EPJ Nuclear Sci. Technol. **10**, 15 (2024)

https://doi.org/10.1051/epjn/2024016

EPJ Nuclear Sciences & Technologies

Available online at:
https://www.epj-n.org

**Status and advances of Monte Carlo codes for particle transport simulation, edited by Andrea Zoia, Cheikh Diop and Cyrille de Saint Jean**

REGULAR ARTICLE

OPEN ACCESS

# Language and design evolution of the OpenMC Monte Carlo particle transport code

Paul Romano*, John Tramm, and Patrick Shriwise

Argonne National Laboratory, 9700 S Cass Ave, Lemont, IL 60439, USA

Received: 9 July 2024 / Received in final form: 9 July 2024 / Accepted: 28 August 2024

**Abstract.** The OpenMC Monte Carlo particle transport code has been continuously developed for 13 years by a large community of contributors. In that time span, the codebase has undergone significant changes that have redefined what OpenMC is and made it an enduring presence in the nuclear science and engineering community. In this paper, we discuss the evolution of programming language use in OpenMC, trends in the overall design of the programming interfaces, and implications for the future of the code.

## 1 Introduction

The OpenMC Monte Carlo particle transport code [1] has been continuously and actively developed since 2011 by over 100 contributors spanning many organizations around the world. Naturally, the code has evolved significantly over time as a result of sponsored research projects and increased use of the code by a rapidly growing user community. While some core features of OpenMC have remained the same over time, other areas of the code have substantially changed and will likely continue to change over time.

Previous works in the literature have highlighted major feature additions to OpenMC [2–6]. In this paper, we discuss the evolution of the OpenMC codebase over time, highlighting key inflection points that have fundamentally altered the trajectory of the code and–in our opinion–are a key reason the code has found an enduring presence in the nuclear community and beyond. There are two main themes we discuss: the evolution in the use of different programming languages in OpenMC, and the evolution of the design of application programming interfaces (APIs) and their interconnectedness.

## 2 Language evolution

The choice of what programming language to use for an application can be one of the most consequential decisions for its development team. This choice affects how easy or difficult it is to build and install the software, what platforms it may be used on, what third-party libraries may be available to use, the pool of users and developers that it may attract, and much more. At its inception, OpenMC was written fully in standard Fortran 2008, which has always been a popular language for scientific computing applications and one with a long history and track record in the nuclear industry. The early differentiators for OpenMC revolved around performance (parallel scaling on large clusters), ease of use (e.g., the use of XML as an input format instead of an arbitrary ASCII format), and its open-source nature. Other than those aspects, the code was not too dissimilar to other popular Monte Carlo codes.

While using Fortran was perhaps a "safe" choice for a code targeting nuclear science applications, it did place limitations on the development of the code and how it was used. Because of this, over time the code has shifted away from the use of Fortran toward the use of Python and C++. The motivations for these two languages are quite different, so they will be discussed separately.

### 2.1 Python adoption

The first major trend in programming language use in OpenMC has been the gradual addition of Python code over time, which now accounts for about two-thirds of all lines of code. The first introduction of Python in the OpenMC codebase was several Python scripts that were introduced as a way of post-processing output files. This was followed by the addition of a regression test suite built in Python. Then in 2015, a proper Python API was added to the code that enabled users to programmatically generate XML input files. Over time the Python API has grown in scope and contains extensive capabilities, including classes for post-processing, geometry visualization, cross-section generation (`openmc.mgxs`), depletion (`openmc.deplete`), and a nuclear data interface (`openmc.data`).

* e-mail: promano@anl.gov

Table 1: Pull requests on the OpenMC repository that translated Fortran code to C++.

| Component | Pull request | Date | Lines of change |
|---|---|---|---|
| Random number generator | #939 | 2017-12-01 | 529 |
| Constructive solid geometry surfaces | #959 | 2018-01-24 | 3756 |
| HDF5 interface | #996 | 2018-04-25 | 5762 |
| Miscellaneous math functions | #1004 | 2018-05-13 | 2332 |
| Most geometry data | #1012 | 2018-05-27 | 5284 |
| Multigroup transport | #1016 | 2018-06-17 | 8735 |
| Particle type | #1035 | 2018-08-06 | 1371 |
| Nuclear data hierarchy | #1042 | 2018-08-10 | 52447 |
| Material class | #1045 | 2018-08-15 | 1022 |
| Thermal scattering data | #1047 | 2018-08-16 | 2219 |
| External source distributions | #1059 | 2018-08-23 | 2006 |
| "Find cell" and "distance to boundary" | #1061 | 2018-08-24 | 3988 |
| Simulation settings | #1062 | 2018-08-28 | 1591 |
| Meshes | #1066 | 2018-09-05 | 3815 |
| Multigroup physics | #1092 | 2018-10-12 | 878 |
| MPI parallelism | #1094 | 2018-10-15 | 2122 |
| Tally filters | #1101 | 2018-10-19 | 6371 |
| Initialization/finalization | #1105 | 2018-10-25 | 2528 |
| Geometry visualization | #1107 | 2018-10-26 | 2502 |
| Remaining tally filters | #1110 | 2018-10-28 | 1423 |
| Neutron physics | #1128 | 2018-11-26 | 4402 |
| Probability tables | #1136 | 2018-12-06 | 771 |
| Cross section lookups | #1144 | 2019-01-02 | 2566 |
| Photon physics | #1148 | 2019-01-15 | 3577 |
| Material class | #1152 | 2019-01-30 | 5942 |
| Output functions | #1154 | 2019-02-06 | 2612 |
| Tally scoring | #1162 | 2019-02-11 | 11805 |
| Volume calculations | #1164 | 2019-02-12 | 1354 |
| Transport and surface crossing | #1169 | 2019-02-15 | 4587 |
| Remaining code | #1171 | 2019-02-21 | 11514 |

While it was not immediately the case when the Python API was introduced, it has now become the primary means for users to interact with the code, and so from a user's perspective, OpenMC is essentially a Python code. The growing adoption and use of Python in OpenMC has come at a time when Python itself was also growing rapidly in popularity, arguably having now become the most popular programming language in the world [7]. Thus, the Python API has been a major asset for OpenMC, both in attracting power users who are able to create sophisticated workflows as well as more casual scientists and engineers, most of whom come in with at least some basic familiarity with Python. The prevalence of Python code has also helped attract less experienced developers who are eager to contribute to an open-source project (e.g., student projects), which creates a talent pipeline for the development team.

### 2.2 Fortran to C++ conversion

The decision to use Fortran when starting the development of OpenMC was based on the initial research focus for the code on parallel algorithms coupled with the fact that coarray features had been added to the Fortran 2008 standard, as explained in [1]. However, coarrays were never actually utilized and instead, parallelism was enabled with a traditional combination of MPI for distributed memory and OpenMP for shared-memory parallelism. This obviated the primary *raison d'être* for using Fortran in the first place. Recognizing this, in 2016 the OpenMC community agreed in principle[1] to translate the code from Fortran to C++, which was driven by many factors, including the use of the standard library and external libraries, compiler and hardware vendor support, interoperability with community codes, and the available talent pool.

[1] https://github.com/openmc-dev/openmc/issues/603

There were two possible approaches to actually carry out that transition: either start from scratch and build an entirely new code in C++, or gradually translate the codebase one module at a time. While no official strategy was agreed upon, the latter approach occurred organically. The first Fortran to C++ translation happened in December 2017 and over a period of about 15 months, a total of 30 pull requests were made that translated components one at a time. Table 1 shows the full list of language translation pull requests submitted during that time period. This list highlights the fact that most of the changes were modest in scope (a few thousand lines of code changed or less). In the end, only three "hero" pull requests that modified more than 10,000 lines of code were needed.

While a gradual transition of the codebase seemed inconceivable at the onset, in retrospect it is very clear to us that this was truly the only feasible strategy. OpenMC had already gained enough of a following and a development community that many changes were happening unrelated to the translation of the code from Fortran to C++. Notably, while the transition was underway, several major feature were added to OpenMC, including depletion [5], photon transport [2], and CAD-based geometry [4]. Early on in the translation effort, additional features mostly added new Fortran code, whereas once the translation effort had reached maturity, new features added to the code were coming in directly as C++. The gradual transition also had the benefit that it required very little coordination among developers; one developer could simply let others know they were going to work on translating a particular module and all other work could proceed as normal. On the flip side, trying to rewrite the core codebase from scratch effectively would have meant pausing all ongoing developments, which would have significantly hampered the further growth of the code.

While the overall experience of gradually translating the code was mostly positive, it was not without downsides. One effect of the gradual transition is that some of the resulting C++ code was not "idiomatic" C++ and inherited the procedural design style of the existing Fortran code. Some of these design patterns still exist today, representing a continuing source of technical debt–that is, poorly designed or inefficient code that was originally written to save time but that has to be "paid back" in the form of future maintenance costs. A ground-up rewrite would have given more opportunity to adopt better design patterns (e.g., avoiding global data).

Figure 1 shows a breakdown of the percentage of various programming languages over time as measured by the Linguist tool[2], illustrating both the increasing prevalence of Python as well as the elimination of Fortran by early 2019. Note that C++ code did exist in OpenMC prior to the translation effort solely from embedded third-party libraries. As of the time of writing, OpenMC is 68% Python and 31% C++. Of the Python code, about three-quarters is application code and the remaining quarter is test scripts.

[2] https://github.com/github-linguist/linguist

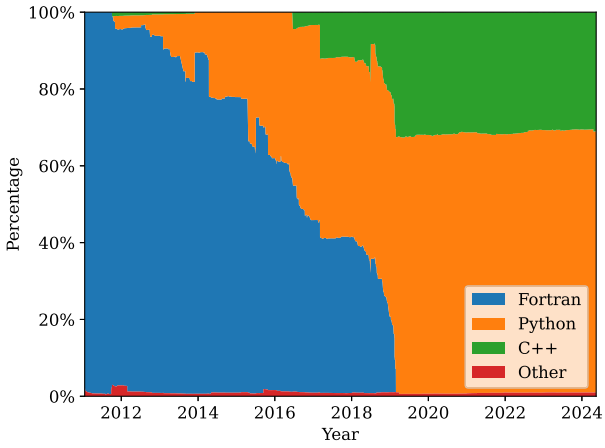


Fig. 1: Breakdown of programming language use in OpenMC over time as measured by percentage of the total lines of code.

### 2.2.1 Translation strategies

During the course of the translation effort, our team adopted several strategies that allowed the code to exist in a transitional state, with some functionality written in Fortran and some functionality in C++ simultaneously. The primary mechanism that allowed all of this to work was features in the Fortran 2003 standard that enable interoperability with C, namely the ability to reference functions defined in C from Fortran, and the ability to expose Fortran functions with an interface that can be called from C. While C++ is of course not identical to C, one can define C++ functions that obey C calling conventions and therefore appear to Fortran as though they are C functions, enabling the use of the C interoperability features from the Fortran standard.

With Fortran code calling C/C++ code and vice versa, any data passed between the two must be of equivalent datatypes. To ensure this, the `ISO_C_BINDING` intrinsic module in Fortran provides type parameters that match C datatypes. For example, declaring `real(C_DOUBLE)` in Fortran is equivalent to a `double` in C. With these named constants, it is possible to match the basic datatypes from C (integer and floating point types, booleans, and characters). Derived types are also defined in the `ISO_C_BINDING` module that match any C object pointer type.

The first strategy that we employed when translating Fortran code to C++ was to have a set of functions written in C++ and callable from Fortran wherein the data passed between the two languages is limited to basic types like int or double. The set of functions is usually related to one specific area in the code (for example, random number generation). This strategy worked well when no internal state needed to be exposed to code that hadn't yet been translated. A concrete example of this strategy is shown in Listing 1. In this example, multiple functions for generating random numbers and advancing the random number generator state were exposed. The Fortran code

```cpp
extern "C" {
  double prn();
  void advance_prn_seed(int64_t n);
  ...
}
```

```f90
interface
  function prn() result(pseudo_rn) bind(C)
    real(C_DOUBLE) :: pseudo_rn
  end function prn

  subroutine advance_prn_seed(n) bind(C)
    integer(C_INT64_T), value :: n
  end subroutine advance_prn_seed
  ...
end interface
```

Listing 1: Example of a simple interface between C++ (top) and Fortran (bottom) for a function returning a random number.

could call these functions without needing to "own" any data.

In most cases, the simple strategy above didn't work because both the Fortran and C++ code needed access to the internal details of a particular object. To handle such cases, our strategy was to have a class implemented entirely in C++ and a "mirror" object on the Fortran side that holds an opaque pointer to the corresponding C++ object. Thus, to access an attribute of the class from Fortran, a type-bound procedure would be called that interfaces with a function exposed from C++ that accepts the opaque pointer. Listing 2 shows how this strategy worked in practice for the Surface class in OpenMC. On the C++ side, a function (`surface_pointer`) is defined that returns a pointer to a Surface object for a given index in an array of surfaces. Additionally, one `extern` "`C`" function is exposed for each operation on the Surface; the function takes as its first argument the Surface pointer. In the Fortran code, a simple derived type holds the Surface pointer and has a corresponding type-bound procedure for each operation on the surface. Thus, calling `surface%reflect()` from Fortran would pass the pointer to the `surface_reflect()` interface function defined in C++, which would then call the `reflect()` method on the actual Surface object. Note that simple data members on the C++ class still have to be accessed with via type-bound procedures on the Fortran side (instead of writing `surface%id`, one would write `surface%id()`).

The strategy of having a Fortran class that owns a single opaque pointer can become cumbersome if all the data members of the class are needed on the Fortran side. There were a few cases where this was true when translating OpenMC. For example, both the Fortran and C++ code needed access to all data members of the main Particle class. To handle this, the derived type on the Fortran side was given the `BIND` attribute so that it could be interoperable with a C struct. Unfortunately, one side effect was

```cpp
class Surface {
public:
  bool reflect(double xyz[3], double uvw[3]);
  ...
};

extern "C" {
  Surface* surface_pointer(int index) {
    return global::surfaces[index];
  }
  bool surface_reflect(Surface* surf, double xyz[3],
                       double uvw[3]);
  ...
}
```

```f90
interface
  function surface_pointer(index) bind(C) result(ptr)
    integer(C_INT), intent(in), value :: index
    type(C_PTR)                       :: ptr
  end function surface_pointer

  subroutine surface_reflect(surf_ptr, xyz, uvw) bind(C)
    type(C_PTR),    intent(in), value :: surf_ptr
    real(C_DOUBLE), intent(in)        :: xyz(3)
    real(C_DOUBLE), intent(inout)     :: uvw(3)
  end subroutine surface_reflect
  ...
interface

type :: Surface
  type(C_PTR) :: ptr
contains
  procedure :: reflect => surface_reflect
  ...
end type
```

Listing 2: Example of an interface between C++ (top) and Fortran (bottom) for a surface class.

that it was not possible to have type-bound procedures on the derived type, so instead, a set of interoperable functions were declared that took the Particle derived type as their first argument.

### 2.3 Execution on GPUs

Under the Exascale Computing Project, the OpenMC team has developed a branch of the code that is capable of executing on GPUs by using the target offload features of the OpenMP 5.0 standard [8]. This capability is not currently part of the production branch of OpenMC, primarily because some sacrifices had to be made in the design of the code in order to be compatible with the GPU execution models. For instance, many classes were changed to remove runtime polymorphism, which is generally not supported in most GPU programming models. More complex algorithms were also utilized to avoid dynamic memory allocation in device regions of the code. Additionally, usage of the C++ standard template library (STL) had to be curtailed in device regions of the code, as the STL is not typically available on the device. A significant amount of data mapping code also had to be introduced to facilitate serialization and deep copying of most of OpenMC's data structures into device memory. Bringing these devel-

opments back into the production branch would represent the largest programming model change in OpenMC since the Fortran to C++ transition. It remains to be seen whether this could be done while upholding other goals such as maintainability, full testing coverage, and readable code that is relatively easy for newcomers to grasp. However, newer features in the OpenMP standard (and their recent support in production compilers) may remove the need for some of these added complexities, potentially reducing the degree of complexity required for OpenMP offloading to devices in OpenMC in the near future.

## 3 Design evolution

The language transitions discussed in Section 2 are just one part of the story of the evolution of OpenMC. Since the Python API was added to the code in 2015, the development team has sought to improve it over time by adopting intuitive design patterns. Below, we discuss some of the ways that the internal design of APIs and user interaction has changed.

### 3.1 File formats and serialization

Originally, an OpenMC model always consisted of at least three XML files: geometry.xml, materials.xml, and settings.xml. Optional tallies were specified separately in a tallies.xml file. This decision was made for a number of reasons, including the ability to reuse materials across multiple models and to separate simulation settings from the geometry and materials so that they could be easily and independently modified. However, over time there has been increasing motivation to have a single XML file that represents the entirety of a model (e.g., to contain a benchmark problem in a single XML file). To meet this need, a `Model` class that holds references to geometry, associated materials, settings, and tallies was introduced and has found increasing use throughout the API. At first, serializing a `Model` class to disk would still write separate XML files. However, with the 0.13.3 release of OpenMC [9], it now supports writing a single XML file that holds all model information. This has a number of benefits:

- Multiple models can be stored in a single directory and given arbitrary filenames rather than being forced to use specific filenames.
- Constructs that appear in multiple places are no longer defined redundantly. For example, a mesh might be used in the definition of a tally (in tallies.xml) but also for weight windows (in settings.xml). With separate files, the mesh would need to be defined multiple times, which can create ambiguities during input processing. With a single XML file, the mesh is defined once and can then be referenced by its unique ID.
- Exporting and importing a model becomes cleaner from Python code. For example, users only need a single method call from Python to export or import a model.

Another evolution in handling files and serialization is how information that has been exported once is read back into the Python API. When OpenMC is executed, it will produce a summary.h5 file that contains the geometry information and a statepoint.h5 model that contains tally definitions and results. The Python API can reconstruct the original model based on these files, but having it in an HDF5 format means that a whole set of export/import functions is needed to understand that format (in addition to the XML format). While the .h5 formats have long existed in OpenMC, more recently the `Model` class was given a `from_model_xml()` method that allows the Python object hierarchy to be recreated from the model XML file. In the long term, this means that the serialization of the model information to HDF5 files may become unnecessary, creating a single canonical representation for serialization. Doing so also would make it easier for OpenMC to shift to some data serialization format if desired (e.g., JSON or YAML).

### 3.2 Internal execution of OpenMC

When OpenMC is built, the user is provided an `openmc` executable that can be run from a command line. The Python API also provides multiple methods for directly executing the code from a Python script. The `openmc.run()` function and the `Model.run()` method calls the `openmc` executable in a subprocess. Additionally, OpenMC has a lower-level C/C++ API that enables users to control execution by making calls directly into a shared library (libopenmc) [6]. Python bindings to the C/C++ API exist as part of the `openmc.lib` module, meaning that advanced features enabled by the use of the C/C++ API can be part of a Python-driven workflow as well. Both of these capabilities were originally designed for direct use by users, but increasingly they have found use internally in OpenMC itself, providing a way to execute numerically intensive code from Python while still being seamless for a user.

Some use cases for internal execution of OpenMC from the Python API are quite obvious. For example, a `search_for_keff()` function that performs a criticality search has long existed in the Python API and iteratively runs OpenMC, modifying a parameter until an appropriate value is found that results in a target $k_{\text{eff}}$. The built-in depletion solver is part of the `openmc.deplete` module also calls OpenMC internally (via the `openmc.lib` module) at each depletion timestep in order to update reaction rates in a depletion matrix.

Beyond these obvious use cases, there have been other areas where the ability to execute OpenMC internally has been very beneficial. As one example, OpenMC used to have multiple methods for generating a rasterized slice plot of geometry. One method was to run the `openmc` executable in plotting mode, which would directly produce a PNG file from the C++ code. Another method was to call a `plot` method from Python, which then relied on logic written in Python to determine what cell/material was present at a given spatial position. The need to have redundant logic for doing "find cell" operations (in both

C++ and Python) led to inconsistencies in behavior and ultimately many bugs. In version 0.13.1 [10], the Python `plot` methods were overhauled so that instead of relying on redundant logic written in Python, they would directly run the `openmc` executable under the hood to generate the rasterized plot as a PNG file. To the user, the result is exactly the same; if they execute the `plot` method from a Jupyter Notebook or Python interpreter, the resulting image is displayed. Under the hood though, OpenMC creates a temporary directory, exports the model, calls the `openmc` executable, and loads the PNG file that was generated.

For some use cases, there is a need to call specific functions from the OpenMC C API rather than simply running the executable to generate a result. For example, in the depletion module of OpenMC, there is a function that can generate a set of microscopic cross sections for depletion based on an input multigroup flux. Performing the flux collapse of continuous-energy neutron cross-sections would be inefficient from Python, so instead the `openmc.lib` module is used to call a function in the C API that can perform the collapse based on cross-section data loaded from the C++ code. This helps reduce redundant code while improving performance. Another recent example is a function for calculating volume fractions of materials within mesh elements. This type of calculation can be done by leveraging the existing mesh infrastructure in the C++ code, so a C API function was written that can be called from Python. Again, to the user, the execution of functions from the OpenMC shared library under the hood is invisible.

### 3.3 Type hints

One of the most exciting areas of development related to the Python language itself is the rapid expansion and adoption of capabilities for type hinting–optional annotations that can be added to variables, function arguments, and return values that indicate expected types. Recently, type hints have started to be added to Python code within OpenMC. This benefits both users and developers; for users, type hints provide a better experience by indicating what types are valid for a particular function/method, enabling autocompletion, and allowing the use of static type checkers. For developers, type hints can help provide insights into existing code since many IDEs are able to infer variable types in the middle of a function. As a result, we expect that future versions of OpenMC will be fully or mostly type-hinted.

### 3.4 Solvers

OpenMC is most often thought of as "a Monte Carlo code," and indeed the Monte Carlo transport solver within OpenMC is its main feature. The scope of OpenMC has grown enough such that it does not make sense to classify it as only a Monte Carlo code though. For one, OpenMC has a built-in depletion/transmutation solver in the `openmc.deplete` that can be used independently of the Monte Carlo transport solver, placing it among other depletion solvers such as ORIGEN [11] and FISPACT [12]. The nuclear data interface in the `openmc.data` module can also be used on its own for inspecting ENDF files and analyzing nuclear data. Recently, a solver based on the random ray method [13]–a stochastic transport method closely related to the method of characteristics–was added to OpenMC[3], leveraging the existing constructive solid geometry representation and multigroup transport capabilities. The random ray solver (which has a relatively uniform spatial uncertainty distribution) is intended to be used for weight window generation and/or as a fast, reduced-order solver instead of the Monte Carlo solver. Thus, with this collection of solvers and capabilities, OpenMC can be seen as a framework for particle transport work encompassing more than just its core Monte Carlo solver.

### 3.5 Conclusion

OpenMC has undergone substantial change and evolution since it began as a small research code in 2011. The original Fortran code that once made up all of OpenMC has been completely replaced by C++, and the introduction of a Python API has redefined how users and developers interact with the code. The design of OpenMC continues to change over time, from what the code expects in terms of input files (Sect. 3.1) to how the Python API collects information from running the `openmc` executable (Sect. 3.2).

Even though the makeup of programming languages in OpenMC is now relatively stable, as shown in Figure 1, we fully expect that the design of the code will continue to evolve and improve over time in response to user needs, lessons learned from years of experience, and underlying changes in the Python ecosystem and the C++ language standard. One major question that the development team is likely to confront is whether the Python API should be broken up into multiple Python modules, each of which would have a limited scope. For example, one could imagine the nuclear data interface and/or the depletion solver existing as their own independent packages. This question is closely tied to how the application is ultimately packaged and distributed, which is also likely to change as packaging/distribution systems evolve.

#### Funding

This work is supported by the U.S. Department of Energy Office of Fusion Energy Sciences under award number DE-SC0022033. This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration.

#### Conflicts of interest

The authors declare that they have no competing interests to report.

#### Data availability statement

The data that support the findings of this study are openly available at the following DOI: 10.5281/zenodo.12696522.

[3] https://github.com/openmc-dev/openmc/pull/2823

### Author contribution statement

Paul K. Romano: Conceptualization, Writing – Original Draft, Project administration, Funding acquisition, Visualization, Software. John Tramm: Writing – Review & Editing, Software. Patrick Shriwise: Writing – Review & Editing, Software.



### References

1. P.K. Romano, N.E. Horelik, B.R. Herman, A.G. Nelson, B. Forget, OpenMC: A state-of-theart Monte Carlo code for research and development, Ann. Nucl. Energy **82**, 90 (2015)
2. A.L. Lund, P.K. Romano, *Tech. Rep. ANL/MCS-TM-381*, (Argonne National Laboratory, Lemont, Illinois, 2018)
3. W. Boyd, A. Nelson, P.K. Romano, S. Shaner, B. Forget, K. Smith, Multigroup cross-section generation with the OpenMC Monte Carlo particle transport code, Nucl. Technol. **205**, 928 (2019)
4. P.C. Shriwise, X. Zhang, A. Davis, DAG-OpenMC: CAD-based geometry in OpenMC, Trans. Am. Nucl. Soc. **122**, 395 (2020)
5. P.K. Romano, C.J. Josey, A.E. Johnson, J. Liang, Depletion capabilities in the OpenMC Monte Carlo particle transport code, Ann. Nucl. Energy **152**, 107989 (2021)
6. P.K. Romano, P.C. Shriwise, S.M. Harper, A.E. Johnson, A.L. Lund, J. Liang, J.R. Tramm, A. Davis, D. Short, Y. Park, Recent developments in the OpenMC Monte Carlo particle transport code, in *PHYSOR* (Pittsburgh, Pennsylvania, 2022)
7. S. Cass, *The Top Programming Languages 2023* (IEEE Spectrum, 2023)
8. J. Tramm, P. Romano, P. Shriwise, A. Lund, J. Doerfert, P. Steinbrecher, A. Siegel, G. Ridley, Performance portable Monte Carlo particle transport on Intel, NVIDIA, and AMD GPUs (2024) https://doi.org/10.48550/arXiv.2403.12345
9. P.K. Romano et al., OpenMC 0.13.3 (2023) https://doi.org/10.5281/zenodo.7783023
10. P.K. Romano et al., OpenMC 0.13.1 (2022) https://doi.org/10.5281/zenodo.7010045
11. I.C. Gauld, G. Radulescu, G. Ilas, B.D. Murphy, M.L. Williams, D. Wiarda, Isotopic depletion and decay methods and analysis capabilities in SCALE, Nucl. Technol. **174**, 169 (2011)
12. J.C. Sublet, J.W. Eastwood, J.G. Morgan, M.R. Gilbert, M. Fleming, W. Arter, FISPACT-II: An advanced simulation system for activation, transmutation and material modelling, Nucl. Data Sheets **139**, 77 (2017)
13. J.R. Tramm, K.S. Smith, B. Forget, A.R. Siegel, The Random Ray Method for neutral particle transport, J. Comput. Phys. **342**, 229 (2017)

**Cite this article as:** Paul Romano, John Tramm, Patrick Shriwise. Language and design evolution of the OpenMC Monte Carlo particle transport code, EPJ Nuclear Sci. Technol. **10**, 15 (2024)