university of groningen

faculty of science and engineering

PanPelagos-Core — Developing a Fast and Flexible Lagrangian Fluid Transport Simulator for Advanced Physics Applications

Research Internship at University of Groningen

June 2025

Authors:
Fabio Fragiacomo
Djairo Hougee
Martin Opat

Primary Supervisor:
Dr. Christian Kehl

Secondary Supervisor:
Dr. Julian Köllermeier

Abstract

There are various sources for pollutants in domestic and international waters, among which shipping container accidents releasing microplastics, drilling rigs spilling oil, and accidental release of nucleotide particles are some of the most pressing. To effectively plan disaster responses and assess the long-term effects of such events, particle tracking in natural environments is becoming increasingly relevant. Numerous Lagrangian transport simulators exist in the literature; however, these are generally constrained in performance or usability by the use of interpreted or legacy languages, respectively. This project presents a performant Lagrangian simulator developed in C++, designed with an extensible architecture and support for user-accessible Python components. To minimise Python's runtime overhead, Shedskin, a Python-to-C++ transpiler, is used. Furthermore, the simulator supports a plug-n-play workflow, exchanging select code snippets with user-written implementations at runtime. These plugins are compiled separately from the project, preventing extraneous recompilation. In performance evaluation, PanPelagos-Core was 7.1 ± 0.1 times slower than current state-of-the-art simulators, primarily due to its Array-Of-Structure data layout. However, many performance optimisations, including switching to a Structure-Of-Arrays data layout and parallelisation, remain to be implemented.

Workload attribution form
MSc Computing Science – Msc Internship Project

Purpose / format:
In order to assure a fair work balance, equal comprehension and subject progression, and the meeting of the learning objectives for every student, you are requested to fill out the following table. In the table, please fill out the following things:

- Header - row: either student number (S-number) or name of the team members; 1 name or S-number per column
- 1st column: the list of tasks/exercises/learning objective (from 1 to *N*) contained in the learning activity; 1 factor per row
- Other cells: mark with an 'X' which assignment each member has worked on. Multiple markers per row (i.e. per task/exercise) possible.

After filling out the table, you are required to **all sign the attribution form**. The signature can (of course) be imported digitally. The attribution form **must be consensual**, meaning that an attribution form without the signature of all members will count as a 'failed' assignment (grade: 0). This assures all team members agree with the written workload distribution.

In the final step, save / export / print the document as a **PDF file** and upload the PDF with your submitted code to *Brightspace.*

Deliverable	Name/Task	Djairo	Fabio	Martin
Report	Abstract	xx		xx
	Introduction	xxx		x
	Literature review – Ocean Simulation	xxx		x
	Literature review - LLVM	xxx		
	Literature review - Codon	xxx		x
	Literature review - Swig		xxx	
	Literature review - ShedSkin	x		xxx
	Implementation - Architecture		xx	xx
	Implementation - Plugins	xxx		
	Implementation – Swig		xxx	
	Implementation – ShedSkin			xxx
	Implementation – Paraview		xxx	
	Methods			xxx
	Results			xxx
	Discussion	xx	xx	xx
	Conclusion	xx	xx	xx
Code	Architecture	xx	xx	xx
	Core[1]	xxxxx	xxxxx	xxxxxxxx
	Plugins	xxx		
	ShedSkin			xxx

1 Determined by looking at relative contributions of each module.

	Swig		xxx	
Text that comes with the code but is not code	Documentation	x	xxx	x
	How-to-use guide	xx	xx	
Misc.	Codon Research		xxx	
	Performance optimisation	x		xxx

xxx = **Primary author**
xx = **Joint authorship**
x = **Edits / revision**

university of groningen

1. Signatures

We hereby declare that we have distributed the work of the assignment in the above-indicated contributions.

Student number: s4686772
Student name and Signature:

Fabio Fragiacomo

Student number: s4764153
Student name and Signature:

Djairo Hougee

Student number: s4704126
Student name and Signature:

Martin Opat

Contents

1 Introduction **8**

2 Literature review **8**
- 2.1 Ocean Simulation . . . 8
- 2.2 Language interoperability . . . 9
  - 2.2.1 LLVM . . . 9
  - 2.2.2 Codon . . . 9
  - 2.2.3 Swig . . . 9
  - 2.2.4 ShedSkin . . . 10

3 Implementation **10**
- 3.1 Architecture . . . 10
  - 3.1.1 Core . . . 10
  - 3.1.2 Particle . . . 11
  - 3.1.3 Kernel and Boundary Handler . . . 11
  - 3.1.4 Grid and Field . . . 11
  - 3.1.5 Algebra . . . 12
- 3.2 Plugins . . . 12
- 3.3 Swig . . . 12
- 3.4 ShedSkin . . . 14
- 3.5 Paraview . . . 15

4 Methods **15**
- 4.1 Data . . . 15
- 4.2 Benchmarking . . . 15

5 Results **16**

6 Discussion **17**
- 6.1 Performance & Memory Usage . . . 17
- 6.2 Modularity . . . 18
- 6.3 Future Work . . . 19

7 Conclusion **19**

8 References **21**

Appendix A **23**
- Source Code . . . 23

Appendix B **24**
- Array vs Linked List Measurements . . . 24
- Swig Methods Measurements . . . 25

Appendix C **26**
Plugin factory functions 26
System Architecture Diagrams 26

List of Figures

3.1 Swig wrapping and embedding workflow. 13
3.2 ShedSkin transpilation and integration workflow. 14
5.1 Performance comparisons of different technologies. 16
5.2 CPU time comparisons. 17
B.1 Boundary and Particle container comparisons 24
B.2 CPU time comparison between C++ and Swig. 25
C.1 System Architecture Diagram 27

1 Introduction

Particle ocean simulation is a widely used technique to model a variety of phenomena, both natural and man-made. Applications include oil spill modelling, disaster relief planning, and plastic particulate dispersal studies, which all make extensive use of simulation programs where real data does not exist or is not accessible. A multitude of Eulerian-Lagrangian ocean simulators exist in the literature [1]. However, due to different intended application areas and design priorities, the majority of these simulators are either highly configurable, mostly simulators written in interpreted languages like Python, e.g. OceanParcels [2], or computationally performant - simulators written in compiled languages like C++ or FORTRAN, e.g. TRACMASS [3]. Few to no simulation systems excel at both performance and flexibility, as found in prior work by Hougee et al. [4].
This project puts forth PanPelagos-Core (PPC), an Eulerian-Lagrangian particle simulator which aims to solve this gap by focusing on both performance *and* flexibility. PPC is designed as a robust foundation, balancing flexibility and performance while supporting diverse future extensions.
The main question this project aims to answer maps to the above description: "How is a flexible, yet high-performance Lagrangian fluid transport simulation framework designed and developed?". To aid in answering this question, certain key areas are further explored: First, **language interoperability**, required for flexibility, is an accessible interface to the simulator. To address this, the sub-question "*How can Python-based components be integrated into a particle simulator while retaining C++ performance?*" is investigated.
Next is **modularity**; another significant aspect of flexibility. A modular architecture enhances the range of applications and use cases for the simulator. For this aspect, the sub-question "*What design decisions are required such that critical components of a particle simulator can be exchanged at runtime without impacting usability and performance?*" is answered.
Lastly, **efficiency**, which is key to performance. For this focus area, the sub-question "*What design choices can be made to increase the performance of a Lagrangian particle simulator?*" is examined.

2 Literature review

2.1 Ocean Simulation

Two simulators, which are primarily aimed at similar use-cases to PPC, are OceanParcels [5] and TRACMASS [3]. OceanParcels focuses on high configurability achieved through the use of its `Kernels` system, where particle advection logic can be specified by user-delivered snippets of Python code. This code is compiled to C code using Just-In-Time (JIT) compilation to minimize the performance penalty of interpreted code [2]. This system is primarily restricted on two fronts: snippets must adhere to the OceanParcels-defined function signatures, and are restricted to the standard Python library (and select extra functionality) [2], [6].
TRACMASS, being written in Fortran, a compiled language, enables more performant simulations. However, it is less configurable than OceanParcels [4]. Setting up a customised TRACMASS experiment involves editing exposed parts of the code, and even then, the user is limited in several aspects. For example, TRACMASS absolutely requires the use of its `killzones` [7]. Additionally, it calculates particle advection using an analytical model. Unlike explicit numerical schemes such as the forward-Euler or 4th-order

Runge-Kutta (RK4) methods [8], this model supports exact and stable integration in both time directions, allowing the tracing of the origin of particles [3].

2.2 Language interoperability

For ease of use, it is ideal if certain simulator functionality is exposed as an API in one or more higher-level programming languages. The preferred language among domain experts is increasingly Python, as seen in popular simulators like OceanParcels.
To provide this higher-level API for PPC, technology is required that can understand snippets of code written in interpreted languages and use them in a C++ executable format. This section compares examples of such technology, with a specific focus on Python-C++ interoperability.

2.2.1 LLVM

LLVM consists of a collection of modular and reusable compiler and toolchain technologies. It can be used to create a frontend for any programming language and a backend for any instruction set architecture.
LLVM is written in C++, with frontends available for a variety of languages including Fortran, Java, Rust, and Python [9]. LLVM operates at a low level, requiring developers to write wrappers around any code to be compiled. This fine-granular control enables highly performant wrapping when used properly [10], but requires careful handling, leading to increased tedium during development. This is especially apparent when compared to alternative solutions, which themselves build on LLVM. There is, however, a distinct advantage in using LLVM over most derivative tools: LLVM is built with performance in mind, and making use of it enables easy compile-time optimisations [11].

2.2.2 Codon

Codon is a framework which compiles Python to native machine code. Codon supports most of the Python standard library, as well as a fully native NumPy implementation [12]. Codon is built on LLVM, and supports inline LLVM intermediate representation (IR) code written by the developer [13]. Writing such IR code directly greatly increases the granular control the developer has over the codebase, which can be a boon for code performance, but with the drawback of increased development tedium.
Comparing eight Python compilers, including Codon and Numba, Stoico et al. found that Codon performs best on the run-time metric [14]. Its documentation also describes speedup when compared to vanilla Python as being on the order of 10-100x or more. Unfortunately, Codon only compiles down to shared object files. These files are intended to be run as stand-alone programs[1], making interoperability with external C++ code extremely difficult.

2.2.3 Swig

Swig [15] is an established software development tool, first released in 1996. It connects C and C++ to other higher-level target programming languages, including Python. The typical use case for Swig is prototyping C/C++ code by generating 'glue code' between C/C++ interfaces and the target language. While Swig can support many C++ features, there are some notable limitations, including the lack of support for templates and inheritance. Swig, however, provides workarounds for these features: For templated C++

[1] or loaded by Python code exclusively

code to be wrapped by Swig, it must be instantiated explicitly. Inheritance at the C++ level is not wrapped; instead, Swig replicates the inheritance structure in proxy classes generated in the target programming language (that is the case for Python).

2.2.4 ShedSkin

ShedSkin [16] is a Python-to-C++ transpiler (translator & compiler) aiming to speed up statically typed Python code. It is unique among the compared solutions in that ShedSkin generates (and leaves on disk) human-legible C++ code.
The primary use case for ShedSkin is speeding up standalone numerical algorithms. Additionally, it allows writing 'skeleton code' (functions with empty bodies) in Python, then compiling this skeleton along with any other functions to C++. Native C++ code can then be manually inserted into the empty, compiled skeleton. ShedSkin supports classes, extension module generation, and C++ interoperability by using *type models*, making it a primary candidate for trans-lingual project development. As ShedSkin is in active development, there are some limitations. Notable among these is the lack of NumPy support.

3 Implementation

This section first discusses the architecture of PPC, then focuses on two distinct workflows that detail the setup for Python-C++ interoperability – Swig and ShedSkin.

3.1 Architecture

In the survey conducted by Hougee et al. [4], all the surveyed simulators followed a modular architecture. While each project structures its modules differently, they all emphasise keeping modules as independent of each other as possible. Independent modules promote extensibility since any required changes are confined within the module in which they need to be implemented.
Focusing exclusively on modularity typically reduces the program's overall performance. This is resolved in PPC by building with a modular nature where appropriate, but reducing the flexibility of different modules where performance is critical. The description is divided into two sections: (1) the core of the simulator and (2) the plugin system.

3.1.1 Core

The core of the simulator is implemented in C++. In the survey [4], the simulators written in Python prioritise readability and accessibility, whereas the ones written in C++ focus on performance. The PPC codebase focuses on a balance between performance and flexibility (for supported data types), which necessitates the extensive use of C++ templates, making readability a challenge for inexperienced C++ users.
The core consists of a central simulation class which contains and initialises all the modules of the simulator. The simulation class is exposed to the main entry-point of the program through a 'facade pattern' [17]. Internally, it drives the data input, runs the simulation loop, and handles the data output. The `config` class includes the external `tomlplusplus` library. It contains an interface to parse values from input configuration files, but no methods are provided to modify the configuration file. Since it is read-only, it is thread-safe; thus, a `config` object can be instantiated as a global variable.

3.1.2 Particle

The 'particle' module revolves around the `particleSet` abstract class. It contains reader and writer abstract classes for particle input and output (I/O), so that any subclassed readers and writers are compatible. This decision, similarly to OpenDrift's [18] Reader class approach, is made to enhance modularity. Some C++ I/O classes have already been implemented, and the structure enables future Python interoperability. The `particleSet` class is implemented in concrete subclasses, which determine the type of data structure used to store the particles. Currently, particle set classes are implemented using an array (with `std::vector`) and a linked list (with `std::list`). The linked list container is more suitable for simulations that require dynamically deleting and adding particles to the particle set [19].
The core particle class has two subclasses - `CppParticle` and `PyParticle` - which are combined via a `ParticleVariant` template alias using `std::variant`. This design enables compatibility with both native C++ and Python-interoperable code, while allowing easy extension with user-defined particle types. The particle class stores its data attributes in a `std::unorderedmap` of `std::any` to store various data types.

3.1.3 Kernel and Boundary Handler

The 'kernel' and 'boundary handler' modules are all similarly structured, with an abstract base class and concrete subclasses. Notably, both classes are added via PPC's plugin system (see below), which eliminates the possibility of class-level templating.
The 'kernel' module's dependencies are only tied to the advect function's signature, which is used for the user-defined plugins; thus, they do not affect the application core but only the code written by the user.
The `Kernel` class addresses this restraint by limiting any derived subclasses to a single data type, i.e. float *or* double. Additionally, `Kernel` is inherently coupled to the `Iterable`, `Field`, `Grid`, and `Boundary Handler` classes through its advection function; however, this reflects the fundamental algorithmic structure of an advection kernel, not a modularity flaw. The `BoundaryHandler` class is designed around the no-templating in the same way as the `Kernel`.
Furthermore, for increased flexibility, the `BoundaryHandler` defines a `handle` function signature for both a single particle and a particle set. It then depends on the specific use case which signature is more suitable.

3.1.4 Grid and Field

The `Grid` and `Field` classes manage field operations. This means they, along with the `FieldGridManager`, are responsible for loading and accessing field data. Additional logic is included to handle file path management and access.
Additionally, the field module utilises a configurable n-depth buffer. This system will *automatically* load (and store in memory) the defined number of time-steps, which reduces the cost of waiting for new data to be fetched and thereby reduces synchronisation time and overhead.
To avoid templating in the `Kernel` and retain its plugin system compatibility, the abstract `Grid` class must also avoid templates. It achieves this by defining both float and double versions of each function. It is then up to the user to decide whether to implement both or either option.

3.1.5 Algebra

Lastly, an 'algebra' module is implemented, defining custom classes designed to perform optimised (linear) algebra operations. At present, this module defines only the `Vec` class used for storing and manipulating an N-dimensional vector of an arbitrary data type. Special attention was given to the 3-dimensional vector, including a subclass defining additional operations (e.g. the cross product) which are only defined in the 3rd dimension[2]. Furthermore, for $N \leq 4$, a float data type `Vec` class can be sped-up by storing all values in a single `__m128` variable, which fits exactly four (32-bit) floats, allowing for highly-optimised data operations. Currently, this is implemented only for $N = 3$.

3.2 Plugins

To satisfy the requirement of runtime-exchangeable critical components, PPC utilises a plugin system. This system is responsible for dynamically loading (separately compiled) snippets of code, which determine certain behaviours of the system. Currently, the system supports plugins for the 'kernel', 'boundary handler', and particle I/O through the 'particle reader' and 'particle writer'.

Dynamic linking of plugins with the core application is achieved through the use of a shared base class; each plugin component provides (in the core) an abstract class. This abstract class defines the interface, through function signatures, by which the plugin is accessed by the core. The plugin implementation must then define the body of these functions, through which different runtime functionality is achieved.

This class-based approach provides the maximum amount of freedom to the user while outlining clearly how the core program interacts with plugins. It does, however, place two key limits on the system, as each plugin must: (1) adhere to the function signature of its base class, and (2) include factory methods for instantiation and destruction.

This second limitation is tightly coupled to the way dynamic linking is achieved. PPC makes use of a plugin-agnostic `DLLoader` class, which manages pointers to plugins from within the core program. For this dynamic loading to function, the compiled plugin object needs to provide entry- and exit points through which the loader creates and destroys plugin instances.

The loader makes use of the system calls for the appropriate platform[3] to procure instances of the plugin. Due to this, it is key that the plugin's factory functions are easily identifiable. This is not a given, as C++ functions are 'mangled' to provide unique names during the compilation process [20]. To circumvent this, the factory functions must be wrapped in the `extern "C"` directive, which does not otherwise impact their behaviour. A typical plugin factory can be found in appendix C, listing 1.

Which plugins to use is determined at run-time through the use of configuration keys. This configuration can also be used to initialise attributes inside the plugin - e.g. the used `dt` - as an alternative to hard-coding these; this further enhances usability, as the same plugin can be used for different simulation setups, without needing to be recompiled.

3.3 Swig

Swig enables domain experts to write advection kernel Python scripts that can be plugged into the simulation. Currently, the PPC build that uses Swig supports Swig-enabled Python plugins for the advection kernel.

[2] Technically, the cross product also exists in 7D, but that is not very helpful in this context.

[3] tested extensively on Linux/Unix, theoretically functional on Windows

The two methods for integrating C++ with Python are 'extending' and 'embedding'. The former gives Python access to C++, whereas the latter gives C++ access to Python. Swig allows both approaches, and PPC's implementation uses both. Swig gives Python access to the C++ interfaces by wrapping them, such that they can be imported into Python. On the other hand, C++ can run Python code by embedding the Python interpreter within C++. The Python being run is interpreted Python (not compiled nor transpiled), which negatively impacts performance.

The program flow starts in C++, where the simulation objects are initialised. The simulation initialises the Python interpreter before the simulation loop and keeps it alive until the simulation is completed. The main simulation loop runs in C++ and calls the advection kernel plugin written in Python every iteration. Shared pointers of C++ objects are passed as function arguments into the Python environment, where the Python code can call their methods to manipulate them. To generate C++ wrappers, Swig requires the user to create a `.i` interface file. It includes the C++ interfaces and Swig directives that provide additional features. The file is used to generate a `wrap.cxx` wrapper file, which gets compiled into a `.so` module. An additional Python file is generated, containing a class meant to act as a proxy between Python and the module. The Python 'proxy class' can be imported as a regular Python import into Python files and used as a regular Python class. Whenever methods of the class are called, the Python 'proxy class' redirects the call to the module containing the wrappers.

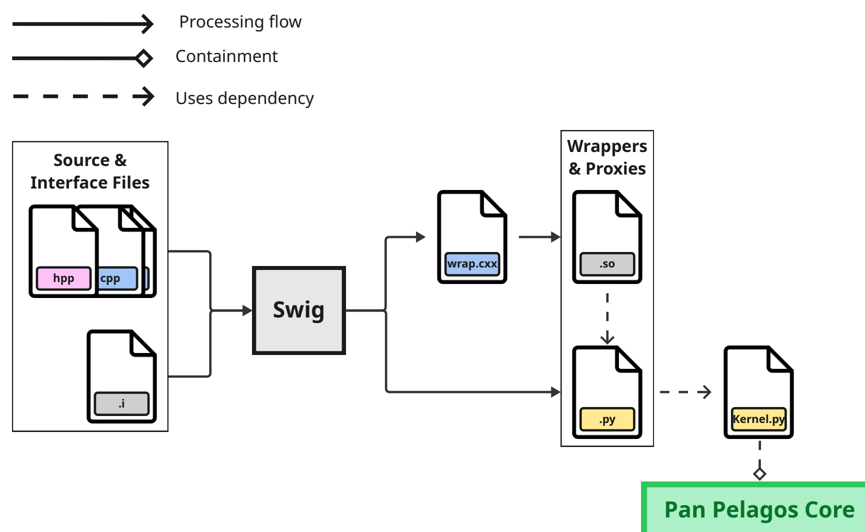


Figure 3.1: *Swig wrapping and embedding workflow.*

Swig embedding the Python interpreter means that the user can write plugins in pure Python, sparing them from having to do memory management. Additionally, most of the dynamic features of the language can still be used. Dynamic typing, dynamic code execution, and dynamic imports function as they should. Notably, 'duck typing', reflection and runtime object attribute extension all work on the classes wrapped by Swig. The only caveat is that the interface file of the class needs an additional Swig directive (`shadow`). The attributes added during the Python runtime to an object created in C++ only exist within Python.

The advection function signature of the kernel expects shared pointers of the C++ classes `BaseIterable`, `Field`, `Grid` and `BaseBoundaryHandler`. These classes have been

wrapped with Swig so that the internals of the parameters can be used in Python. The Vec3 class is also wrapped because many methods return Vec3 objects. The currently implemented advection kernels with Swig are a Forward Euler kernel and three versions of an RK4 kernel. The version called 'Swig Python loop' contains the loop that iterates over the particle set parameter directly inside the Python implementation. 'Swig single particle' receives one particle at a time, thus keeping the particle loop in C++. Both of these versions use a trilinear interpolation function implemented in Python, whereas 'Swig optimised' calls the C++ implementation.

3.4 ShedSkin

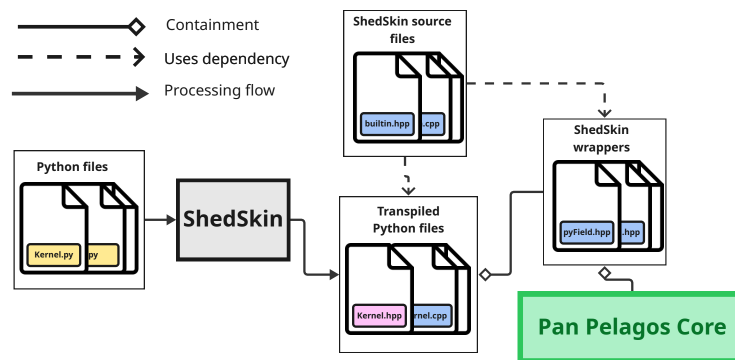


Figure 3.2: *ShedSkin transpilation and integration workflow.*

To utilise ShedSkin for enabling Python-based components inside PPC, one must stray from ShedSkin's intended use – speeding up mainly standalone algorithms. ShedSkin introduces wrappers for many components of the C++ standard library, ranging from low-level types, such as `int` and `float`, to complex containers, including `std::list` and `std::unordered_map`. These wrappers ensure that the generated code's functionality matches that of the original (Python) code. However, this additional level of abstraction necessitates the development of PPC-side 'ShedSkin wrappers' which act as a compatibility layer between the ShedSkin-generated code and the rest of PPC. This dependency is illustrated in Figure 3.2. Furthermore, we observe that the Python side of PPC has zero dependencies on the C++ side, allowing users to develop Python-side functionality (e.g., the kernel) without needing to examine any C++ code. With this setup, the following Python-side implementations are supported: `BoundaryHandler`, `Field`, `Grid`, `Kernel`, `Particle`, `Interpolation`, and `Vec`.

A current limitation of ShedSkin is that the Python-side components listed above must be used together, meaning that, for example, a Python-side 'kernel' cannot be used with a C++ 'boundary handler'. Similarly, the `PyParticle` variant must be selected. This constraint may be lifted in the future by utilising ShedSkin's support for skeleton implementations. A prime candidate for this approach is the performance-critical `Vec` class, which could be (re)implemented in native C++ to improve performance. ShedSkin-based components can be enabled/disabled by using the `ShedSkin` compilation flag. ShedSkin works alongside the plugin system, with the Python-side `Kernel` and `BoundaryHandler` classes being added as plugins, similar to the native C++ implementation of PPC.

Lastly, ShedSkin allows any statically written Python code to be compiled into an optimised Python library. Using this built-in feature, a proof-of-concept ShedSkin NumPy

implementation was developed and is distributed with PPC. Currently, only a limited subset of around 160 NumPy functions is supported; nevertheless, more functions can be added in the future using the same approach.

3.5 Paraview

PPC has Paraview state files that can be used to directly load a Paraview visualisation. While the current output format for particle positions for the simulation (VTK) is widely supported in other visualisation software, having pre-made Paraview state files removes the need to set up an entire visualisation scene and provides an out-of-the-box solution to visualising results. The state file is a Python file containing preset and user-configurable settings to launch a Paraview scene. The user can load from different files without having to make a new one for each new configuration.

4 Methods

4.1 Data

For development testing, two specific datasets were used: (1) a time-independent laminar flow, and (2) the Double-Gyre (DG) [21]. The DG dataset, a popular benchmark derived from the analytical Double-Gyre model [22], was also used to benchmark the software's performance and memory consumption. Specifically, a DG dataset with a spatial resolution of $539 \times 269 \times 28$, i.e. over 4 million vertices, was chosen.
Both datasets store a hydrodynamic 3D velocity field, discretised on a uniform Eulerian grid. The only assumption made about the input dataset is this Eulerian discretisation; thus, the software accepts any Eulerian dataset input. The above-described datasets are stored in the NetCDF format [23]. However, *any* input format is supported, provided that a corresponding reader subclass is implemented. Data can be loaded and stored with either 32-bit or 64-bit precision, depending on user preference.

4.2 Benchmarking

For performance testing, the same setup as in [24] was used. Particle tracing was simulated by advecting particle positions, modelled by the 3D Dirac-Delta function $\delta^3\left(\mathbf{r}-\mathbf{r}_p\right)$:

$$\frac{\partial \delta^3\left(\mathbf{r}-\mathbf{r}_p\right)}{\partial t} + \mathbf{u}(t;\mathbf{r}) \cdot \nabla \delta^3\left(\mathbf{r}-\mathbf{r}_p\right) = 0 \;\Rightarrow\; \frac{d\mathbf{r}_p}{dt} = \mathbf{u}(t;\mathbf{r}), \tag{4.1}$$

where $\mathbf{r}$ represents a point within the simulation domain, $\mathbf{r}_p$ is the position of a particle, and $\mathbf{u}(t;\mathbf{r})$ is the fluid velocity vector field at time t and position $\mathbf{r}$. The resulting ODE from eq. (4.1) was solved numerically using the RK4 method. This method was chosen to enable a meaningful comparison with the results in [24].
The execution time of a single advection step, i.e. advancing all particles by a single simulation timestep dt, and the memory usage were measured for multiple particle counts, ranging from 1,500 to 2,200,200 in exponential steps. A periodic boundary condition was applied to ensure the particle count remains constant throughout a measurement.
The time measurements were taken both by the internal CPU timer and a wall clock timer, utilising the `<time.h>` and `<chrono>` libraries, respectively, to allow for accurate evaluation. For memory usage measurements, to capture the physical memory, including stack and heap, used by the program, the number of resident pages was read from the system files `/proc/self/statm` and converted using the system page size.

5 Results

The following section presents the results of the benchmarking described in Section 4.2. The performance and memory usage measures are compared for different implementations of PCC, namely: standalone C++, integrated ShedSkin, and integrated Swig. Additionally, standalone ShedSkin and Python simulators are presented for reference.

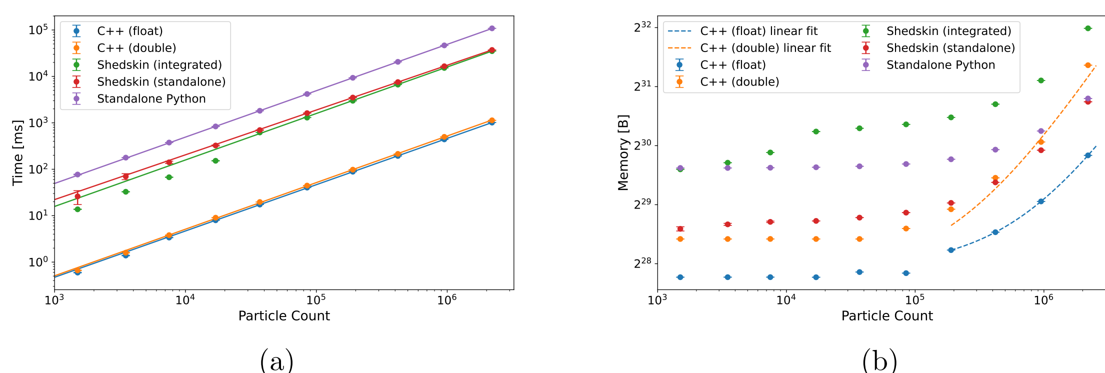
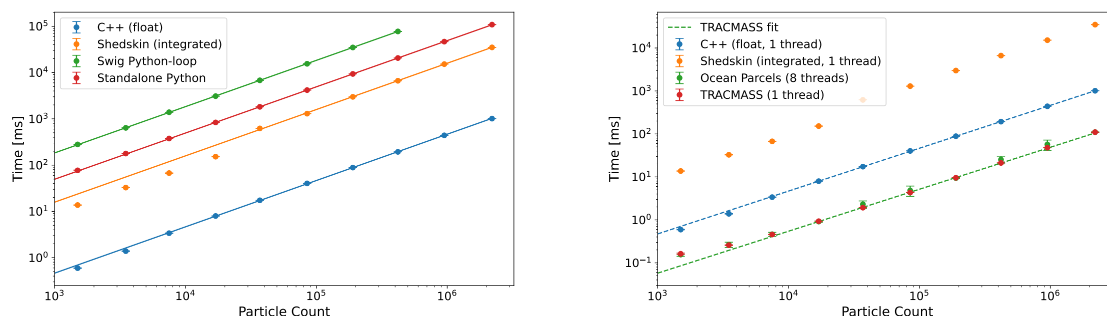


Figure 5.1: *CPU time* (a) *and memory consumption* (b) *comparisons of float and double precision of the pure C++ implementation vs the integrated and standalone ShedSkin implementation. A standalone Python-based simulator is added for reference.*

Figure 5.1 displays the benchmarking results between different C++ implementations and the ShedSkin-integrated implementation. In Figure 5.1a, the CPU time measurements are shown, displaying a linear dependency for all methods. Visibly, both the float and the double precision of the standalone C++ implementation outperform all the others, with minimal[4] difference between them. Between the integrated and standalone ShedSkin implementations, a slight difference is observed. The integrated implementation is 1.81 ± 0.03 times faster than the standalone equivalent. However, the difference becomes more negligible as the particle count increases. The standalone Python implementation performs the worst, being 33 ± 1 times slower than integrated ShedSkin and 104 ± 1 times slower than the standalone C++ implementation.
Figure 5.1b shows the average memory consumption throughout the particle advection process. Consumed memory remains constant for all implementations until $\sim 10^5$ particles, with the exception of integrated ShedSkin, which encounters an increase already at $\sim 10^4$ particles. Moreover, we observe that for all methods, the memory consumption is in the order of gigabytes (GB)[5]. Standalone C++ with float precision is the most memory-efficient implementation, requiring less than 1 GB for all measured particle counts. Conversely, the integrated ShedSkin implementation is the least memory-efficient, requiring over 4 GB of memory for a particle count of $2,200,200$, and is the only method less memory-efficient than the standalone Python implementation. Lastly, C++ with double precision is the second most memory efficient for particle counts smaller than $\sim 10^5$; however after this point its memory usage starts increasing with a steep slope of 985.3 ± 0.1 bytes per particle, in contrast to the float precision implementation which increases by 324.7 ± 0.1 bytes per particle.

[4] float-precision is 1.02 ± 0.01 times faster
[5] 1 GB $= 2^{30}$ B



(a) *The different methods vs a standalone Python-based simulator.*

(b) *Established Lagrangian simulation packages vs the different methods.*

Figure 5.2: *CPU time comparisons.*

In Figure 5.2a, the CPU time measurements for the different methods are shown. All four methods show a linear relation with the particle count. Standalone C++ again proves the fastest, followed by integrated ShedSkin, standalone Python, and lastly Swig. Notably, the comparable Swig implementation plotted in Figure 5.2a is 3.67 ± 0.02 times slower than standalone Python, 11.7 ± 0.1 times slower than the integrated ShedSkin implementation, and 398 ± 6 times slower than the C++ implementation.

Figure 5.2b compares the two most performant implementations of PPC with Ocean Parcels [5] and TRACMASS [3]. Notably, we see both Ocean Parcels and TRACMASS outperform the most efficient implementation of PPC by 7.1 ± 0.1 times.

6 Discussion

6.1 Performance & Memory Usage

The two different regimes in Figure 5.1b reflect whether memory usage is dominated by the stored field files (for particle counts below $\sim 10^5$) or by the particles themselves (for particle counts above $\sim 10^5$). A different regime threshold of $\sim 10^4$ is observed only for the least efficient integrated ShedSkin implementation. The reason for this difference is the combination of ShedSkin's integrated garbage collector with PPC's memory management system, which utilises C++'s (smart) pointers. In the current implementation, all particle-related memory must pass through both memory management systems. Even though the data is passed by reference in performance-critical sections, each memory system retains additional memory in case it is needed in the future.

Furthermore, a considerable difference in slopes in the particles-dominating region was observed in Figure 5.1b between the most efficient C++ float precision implementation of PPC and its double equivalent. Partially, this discrepancy is caused by the doubled memory requirements of double-precision (64-bit) compared to float-precision (32-bit). However, another significant contribution comes from the overhead of the flexible `std::unordered_map` particle attributes container. For the double-precision implementation, the attribute map is used to store all particles' attributes, whilst the float-precision implementation has been optimised, e.g. by extracting read & write access-heavy particle position into a dedicated variable inside the `Particle` class. Lastly, no meaningful difference in CPU time was observed between the C++ float and double implementations. This is likely because modern instruction set architectures (including the architecture used during testing) implement both float and double in hardware, which reduces the legacy

performance penalty of double precision.

Even the most performant C++ implementation of PPC is currently around 7 times slower than established state-of-the-art Lagrangian simulators, as seen in Figure 5.2b. [6] The performance deficiency is primarily caused by the employment of an Array-Of-Structures (AoS) instead of a Structure-Of-Arrays (SoA) for particle data layout [25]. An SoA layout improves memory access patterns by improving spatial locality, thereby reducing cache misses and improving performance. Additionally, implementing an SoA data layout is expected to increase the benefits of parallelisation in the future [26]. Lastly, the measurements in Figure 5.2b are in agreement with [26], which found performance differences between SoA and AoS layouts to be approximately an order of magnitude ($\sim 10\times$).

As for the version using Swig, B.2 shows that the most significant slowdown is caused by using the Python-implemented interpolation. It is expected that an interpreted Python implementation will be slower than C++; however, figure 5.2a shows that a stand-alone Python implementation without the use of Swig is faster. The Python implementation with Swig-wrapped objects was expected to be faster than stand-alone Python because the method calls would be executed using faster C++ code. Critically, Swig could not wrap some of the C++ methods that return pointers or references, such as the 'getter' method used to retrieve a `std::variant` reference from the set of attributes of a particle; thus, needing to retrieve a copy and set it back in the set once modified. The methods used to achieve flexibility in the core C++ implementation of the simulator hinder interoperability with Swig, resulting in a Python implementation that underperforms.

Integrating ShedSkin into PPC, partially replacing ShedSkin's upper abstraction layers with the PPC-side ShedSkin wrappers, resulted in a faster implementation than standalone ShedSkin, as seen in Figure 5.1a. This result highlights the potential for further performance improvements by developing a dedicated PPC-optimised fork of the ShedSkin source. By doing so, the performance of ShedSkin-integrated components can be improved by eliminating ShedSkin's garbage collector to decrease memory usage and reducing abstraction in performance-critical sections to increase performance.

6.2 Modularity

PPC's general architecture and code layout is designed with high modularity in mind; the least modular components involved in the simulation process are strictly required for operability (e.g. the `particleSet` and `Kernel` classes); however even these components have been designed for optimum flexibility, including multiple layers of abstraction to enable the widest possible variety of applications.
Aside from this general architecture, the implemented plugin system is as modular as realistically feasible; any more freedom on the user's side (e.g. by using raw data types over class-based objects) would have incurred an unacceptable performance penalty.
It should be mentioned that PPC's plugin system is loosely inspired by the OpenSceneGraph (OSG) system of the same name. The biggest similarity between OSG and PPC's plugin systems is their purpose: OSG uses plugins for file I/O. However, OSG plugins are mostly automatic; a `Registry` keeps track of available plugins, and automatically loads the appropriate plugin based on file extension [27].

[6] It should be noted that the OceanParcels measurements in Figure 5.2b are run on eight threads, whilst PPC uses only one.

6.3 Future Work

As PPC is intended to be the core of a larger simulator system, a multitude of avenues remain open for future work. First, performance; PPC currently runs as a single-threaded application. Yet, particle advection is often suited to a Single Instruction, Multiple Data (SIMD) parallelisation structure. Additionally, for use cases involving distributed computing, parallelisation utilising the message-passing paradigm could be implemented.
The plugin system represents another avenue for future expansion; additional components of the simulator system may be converted to use the plugin architecture, namely, the components related to particle storage and field functionality would benefit from this. The particle class itself could also be included in this expansion; the current implementation is highly flexible, but this flexibility comes at a performance penalty due to the overheads of the `std::any` datatype and the `std::unordered_map` used to store particle attributes. By converting the particle class into a plugin component, this flexibility could be exchanged for performance when necessary.
Optimising the Swig version of PPC could allow for an alternate build of the simulator that is less performant but easier to work with, thanks to Python's dynamic features. Domain experts would use this version in situations where the ease and time required to write a Kernel plugin are more important than the simulation's performance.
Utilising ShedSkin in a performance-focused project revealed apparent performance limitations due to its high level of abstraction. Based on an indirect comparison with [14], Codon is theoretically about 64% faster than ShedSkin. This suggests a potential future research direction: building a simulator entirely using Codon, bypassing native C++ altogether. While this results in an overall less flexible simulator, it provides more performant Python-C++ interoperability.

7 Conclusion

The PanPelagos-Core was developed, introduced, and discussed. A performant yet flexible architecture was designed by identifying performance-critical regions and partially reducing the flexibility in these regions only. Python-based components were integrated in two distinct ways: (1) using Swig, making sure to offload computation-heavy sections to C++ to reduce performance impact, and (2) via ShedSkin, transpiling Python code directly into native C++. Certain program components can be changed at runtime with minimal impact on usability and performance by utilising PanPelagos-Core's modular architecture and plugin system. In comparison with state-of-the-art Lagrangian simulators, PanPelagos-Core is currently approximately 7 times slower, primarily due to its Array-Of-Structures data layout. In a future release of PanPelagos-Core, this data layout will be exchanged for a Structure-Of-Arrays to match the performance of established Lagrangian simulators. Additionally, parallelisation will be added to further improve performance.
In conclusion, PanPelagos-Core is currently underperforming, but it remains faster than many alternative implementations. With an average memory usage of well below 8 GB, it can be efficiently used on any modern computer. Additionally, PanPelagos-Core is highly flexible on multiple fronts (both behaviorally and in specific component attributes), which broadens the array of potential use cases[7].

[7]For example, OceanParcels is limited exclusively to the zarr data format, whereas PanPelagos-Core allows any output format through user-written plugins

Acknowledgments

We would like to thank our supervisors, Dr. Christian Kehl and Dr. Julian Koellermeier, for their help and support.
Grammarly was used to proofread this document.

8 References

[1] S. Garnier, R. O. Murray, P. A. Gillibrand, A. Gallego, P. Robins, and M. Moriarty, *Particle tracking modelling in coastal marine environments: Recommended practices and performance limitations*, Ecological Modelling, vol. 501, p. 110 999, 2025.

[2] M. Lange and E. van Sebille, *Parcels v0. 9: Prototyping a lagrangian ocean analysis framework for the petascale age*, Geoscientific Model Development, vol. 10, no. 11, pp. 4175–4186, 2017.

[3] K. Döös, J. Kjellsson, and B. Jönsson, *Tracmass—a lagrangian trajectory model*, Preventive methods for coastal protection: Towards the use of ocean dynamics for pollution control, pp. 225–249, 2013.

[4] D. Hougee, B. Bruma, and A. Ray, *Survey of technical principles & performance limitations of eulerian-lagrangian ocean transport simulators*, 2025.

[5] P. Delandmeter and E. van Sebille, *The parcels v2.0 lagrangian framework: New field interpolation schemes*, Geoscientific Model Development, vol. 12, no. 8, pp. 3571–3584, 2019. DOI: 10.5194/gmd-12-3571-2019.

[6] *Parcels documentation*, https://docs.oceanparcels.org/, Accessed: 23-06-2025.

[7] *Tracmass documentation*, https://www.tracmass.org/docs.html, Accessed: 25-06-2025.

[8] W. H. Press, *Numerical recipes 3rd edition: The art of scientific computing*. Cambridge university press, 2007, pp. 907–915, ISBN: 9780521880688.

[9] *Llvm documentation*, https://llvm.org/, Accessed: 24-06-2025.

[10] C. A. Lattner, *Llvm: An infrastructure for multi-stage optimization*, 2002.

[11] C. Lattner and V. Adve, *Llvm: A compilation framework for lifelong program analysis & transformation*, in *International symposium on code generation and optimization, 2004. CGO 2004.*, IEEE, 2004, pp. 75–86.

[12] *Codon documentation*, https://docs.exaloop.io/codon, Accessed: 20-06-2025.

[13] A. Shajii, G. Ramirez, H. Smajlović, *et al.*, *Codon: A compiler for high-performance pythonic applications and dsls*, in *Proceedings of the 32nd ACM SIGPLAN International Conference on Compiler Construction*, 2023, pp. 191–202.

[14] V. Stoico, A. C. Dragomir, and P. Lago, *An empirical study on the performance and energy usage of compiled python code*, arXiv preprint arXiv:2505.02346, 2025.

[15] *Swig documentation*, https://www.swig.org/, Accessed: 24-06-2025.

[16] M. Dufour, *Shed skin: An optimizing python-to-c++ compiler*, Delft University of Technology, 2006.

[17] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1994, ISBN: 978-0201633610.

[18] *Opendrift documentation*, https://opendrift.github.io/, Accessed: 30-06-2025.

[19] C. Kehl, E. van Sebille, and A. Gibson, *Speeding up python-based lagrangian fluid-flow particle simulations via dynamic collection data structures*, 2021. arXiv: 2105.00057.

[20] I. C. ABI, *C++ abi: Name mangling*, Accessed: 2025-06-26, 2023. [Online]. Available: https://itanium-cxx-abi.github.io/cxx-abi/abi.html#mangling.

[21] E. Simonnet, M. Ghil, and H. Dijkstra, *Homoclinic bifurcations in the quasi-geostrophic double-gyre circulation*, English, Journal of Marine Research, vol. 63, no. 5, pp. 931–956, 2005, ISSN: 0022-2402. DOI: 10.1357/002224005774464210.

[22] S. C. Shadden, F. Lekien, and J. E. Marsden, *Definition and properties of Lagrangian coherent structures from finite-time Lyapunov exponents in two-dimensional aperiodic flows*. 2005, vol. 212, pp. 291–292. DOI: 10.1016/j.physd.2005.10.007.

[23] R. K. Rew and G. P. Davis, *Netcdf: An interface for scientific data access*, IEEE Computer Graphics and Applications, vol. 10, no. 4, pp. 76–82, 1990. DOI: 10.1109/38.56302.

[24] M. Opat, C. Kehl, and J. Koellermeier, *Lagrangian fluid transport simulation using mobile devices*, 2025. DOI: 10.2139/ssrn.5249985.

[25] C. Kehl, P. Nooteboom, M. Kaandorp, and E. Sebille, *Efficiently simulating lagrangian particles in large-scale ocean flows – data structures and their impact on geophysical applications*, Oct. 2022. DOI: 10.31223/X5BM0Q.

[26] H. Homann and F. Laenen, *Soax: A generic c++ structure of arrays for handling particles in hpc codes*, Computer Physics Communications, vol. 224, pp. 325–332, Mar. 2018, ISSN: 0010-4655. DOI: 10.1016/j.cpc.2017.11.015. [Online]. Available: http://dx.doi.org/10.1016/j.cpc.2017.11.015.

[27] M. Paul, *Openscenegraph quick start guide*, CGSDC, June, 2007.

Appendix A

Source Code

The source code for the Pan-Pelagos Core can be found here: https://gitlab.com/pan-pelagos_core/pan-pelagos

The raw data, the processed data, and the data processing code can all be found here: https://gitlab.com/pan-pelagos_core/pan-pelagos/-/tree/bench?ref_type=heads

The codebase for PPC is provided with extended documentation through Doxygen, and an additional 'how-to-use' guide is included to help users familiarise themselves with the plugin system. The documentation can be found at: https://rug-vis.github.io/PanPelagos-docs/

Appendix B

Array vs Linked List Measurements

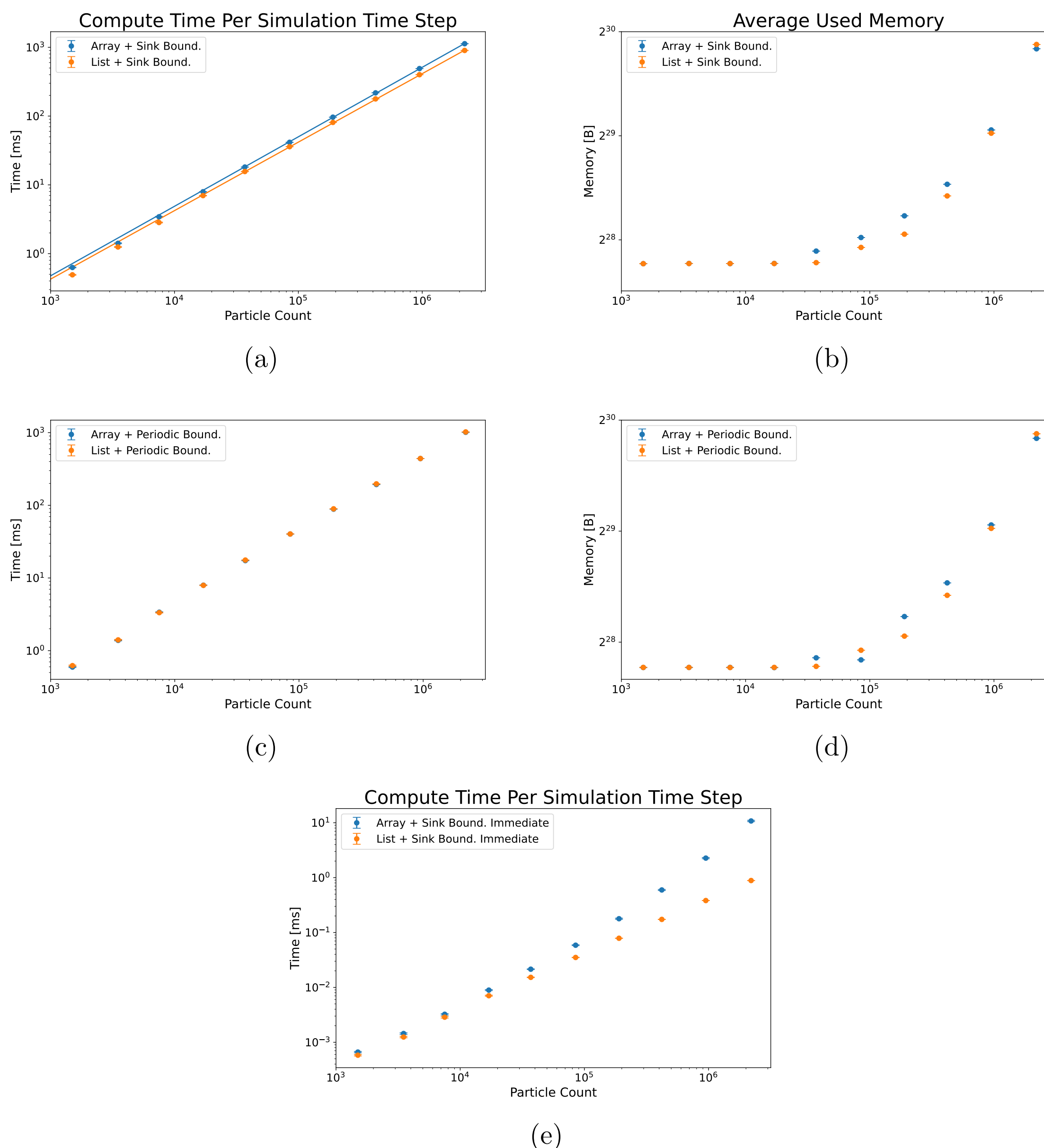


Figure B.1: *CPU time and memory consumption of an array container vs a linked list container compared for a sink boundary and periodic boundary conditions.*

The results in Figures B.1b, B.1c, and B.1d are near-identical for both the contiguous array and the linked list containers. Similarly, in Figure B.1a, no statistically significant difference[8] is observed between the two containers, even for the sink boundary, where particles are being dynamically removed. In these measurements, the particles are first marked as stale and deleted later in batches, which increases the efficiency of the array container. However, as shown in Figure B.1e, if we remove the particles immediately as they exit the simulation domain, the performance of the array container deteriorates drastically. The array container exhibits non-linear behaviour due to $\mathcal{O}(n)$ array element

[8] the linked list is 1.01 ± 0.02 times "faster" in this case

deletion occurring for every (removed) particle, resulting in $\mathcal{O}(n^2)$ overall time complexity, where n represents the array size, i.e. number of particles stored [19].

Swig Methods Measurements

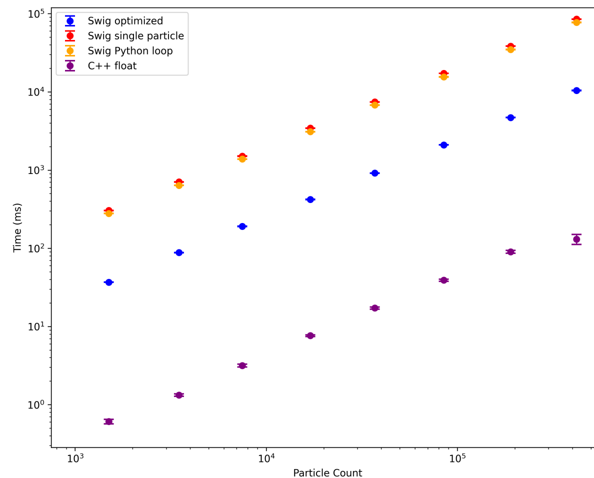


Figure B.2: *CPU time comparison between C++ and Swig.*

In Figure B.2, the CPU time measurements for the different versions of Swig's Runge-Kutta 4th-order kernel implementation are shown. All versions of Swig are slower than the C++ implementation. Calling the kernel one particle at a time is slower than looping through the particles all at once, and using the interpolation implemented in C++ gives the greatest speed-up.

In Swig's performance comparison B.2, the difference between the two unoptimised versions is the trade-off between running a for loop in Python and passing each particle object one by one. The results show that converting the C++ particle object to Python and passing it as an argument causes greater slowdown. All the simulation objects that are created in C++ and passed to Python need to be converted into `PyObjects` using the Python API. The `baseIterable` object needs to be converted only once and can be passed as an argument at every time step. Currently, the field objects for the current and next time step also need to be converted every time step; thus, the performance can be improved further.

Appendix C

Plugin factory functions

Listing 1: *factory.cpp for the RK4Kernel*

```
#include "rk4Kernel.hpp"

extern "C" {
RK4Kernel* allocator() {
  return new RK4Kernel();
}

void deleter(RK4Kernel* ptr) {
  delete ptr;
}
}
```

System Architecture Diagrams

Figure C.1: *System Architecture Diagram*