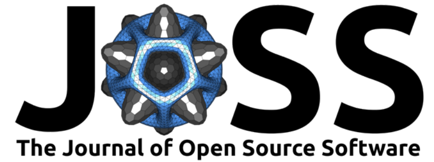


GridFormat: header-only C++-library for grid file I/O

Dennis Gläser [1¶], **Timo Koch** [2], **and Bernd Flemisch** [1]

1 University of Stuttgart, Germany **2** University of Oslo, Norway ¶ Corresponding author

DOI: 10.21105/joss.05778

Software

- Review
- Repository
- Archive

Editor: Daniel S. Katz

Reviewers:

- @lukeolson
- @IgorBaratta

Submitted: 15 August 2023
Published: 16 October 2023

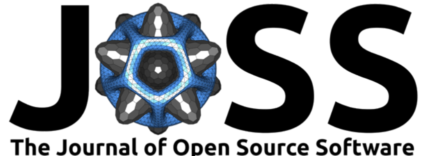
License
Authors of papers retain copyright and release the work under a Creative Commons Attribution 4.0 International License (CC BY 4.0).

Summary

Numerical simulations play a crucial role in various research domains including mathematics, physics, and engineering. Such simulations typically involve solving a set of model equations that describe the physical system under investigation. To find an approximate solution to these equations on a given domain geometry and with specific boundary conditions, the domain is usually discretized into a grid composed of points and cells, on which discretization schemes such as finite differences, finite volumes, or finite elements are then employed. This process yields a discrete solution defined at specific grid positions, which, depending on the scheme, can be interpolated over the entire domain using its basis functions. Due to the high computational demand of such simulations, developers often implement simulation codes in performant C++ and leverage distributed-memory parallelism through `MPI`, the *Message Passing Interface* (Clarke et al., 1994; *MPI Forum*, 2023), to run them on large high-performance computing systems.

Visualization plays a fundamental role in analyzing numerical results, and one widely-used visualization tool in research is `ParaView` (Ahrens et al., 2005; *ParaView*, 2023), which is based on `VTK`, the *Visualization Toolkit* (Schroeder et al., 2006; *The Visualization Toolkit*, 2023). `ParaView` can read results from a wide range of file formats, with the VTK file formats being among the most popular. To visualize simulation results with `ParaView`, researchers need to write their data into one of the supported file formats. Users of existing simulation frameworks, such as `Dune` (Bastian et al., 2008, 2021), `Dumux` (Flemisch et al., 2011; Koch et al., 2021), `Deal.II` (Arndt et al., 2022), `FEniCS` (A. Logg, 2012; *FEniCS*, 2023) or `MFEM` (Anderson et al., 2021; *MFEM*, 2023), can usually export their results into some standard file formats. However, they are limited to those formats that are supported by the framework. Reusing another framework's I/O functionality is generally challenging, at least without runtime and memory overhead due to data conversions, since the implementation is typically tailored to its specific data structures. As a consequence, the work of implementing I/O into standard file formats is currently repeated in every framework and remains inaccessible for researchers developing new simulation frameworks or other research codes relying on I/O for visualization.

To address this issue, `GridFormat` aims to provide an easy-to-use and framework-agnostic API for reading from and writing to a variety of grid file formats. By utilizing generic programming with C++ templates and traits, `GridFormat` is data-structure agnostic and allows developers to achieve full interoperability with their data structures by implementing a small number of trait classes (see discussion below). Users of both simulation frameworks and self-written small codes can write grid-based data into standard file formats with minimal effort and without significant runtime or memory overhead. `GridFormat` comes with out-of-the-box support for data structures of several widely-used frameworks, namely `Dune`, `Deal.II`, `FenicsX`, `MFEM`, and `CGAL` (*CGAL*, 2023; The CGAL Project, 2023).



Statement of Need

`GridFormat` addresses the issue of duplicate implementation effort for I/O across different simulation frameworks. By utilizing `GridFormat` as a backend for visualization file output, framework developers can easily provide their users with access to additional file formats. Moreover, instead of implementing support for new formats within the framework, developers can integrate them into `GridFormat`, thereby making them available to all other frameworks that use `GridFormat`. In addition to benefiting framework developers and users, the generic implementation of `GridFormat` also serves researchers with framework-independent smaller simulation codes.

Three key requirements govern the design of `GridFormat`: seamless integration, minimal runtime and memory overhead, and support for `MPI`. Given that C++ is widely used in grid-based simulation codes for performance reasons, we selected C++ as the programming language such that `GridFormat` can be adopted and used natively. It is lightweight, header-only, free of dependencies (unless opt-in features such as HDF5 output is desired), and supports CMake (*CMake*, 2023) features that allow for automatic integration of `GridFormat` in downstream projects.

A comparable project in Python is `meshio` (Schlömer, 2022), which supports reading from and writing to a wide range of grid file formats. However, accessing it from within simulators written in C++ would introduce an undesirable performance penalty, as well as memory overhead, since `meshio` operates on an internal mesh representation that users have to convert their data into. `Dune` users can employ `dune-vtk` (Praetorius, 2019), which supports I/O for a number VTK-XML file formats and flavours, however, its implementation is strongly coupled to the `dune-grid` interface and can therefore not be easily reused in other contexts. To the best of our knowledge, a framework-independent solution that fulfills the above-mentioned requirements does not exist.

Concept

Following the distinct VTK-XML file formats, `GridFormat` supports four different *grid concepts*: `ImageGrid`, `RectilinearGrid`, `StructuredGrid`, and `UnstructuredGrid`. While the latter is fully generic, the first three assume that the grid has a structured topology. A known structured topology makes it obsolete to define cell geometries and grid connectivity, and formats designed for such grids can therefore store the grid in a space-efficient manner. An overview of the different types of grids is shown in the image below, and a more detailed discussion can be found in the `GridFormat documentation`.

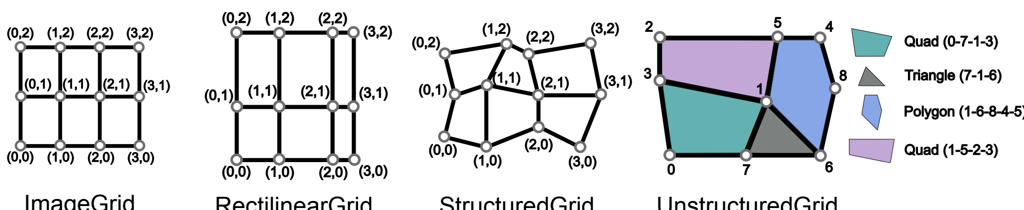
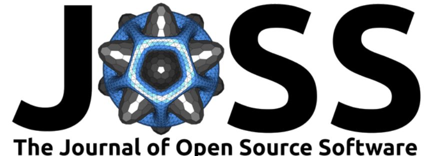


Figure 1: Overview over the grid concepts supported in `GridFormat`. While `UnstructuredGrids` are fully general, the first three have a structured topology.

`GridFormat` uses a traits (or meta-function) mechanism to operate on user-given grid types, and to identify which concept a given grid models. As a motivating example, consider the following function template:

```
template<typename Grid>
void do_something_on_a_grid(const Grid& grid) {
```



```
    for (const auto& cell : grid.cells()) {
        // ...
    }
}
```

In the function body, we iterate over all cells of the given grid by calling the `cells` method. This limits the usability of this function to grid types that fulfill such an interface. One could wrap the grid in an adapter that exposes the required interface method. However, this can become cumbersome, especially if there are certain requirements on the cell type in the iterated range. An alternative is to use traits, which allows writing the function generically, accepting any instance of a grid type that the `Cells` trait class template is specialized for (by using (partial) template specialization):

```
namespace Traits { template<typename Grid> struct Cells; }

template<typename Grid>
void do_something_on_a_grid(const Grid& grid) {
    for (const auto& cell : Traits::Cells<Grid>::get(grid)) {
        // ...
    }
}
```

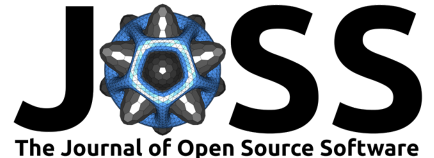
Instead of calling a function on `grid` directly, it is accessed via `Cells`, which can be specialized for any type. If such specialization exists, `do_something_on_a_grid` is invocable with an instance of type `Grid` directly, without the need for wrappers or adapters. Using C++-20 concepts, `GridFormat` can check at compile-time if a user grid specializes all required traits correctly. Error messages emitted by the compiler indicate which trait specializations are missing or incorrect. The traits mechanism makes the `GridFormat` library fully extensible: users can achieve compatibility with their concrete grid type by specializing the required traits within *their* code base, without having to change any code in `GridFormat`. Moreover, `GridFormat` comes with predefined traits for `Dune`, `FenicsX`, `Deal.II`, `MFEM` and `CGAL` such that users of these frameworks can directly use `GridFormat` without any implementation effort.

Note that each of the above-mentioned grid concepts requires the user to specialize a certain subset of traits. For instance, to determine the connectivity of an unstructured grid, `GridFormat` needs to know which points are embedded in a given grid cell. The information is not required for writing structured grids into structured grid file formats. An overview of which traits are required for which grid concept can be found in the `GridFormat` documentation.

The traits are required for writing out grids and associated data, and are not needed when using `GridFormat` to read data from grid files. `GridFormat` provides access to the data as specified by the file format, however, these specifications may not be sufficient in all applications. For instance, to fully instantiate a simulator for parallel computations, information on the grid entities shared by different processes is usually required. Since these requirements are simulator-specific, any further processing has to be done manually by the user and for their data structures. The recommended way to deal with this issue is to add any information required for reinstantiation as data fields to the output. This way, it is readily available when reading the file. For information on how to use these features, we refer to the API documentation and the examples.

Acknowledgements

The authors would like to thank the Federal Government and the Heads of Government of the Länder, as well as the Joint Science Conference (GWK), for their funding and support within the framework of the NFDI4Ing consortium. Funded by the German Research Foundation (DFG) - project number 442146713. TK acknowledges funding from the European Union's



Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 801133.

A. Logg, G. N. W. et al, K.-A. Mardal. (2012). *Automated solution of differential equations by the finite element method*. Springer. https://doi.org/10.1007/978-3-642-23099-8

Ahrens, J., Geveci, B., & Law, C. (2005). ParaView: An end-user tool for large-data visualization. *The Visualization Handbook*. https://doi.org/10.1016/B978-012387582-2/50038-1

Anderson, R., Andrej, J., Barker, A., Bramwell, J., Camier, J.-S., Cerveny, J., Dobrev, V., Dudouit, Y., Fisher, A., Kolev, Tz., Pazner, W., Stowell, M., Tomov, V., Akkerman, I., Dahm, J., Medina, D., & Zampini, S. (2021). MFEM: A modular finite element methods library. *Computers & Mathematics with Applications*, *81*, 42–74. https://doi.org/10.1016/j.camwa.2020.06.009

Arndt, D., Bangerth, W., Feder, M., Fehling, M., Gassmöller, R., Heister, T., Heltai, L., Kronbichler, M., Maier, M., Munch, P., Pelteret, J.-P., Sticko, S., Turcksin, B., & Wells, D. (2022). The deal.II library, version 9.4. *Journal of Numerical Mathematics*, *30*(3), 231–246. https://doi.org/10.1515/jnma-2022-0054

Bastian, P., Blatt, M., Dedner, A., Dreier, N.-A., Engwer, C., Fritze, R., Gräser, C., Grüninger, C., Kempf, D., Klöfkorn, R., Ohlberger, M., & Sander, O. (2021). The Dune framework: Basic concepts and recent developments. *Computers & Mathematics with Applications*, *81*, 75–112. https://doi.org/10.1016/j.camwa.2020.06.007

Bastian, P., Blatt, M., Dedner, A., Engwer, C., Klöfkorn, R., Kornhuber, R., Ohlberger, M., & Sander, O. (2008). A generic grid interface for parallel and adaptive scientific computing. Part II: Implementation and tests in DUNE. *Computing*, *82*(2), 121–138. https://doi.org/10.1007/s00607-008-0004-9

CGAL. (2023). Website: https://www.cgal.org/, code repository: https://github.com/CGAL/cgal.

Clarke, L., Glendinning, I., & Hempel, R. (1994). The MPI message passing interface standard. In K. M. Decker & R. M. Rehmann (Eds.), *Programming environments for massively parallel distributed systems* (pp. 213–218). Birkhäuser Basel. ISBN: 978-3-0348-8534-8

CMake. (2023). Website: https://cmake.org/.

FEniCS. (2023). Website: https://fenicsproject.org/, code repository: https://github.com/FEniCS.

Flemisch, B., Darcis, M., Erbertseder, K., Faigle, B., Lauser, A., Mosthaf, K., Müthing, S., Nuske, P., Tatomir, A., Wolff, M., & Helmig, R. (2011). DuMux: DUNE for multi-{phase, component, scale, physics, …} flow and transport in porous media. *Advances in Water Resources*, *34*(9), 1102–1112. https://doi.org/10.1016/j.advwatres.2011.03.007

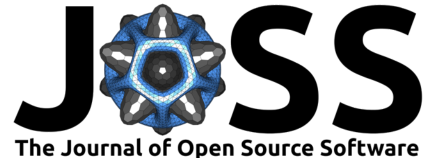
Koch, T., Gläser, D., Weishaupt, K., & others. (2021). DuMu^x 3 - an open-source simulator for solving flow and transport problems in porous media with a focus on model coupling. *Computers & Mathematics with Applications*, *81*, 423–443. https://doi.org/10.1016/j.camwa.2020.02.012

MFEM. (2023). Website: https://mfem.org/, code repository: https://github.com/mfem/mfem.

MPI Forum. (2023). Website: https://www.mpi-forum.org/docs/.

ParaView. (2023). Website: https://www.paraview.org/.

Praetorius, S. (2019). *Dune-vtk*. Published on the Dune GitLab server at https://gitlab.dune-project.org/extensions/dune-vtk.



Schlömer, N. (2022). *MeshIO: Input/output for many mesh formats*. Published on GitHub https://github.com/nschloe/meshio and also accessible via Software Heritage Permalink.

Schroeder, W., Martin, K., & Lorensen, B. (2006). *The visualization toolkit (4th ed.)*. Kitware. ISBN: 978-1-930934-19-1

The CGAL Project. (2023). *CGAL user and reference manual* (5.5.2 ed.). CGAL Editorial Board. https://doc.cgal.org/5.5.2/Manual/packages.html

The visualization toolkit. (2023). Website: https://vtk.org/.