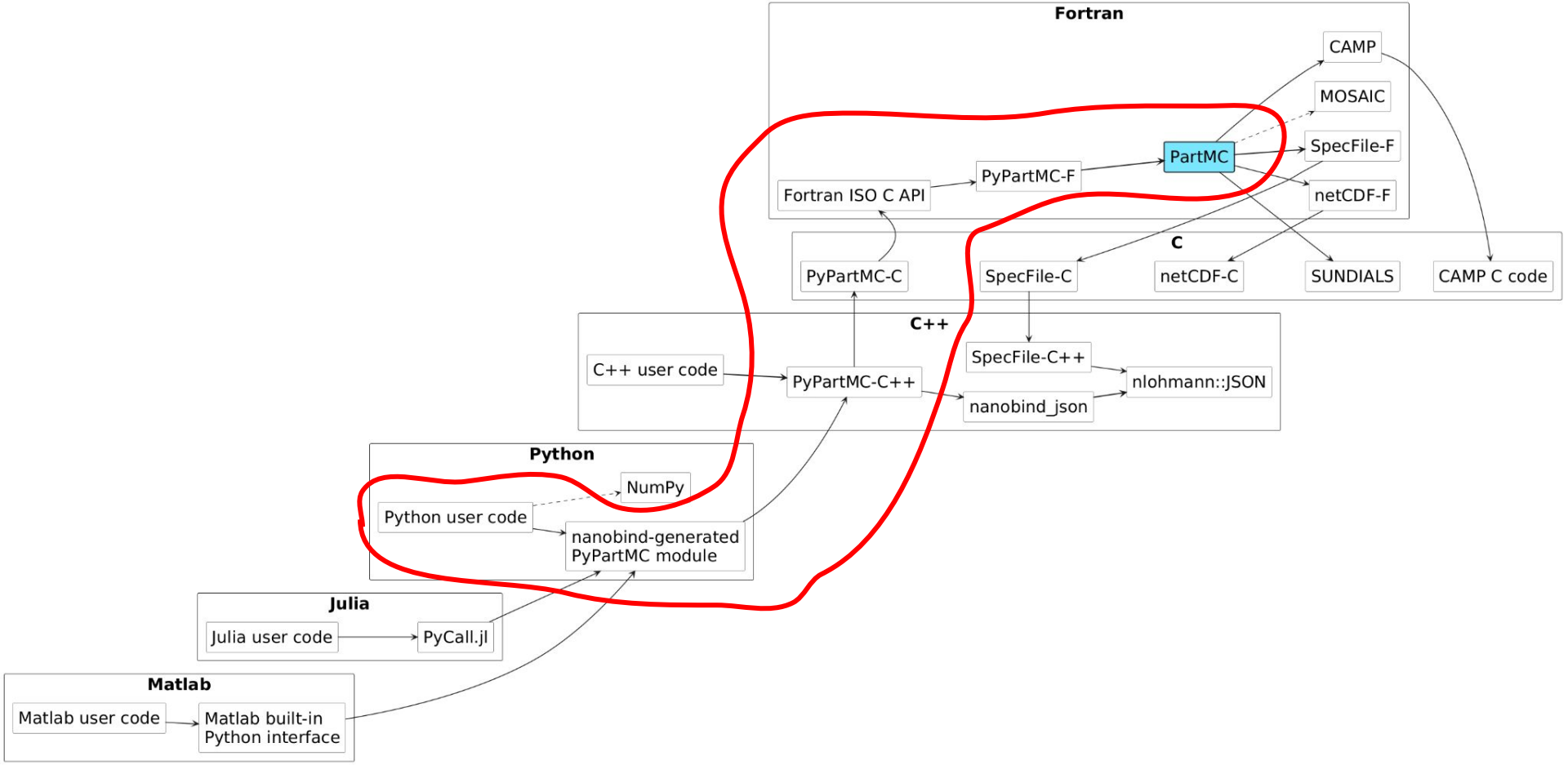
nanobind_json

- We needed JSON operability for nanobind
- We found an unmaintained and broken nanobind_json package
- So we adopted it, and I became its maintainer!

nanobind_json

- We needed JSON operability for nanobind
- We found an unmaintained and broken nanobind_json package
- So we adopted it, and I became its maintainer!




Fortran
CAMP
MOSAIC
SpecFile-F
PartMC
netCDF-F
Fortran ISO C API
PyPartMC-F
C
PyPartMC-C
SpecFile-C
netCDF-C
SUNDIALS
CAMP C code
C++
C++ user code
PyPartMC-C++
SpecFile-C++
nlohmann::JSON
nanobind_json
Python
NumPy
Python user code
nanobind-generated
PyPartMC module
Julia
Julia user code
PyCall.jl
Matlab
Matlab user code
Matlab built-in
Python interface

Code coverage for a multi-language codebase

Code coverage for a multi-language codebase

```
- run: |
    set -xe
    CMAKE_ARGS="-DCMAKE_CXX_FLAGS=--coverage \
                -DCMAKE_C_FLAGS=--coverage \
                -DCMAKE_Fortran_FLAGS=--coverage" \
    pip install --config-settings=build-dir=build -e .[tests]
    pytest -We tests
    lcov --capture --directory ./build/CMakeFiles/_PyPartMC.dir/ \
      --output-file coverage.info --no-external
```

Garbage Collection of Fortran Objects

Garbage Collection of Fortran Objects

```
subroutine f_aero_data_ctor(ptr_c) bind(C)
  type(aero_data_t), pointer :: ptr_f => null()
  type(c_ptr), intent(out) :: ptr_c

  allocate(ptr_f)
  call fractal_set_spherical(ptr_f%fractal)
  ptr_c = c_loc(ptr_f)
end subroutine

subroutine f_aero_data_dtor(ptr_c) bind(C)
  type(aero_data_t), pointer :: ptr_f => null()
  type(c_ptr), intent(in) :: ptr_c

  call c_f_pointer(ptr_c, ptr_f)
  deallocate(ptr_f)
end subroutine
```

Garbage Collection of Fortran Objects

```
extern "C" void f_aero_data_ctor(void *ptr) noexcept;
extern "C" void f_aero_data_dtor(void *ptr) noexcept;
```

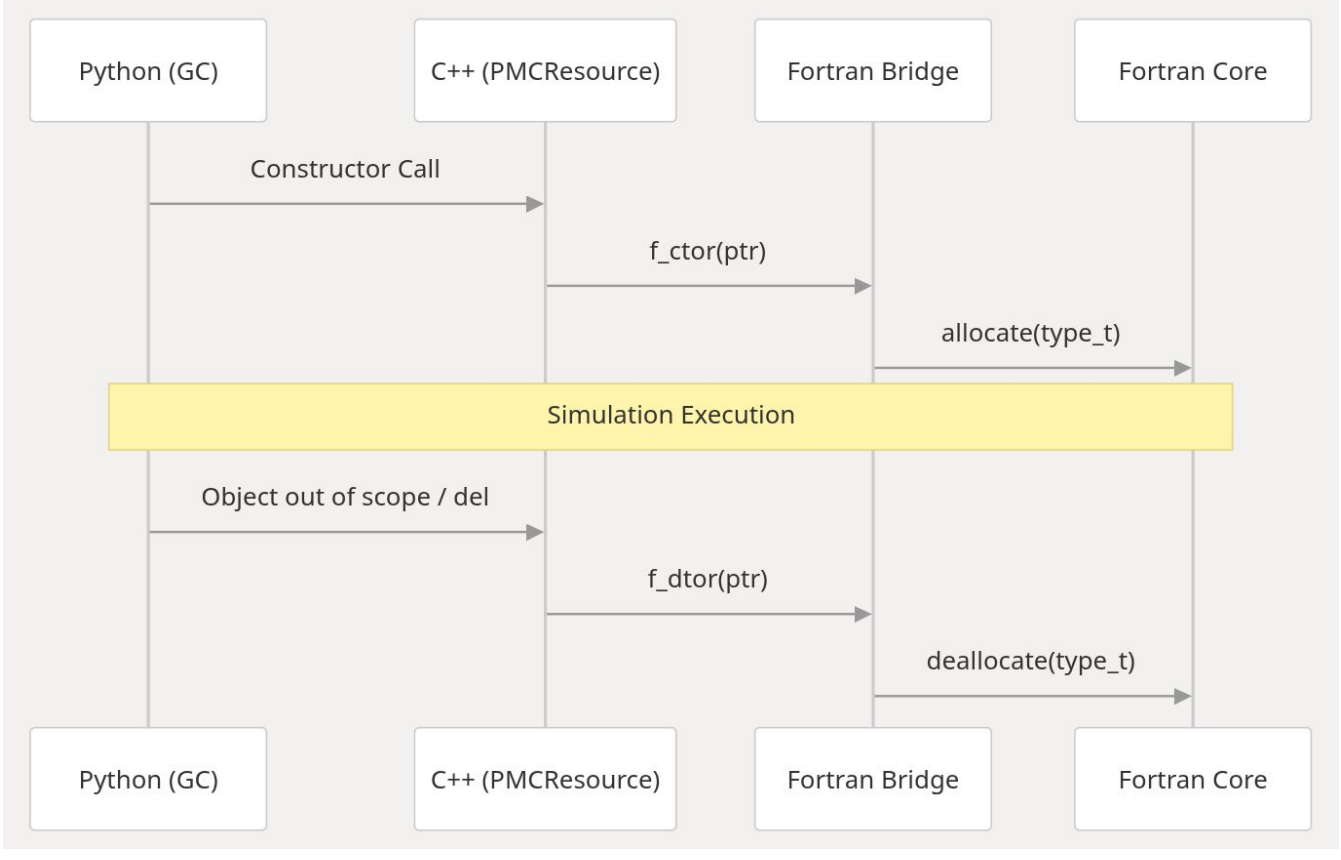
Garbage Collection of Fortran Objects

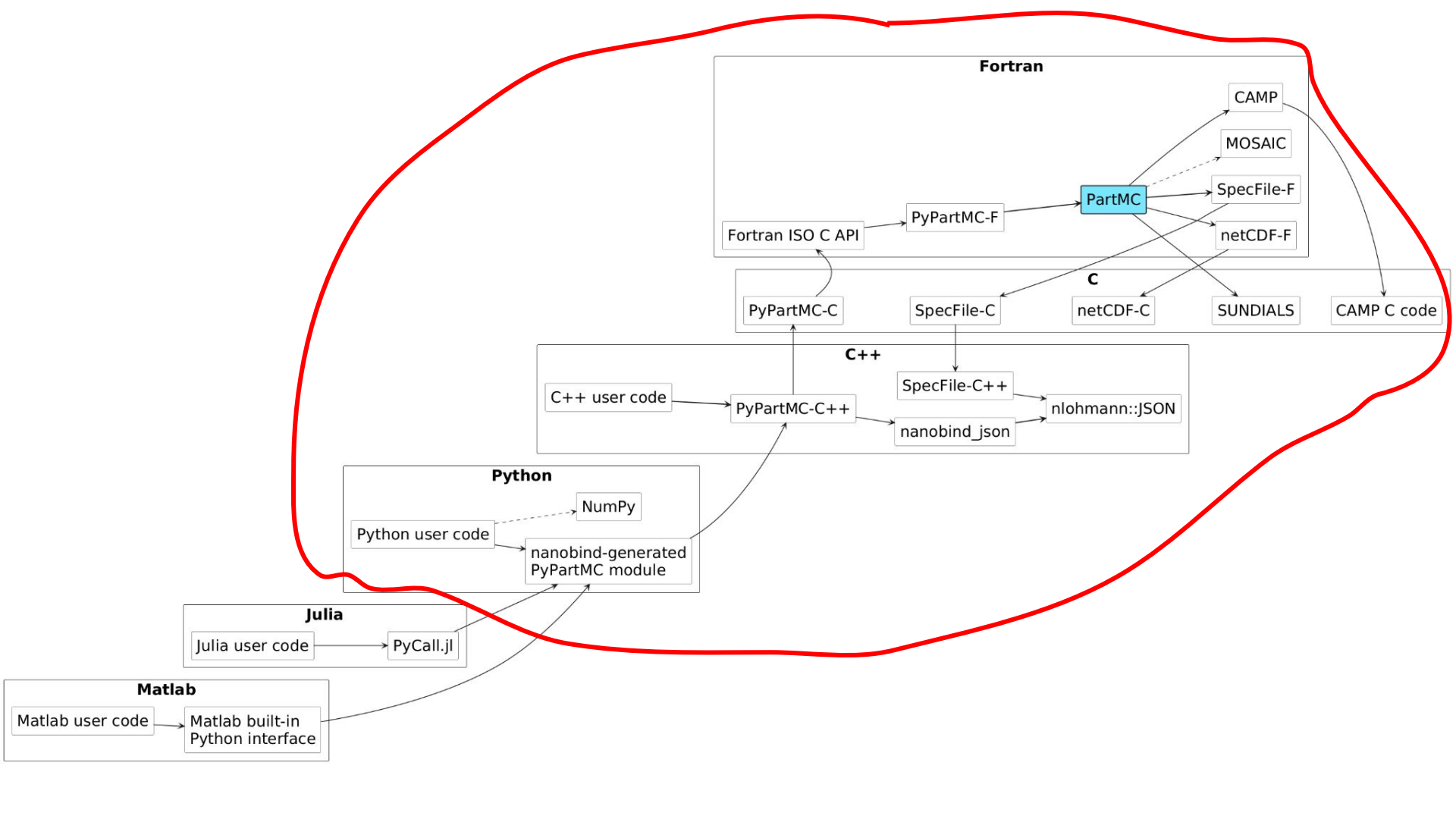
```
class PMCResource {
    void *ptr;
    void (*f_ctor)(void*);
    void (*f_dtor)(void*);

    PMCResource(const PMCResource&) = delete;
    PMCResource& operator= (const PMCResource&) = delete;

  public:
    PMCResource(
        decltype(f_ctor) f_ctor,
        decltype(f_dtor) f_dtor
    ) :
        f_ctor(f_ctor),
        f_dtor(f_dtor)
    {
        this->f_ctor(&this->ptr);
    }
    ~PMCResource() {
        this->f_dtor(&this->ptr);
    }

    const void *f_arg() const {
        return &this->ptr;
    }

    void *f_arg_non_const() {
        return &this->ptr;
    }
};
```

Garbage Collection of Fortran Objects

```
struct AeroData {
    PMCResource ptr;

    AeroData() :
        ptr(f_aero_data_ctor, f_aero_data_dtor)
    {}
```

Garbage Collection of Fortran Objects




Fortran
CAMP
MOSAIC
SpecFile-F
PartMC
netCDF-F
PyPartMC-F
Fortran ISO C API
C
PyPartMC-C
SpecFile-C
netCDF-C
SUNDIALS
CAMP C code
C++
SpecFile-C++
C++ user code
PyPartMC-C++
nlohmann::JSON
nanobind_json
Python
NumPy
Python user code
nanobind-generated
PyPartMC module
Julia
Julia user code
PyCall.jl
Matlab
Matlab user code
Matlab built-in
Python interface

Quest for a single *pip* install

Quest for a single *pip* install

To run PartMC, you need to:

Quest for a single *pip* install

To run PartMC, you need to:

- Install numerous dependencies

Quest for a single *pip* install

To run PartMC, you need to:

- Install numerous dependencies
- Perform multi-language compilation

Quest for a single *pip* install

To run PartMC, you need to:

- Install numerous dependencies
- Perform multi-language compilation
- Have knowledge of Bash, Fortran and Python

Quest for a single *pip* install

To run PartMC, you need to:

- Install numerous dependencies
- Perform multi-language compilation
- Have knowledge of Bash, Fortran and Python

To run PyPartMC, you need to:

Quest for a single *pip* install

To run PartMC, you need to:

- Install numerous dependencies
- Perform multi-language compilation
- Have knowledge of Bash, Fortran and Python

To run PyPartMC, you need to:

- **pip install pypartmc**

Quest for a single *pip* install

To run PartMC, you need to:

- Install numerous dependencies
- Perform multi-language compilation
- Have knowledge of Bash, Fortran and Python

To run PyPartMC, you need to:

- **pip install pypartmc**
- Done!

Quest for a single *pip* install

To run PartMC, you need to:

- Install numerous dependencies
- Perform multi-language compilation
- Have knowledge of Bash, Fortran and Python

To run PyPartMC, you need to:

- **pip install pypartmc**
- Done!
- Or: Single click execution of an example notebook on Google Colab!

Quest for a single *pip* install

- No uniformly used package manager for C++ or Fortran (compared to e.g. *pip*)

Quest for a single *pip* install

- No uniformly used package manager for C++ or Fortran (compared to e.g. *pip*)
- We link everything statically!

Quest for a single *pip* install

- No uniformly used package manager for C++ or Fortran (compared to e.g. *pip*)
- We link everything statically!
- We patch submodules where necessary

Quest for a single *pip* install

- No uniformly used package manager for C++ or Fortran (compared to e.g. *pip*)
- We link everything statically!
- We patch submodules where necessary

```
function(scoped_sundials_setup_config)
  set(PROJECT_SOURCE_DIR ${SUNDIALS_SOURCE_DIR})
  set(SUNDIALS_PRECISION "double")
  set(SUNDIALS_CINDEX_TYPE "int64_t")

  file(MAKE_DIRECTORY ${CMAKE_BINARY_DIR}/sundials)
  execute_process(
    COMMAND ${Python_EXECUTABLE} "-c" "import sys; print(''.join([line for line in sys.stdin if line.startswith('set(PACKAGE_VERSION_')]))"
    INPUT_FILE ${CMAKE_SOURCE_DIR}/gitmodules/sundials/CMakeLists.txt
    OUTPUT_FILE ${CMAKE_BINARY_DIR}/sundials/CMakeLists.txt
  )
  include(${CMAKE_BINARY_DIR}/sundials/CMakeLists.txt)

  set(PACKAGE_VERSION "${PACKAGE_VERSION_MAJOR}.${PACKAGE_VERSION_MINOR}.${PACKAGE_VERSION_PATCH}")
  include(${SUNDIALS_SOURCE_DIR}/cmake/SundialsSetupConfig.cmake)
endfunction()
scoped_sundials_setup_config()
```

Quest for a single *pip* install

```
if (MINGW)
  file(WRITE "${CMAKE_BINARY_DIR}/fix_mingw_wstat64.h" "
#pragma once
#include <sys/stat.h>
#ifdef _WIN64
#  undef _wstat64
#  define _wstat64(path, buf) _wstat64((path), (struct _stat64*)(buf))
#endif
")
  target_compile_options(netcdf_clib PRIVATE
    -include${CMAKE_BINARY_DIR}/fix_mingw_wstat64.h
  )
endif()
```

Biggest CMake trickery

Biggest CMake trickery

- When migrating from **pybind11** to **nanobind**, our build backend changed too

Biggest CMake trickery

- When migrating from **pybind11** to **nanobind**, our build backend changed too
- We needed to switch from **setuptools** to **scikit-build-core**

Biggest CMake trickery

- When migrating from **pybind11** to **nanobind**, our build backend changed too
- We needed to switch from **setuptools** to **scikit-build-core**
- scikit-build wanted to force installation of all CMake dependencies

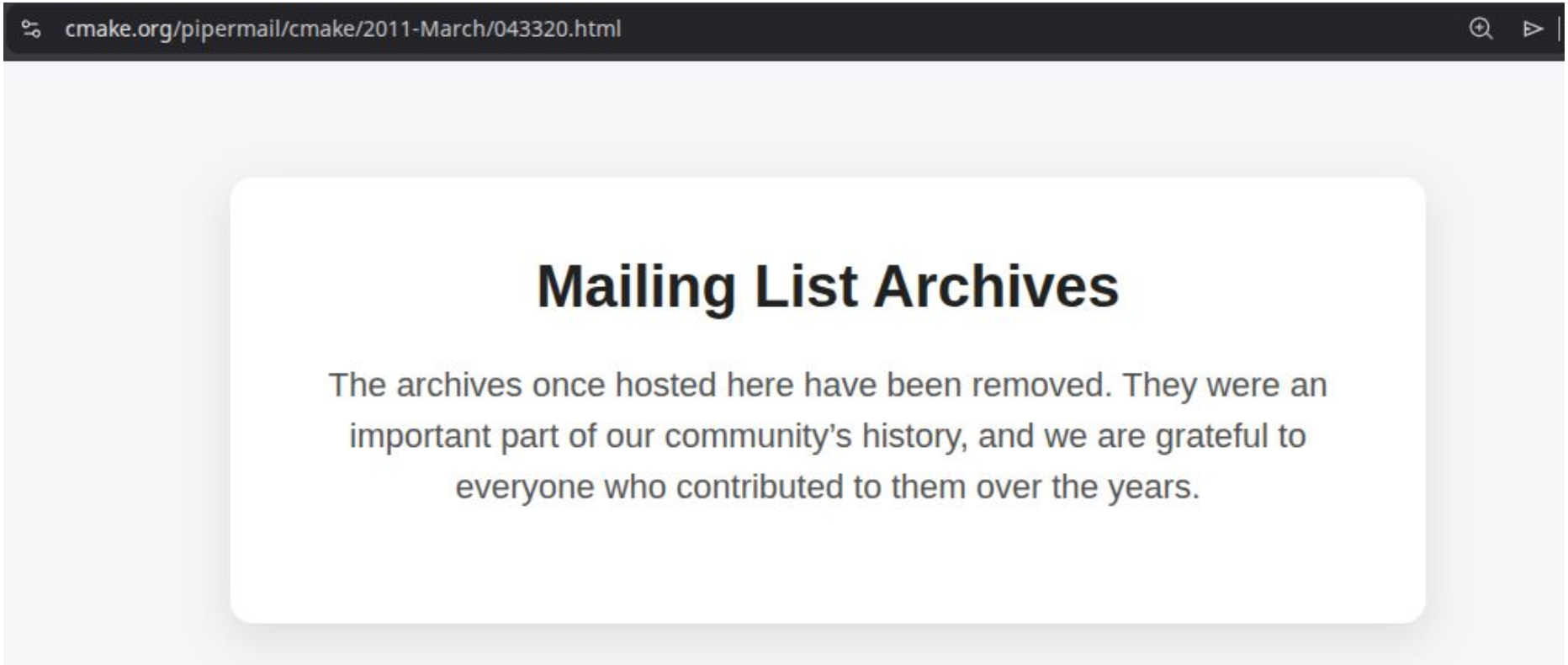
Biggest CMake trickery

- When migrating from **pybind11** to **nanobind**, our build backend changed too
- We needed to switch from **setuptools** to **scikit-build-core**
- scikit-build wanted to force installation of all CMake dependencies
- We need the *install()* to work for PyPartMC, but not any other package

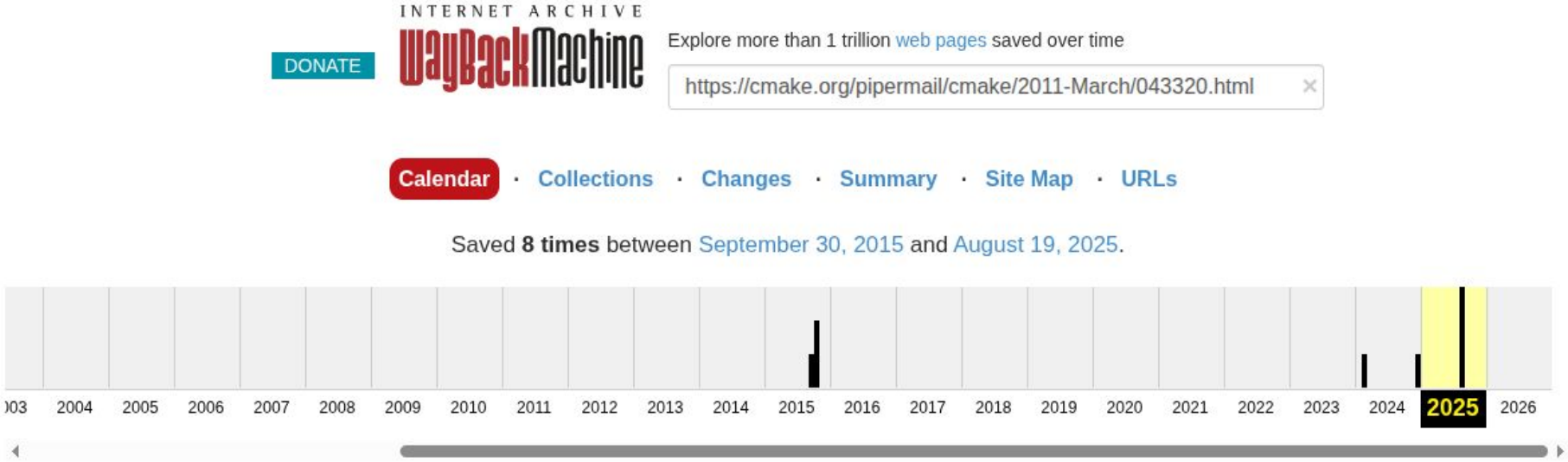
Biggest CMake trickery

- When migrating from **pybind11** to **nanobind**, our build backend changed too
- We needed to switch from **setuptools** to **scikit-build-core**
- scikit-build wanted to force installation of all CMake dependencies
- We need the *install()* to work for PyPartMC, but not any other package
- To prevent it, we had to resort to a undocumented CMake feature!

Biggest CMake trickery



Biggest CMake trickery



Biggest CMake trickery

I'm assuming you mean "_add_library(${name} ${ARGN})"?

The "override and call original with a leading _" is probably not well documented anywhere. While it's useful on occasion, we do not recommend using it if there's an alternate technique that you can use.

When you override a function, there is only one "_" original function reference saved in CMake's data structures. So... if you do it more than once, the _ version now refers to one of the ones you've defined rather than the real "orginal" one. It's more like the last one.

Biggest CMake trickery

```
macro(install)
endmacro(install)
```

Biggest CMake trickery

```
macro(install)
endmacro(install)
```

```
_install(TARGETS _PyPartMC LIBRARY DESTINATION PyPartMC)
```

Are git submodules maintainable?

Are git submodules maintainable?

Yes!

Are git submodules maintainable?

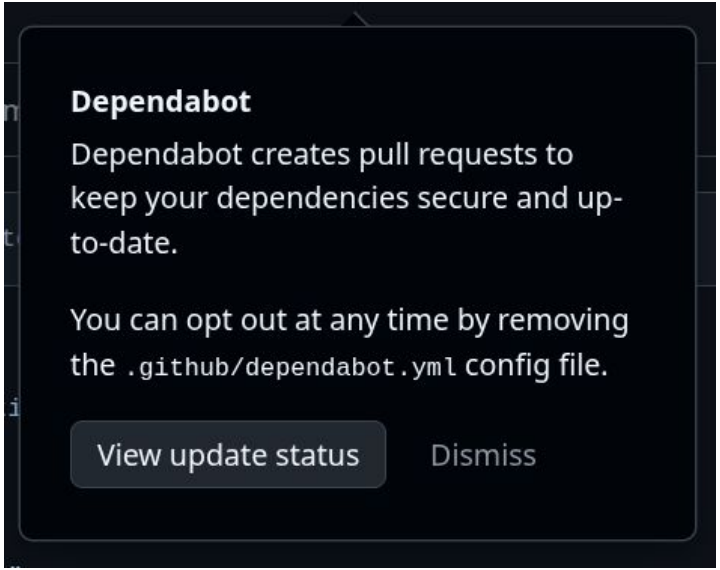
Yes!

```yaml
version: 2
updates:
  - package-ecosystem: "github-actions"
    directory: "/"
    schedule:
      interval: "weekly"
  - package-ecosystem: "pre-commit"
    directory: "/"
    schedule:
      interval: "weekly"
  - package-ecosystem: "gitsubmodule"
    directory: "/"
    schedule:
      interval: "weekly"
```

Are git submodules maintainable?

Yes!

```yaml
version: 2
updates:
  - package-ecosystem: "github-actions"
    directory: "/"
    schedule:
      interval: "weekly"
  - package-ecosystem: "pre-commit"
    directory: "/"
    schedule:
      interval: "weekly"
  - package-ecosystem: "gitsubmodule"
    directory: "/"
    schedule:
      interval: "weekly"
```



Packaging: cibuildwheel

Packaging: cibuildwheel

- Created and maintained by **Python Packaging Authority**

Packaging: cibuildwheel

- Created and maintained by **Python Packaging Authority**
- Easily create Python wheels for different operating systems and architectures!

Packaging: cibuildwheel

- Created and maintained by **Python Packaging Authority**
- Easily create Python wheels for different operating systems and architectures!
- Wheels available for all modern Python versions in CPython (3.9-3.14) and PyPy (3.9-3.11)

Packaging: cibuildwheel

- Created and maintained by **Python Packaging Authority**
- Easily create Python wheels for different operating systems and architectures!
- Wheels available for all modern Python versions in CPython (3.9-3.14) and PyPy (3.9-3.11)
- Build wheels, test your code, and upload it to PyPI all in your **CI workflow**!

Packaging: cibuildwheel

Packaging: cibuildwheel

- Build wheels on ubuntu-latest
- Build wheels on ubuntu-24.04-arm
- Build wheels on macos-14
- Build wheels on macos-15
- Build wheels on macos-15-intel
- Build wheels on windows-latest

```
Building cp313-manylinux_x86_64 wheel
CPython 3.13 manylinux x86_64

▶ Setting up build environment...
                                                    ✓ 0.09s
▶ Building wheel...
                                                    ✓ 179.37s
▶ Repairing wheel...
                                                    ✓ 2.45s
No audit required for this wheel, as it is not abi3
▶ Testing wheel...
                                                    ✓ 13.50s

✓ cp313-manylinux_x86_64 finished in 3 minutes

Building cp314-manylinux_x86_64 wheel
CPython 3.14 manylinux x86_64

▶ Setting up build environment...
                                                    ✓ 0.09s
▶ Building wheel...
                                                    ✓ 178.12s
▶ Repairing wheel...
                                                    ✓ 2.44s
No audit required for this wheel, as it is not abi3
▶ Testing wheel...
                                                    ✓ 14.39s

✓ cp314-manylinux_x86_64 finished in 3 minutes
```

CI example

```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_WINDOWS: pip install delvewheel
    CIBW_ENVIRONMENT_WINDOWS: >-
      CMAKE_ARGS="-DCMAKE_MAKE_PROGRAM=D:/a/_temp/msys64/mingw64/bin/ninja.exe"
      CMAKE_PROGRAM_PATH="D:/a/_temp/msys64/usr/bin"
      CMAKE_GENERATOR="Ninja"
      TEMP="D:/a/_temp/"
    CIBW_REPAIR_WHEEL_COMMAND_WINDOWS: delvewheel repair -w {dest_dir} {wheel}
    CIBW_BEFORE_BUILD_MACOS: brew reinstall gcc; pip install --only-binary=llvmlite llvmlite
    CIBW_ENVIRONMENT_MACOS: MACOSX_DEPLOYMENT_TARGET=`sw_vers -productVersion` SYSTEM_VERSION_COMPAT=0
    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
```

CI example

```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_WINDOWS: pip install delvewheel
    CIBW_ENVIRONMENT_WINDOWS: >-
      CMAKE_ARGS="-DCMAKE_MAKE_PROGRAM=D:/a/_temp/msys64/mingw64/bin/ninja.exe"
      CMAKE_PROGRAM_PATH="D:/a/_temp/msys64/usr/bin"
      CMAKE_GENERATOR="Ninja"
      TEMP="D:/a/_temp/"
    CIBW_REPAIR_WHEEL_COMMAND_WINDOWS: delvewheel repair -w {dest_dir} {wheel}
    CIBW_BEFORE_BUILD_MACOS: brew reinstall gcc; pip install --only-binary=llvmlite llvmlite
    CIBW_ENVIRONMENT_MACOS: MACOSX_DEPLOYMENT_TARGET=`sw_vers -productVersion` SYSTEM_VERSION_COMPAT=0
    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
```

CI example

```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_WINDOWS: pip install delvewheel
    CIBW_ENVIRONMENT_WINDOWS: >-
      CMAKE_ARGS="-DCMAKE_MAKE_PROGRAM=D:/a/_temp/msys64/mingw64/bin/ninja.exe"
      CMAKE_PROGRAM_PATH="D:/a/_temp/msys64/usr/bin"
      CMAKE_GENERATOR="Ninja"
      TEMP="D:/a/_temp/"
    CIBW_REPAIR_WHEEL_COMMAND_WINDOWS: delvewheel repair -w {dest_dir} {wheel}
    CIBW_BEFORE_BUILD_MACOS: brew reinstall gcc; pip install --only-binary=llvmlite llvmlite
    CIBW_ENVIRONMENT_MACOS: MACOSX_DEPLOYMENT_TARGET=$(sw_vers -productVersion | awk -F '.' '{print $1 "." 0}') SYSTEM_VERSION_COMPAT=0
    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
```

Summary: How to create Fortran to Python bindings?

Summary: How to create Fortran to Python bindings?

- Create an interface to C with Fortran’s **ISO C bindings**

Summary: How to create Fortran to Python bindings?

- Create an interface to C with Fortran's **ISO C bindings**
- Create Python bindings with **nanobind**/**pybind11** through C++

Summary: How to create Fortran to Python bindings?

- Create an interface to C with Fortran’s **ISO C bindings**
- Create Python bindings with **nanobind**/**pybind11** through C++
- Add **garbage collection** for Fortran objects if necessary

Summary: How to create Fortran to Python bindings?

- Create an interface to C with Fortran's **ISO C bindings**
- Create Python bindings with **nanobind**/**pybind11** through C++
- Add **garbage collection** for Fortran objects if necessary
- Maintain your git submodules with **dependabot** if necessary

Summary: How to create Fortran to Python bindings?

- Create an interface to C with Fortran's **ISO C bindings**
- Create Python bindings with **nanobind**/**pybind11** through C++
- Add **garbage collection** for Fortran objects if necessary
- Maintain your git submodules with **dependabot** if necessary
- Compile the library with **CMake**

Summary: How to create Fortran to Python bindings?

- Create an interface to C with Fortran's **ISO C bindings**
- Create Python bindings with **nanobind**/**pybind11** through C++
- Add **garbage collection** for Fortran objects if necessary
- Maintain your git submodules with **dependabot** if necessary
- Compile the library with **CMake**
- Package it as a Python wheel with **cibuildwheel** in your CI workflow

Summary: How to create Fortran to Python bindings?

- Create an interface to C with Fortran's **ISO C bindings**
- Create Python bindings with **nanobind**/**pybind11** through C++
- Add **garbage collection** for Fortran objects if necessary
- Maintain your git submodules with **dependabot** if necessary
- Compile the library with **CMake**
- Package it as a Python wheel with **cibuildwheel** in your CI workflow
- Enjoy!

Useful links

- PyPartMC Github: https://github.com/open-atmos/PyPartMC
- PyPartMC SoftwareX article: https://doi.org/10.1016/j.softx.2023.101613
- nanobind_json Github: https://github.com/Griger5/nanobind_json

Thank you!