The Journal of Open Source Software

fieldcompare: A Python package for regression testing simulation results

Dennis Gläser [1¶], **Timo Koch** [2], **Sören Peters** [3], **Sven Marcus** [3], and **Bernd Flemisch** [1]

1 University of Stuttgart, Germany **2** University of Oslo, Norway **3** Technical University Braunschweig, Germany ¶ Corresponding author

DOI: 10.21105/joss.04905

Software

- Review
- Repository
- Archive

Editor: Daniel S. Katz

Reviewers:

- @idoby
- @WilliamJamieson

Submitted: 31 October 2022
Published: 31 January 2023

Summary

In various research areas such as engineering, physics, and mathematics, numerical simulations play an important role. A number of research software simulation frameworks have been established, for instance, `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), and `VirtualFluids` (Kutscher et al., 2022). Numerical software typically has a high inherent complexity as it aims at solving complex physical model equations by using advanced mathematical methods for solving partial differential equations. Beyond this, the model equations often involve parameters that are described by means of empirical constitutive relationships. Thus, a numerical simulation usually brings together various software components: for the domain discretization, the discretization method for the equations, the physics, and a non-linear and/or linear solver to obtain a solution for the discretized equations.

While each of these components can be unit tested, it is important to have system tests that verify that a particular type of simulation can be carried out successfully. By *successful* we mean here that the simulation produces the *correct* results. As sufficiently complex problems often lack analytical solutions, determining correctness of numerical simulations poses a significant challenge. In the absence of an analytical solution, a common strategy is to use a trusted reference for comparison (e.g., data measured in experiments or results from previous publications). From the perspective of software quality assurance, it suffices to define a reference result as the *correct* one and continuously verify that the code still reproduces it. In numerical software, such regression tests play a vital role at the level of system tests (Kempf & Koch, 2017). They make sure that developers notice when a certain change to the code affects the results produced by the simulations. Whether the new results are better or worse has to be decided by the developers, and in the case of the former, the reference results may be updated.

In order to carry out regression tests, one must be able to detect *significant* deviations between newly-computed and reference results. What a *significant* deviation is has to be decided by the developers as well, and adequate tolerances have to be chosen that are big enough to avoid false negatives from machine precision issues, but small enough to ensure that physically relevant deviations in the results are detected. Some numerical software packages as, for instance, `DUNE` and `DuMux` (Flemisch et al., 2011), provide mechanisms to detect such deviations. However, the functionality is not provided independent of the frameworks themselves and is therefore only available to their users. Besides this, only those mesh file formats that are used by the frameworks are supported. Very recently, `DuMux` incorporated `fieldcompare` into its test suite in place of its in-house solutions.

Statement of Need

`fieldcompare` provides a tool that allows for detecting deviations in simulation results given in a variety of file formats. It supports several VTK file formats (Schroeder et al., 2006) out-of-the-box, and a large number of further file formats are accessible via interoperability with `meshio` (Schlömer, 2022). However, `fieldcompare` is not restricted to mesh files, but can be used for any file format that contains data that fits into the concept of *fields* (see Figure 1). In principle, it can support file formats that represent collections of fields, where a field is something that has a name and an associated array of values. Currently, in addition to mesh files, support for DSV (delimiter-separated-values) files is provided; this is a widely-used format to store secondary data from simulations that one possibly wants to include in regression tests. The code is structured in a way that allows for easy integration of more file formats when needed.

A challenge with regression-testing simulation results is that the order of points and cells may change within an otherwise identical mesh. The numerical solution may be the same, but it is organized in a different order in the mesh file. To this end, `fieldcompare` uses the approach of Kempf & Koch (2017), allowing a user to sort the mesh by arranging the point coordinates and cell connectivity in a lexicographically ascending manner to get a unique representation. This avoids false negative regression tests when only the order of the mesh has changed.

Simulations often perform time integrations using a number of discrete time steps. To facilitate comparing the results of an entire time series, the command-line-interface (CLI) of `fieldcompare` offers the option to compare two folders with results. It searches both folders for matching file names and then performs file comparisons for each of the matching pairs.

Finally, to make regression testing in continuous integration pipelines as easy as possible, we developed a GitHub Action around `fieldcompare` (Gläser, 2022), that can be used by projects hosted on GitHub to perform regression tests in their workflows with minimal effort. The CLI of `fieldcompare` also supports exporting the results of a regression test run into the junit xml file format, which is supported by some continuous integration platforms as, for instance, GitLab CI. If the `junit` report is submitted as an `artifact` from a GitLab pipeline, the GitLab web interface nicely displays which field comparisons have failed, passed, or were skipped, without the need to scan the pipeline log. We register field comparisons via the `testcase` element of the junit file format, while a file comparison is registered as a `testsuite`, with the comparisons of all fields contained in the file as underlying `testcases`.

Concept

`fieldcompare` aims to provide a framework that can be used for comparing data structures that represent collections of *fields*. Such a collection exposes fields together with the *domain* on which they are defined. So far, our main focus has been on results of numerical simulations, where the domains are computational meshes and the fields are, for instance, the values of discrete numerical solutions on those meshes. In `fieldcompare`, such collections are represented by the `FieldData` protocol, and two implementations currently exist: `MeshFields` and `TabularFields`. The former represents numerical results, as discussed before, while the latter exposes tabular data, for instance, read from a DSV file.

Two instances of objects conforming to the `FieldData` protocol can then be compared using the `FieldDataComparison` class, which checks that the domains are equal and that a given `Predicate` evaluates to true for the values of those fields that have matching names. A `Predicate` is a *callable* that takes two value arrays and returns a `PredicateResult`, which is an object that can be converted to `bool`, while further exposing information such as detected violations, tolerances used, etc. Note that it is possible to provide different `Predicates` on a per-field basis, and per default, the array values are compared for fuzzy equality.

For an illustration of the concept, see Figure 1.

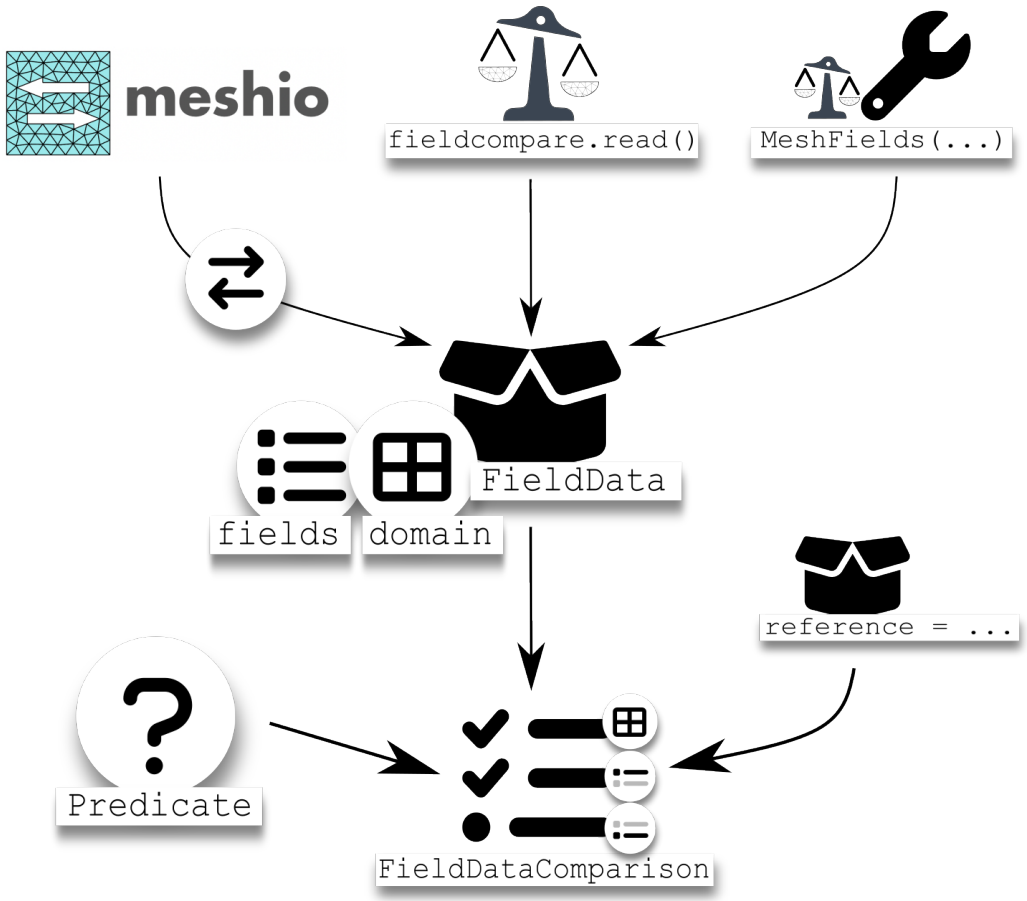

Figure 1: Basic concept: `FieldData` is either read from a file with the I/O facilities provided by fieldcompare, or constructed manually, or by conversion from a `meshio` mesh. Subsequently, it can be passed to the `FieldDataComparison` class to compare them against reference data, optionally using custom predicates to compare the individual field values.

The following code snippet illustrates how to read fields from two mesh files, sort the meshes to avoid false negatives from differing mesh ordering, and check the fields for customized fuzzy equality:

```
from fieldcompare import FieldDataComparator
from fieldcompare.io import read
from fieldcompare.mesh import sort
from fieldcompare.predicates import FuzzyEquality

result_fields = read("test/data/test_mesh.vtu")
reference_fields = read("test/data/test_mesh_permutated.vtu")
comparator = FieldDataComparator(result_fields, reference_fields)

# The `FieldDataComparator` has a default choice for predicates
# that it uses. But, we can (optionally) pass in a selector
# function (which will be invoked with the two fields to be
# compared) from which we can return the predicate we would
# like to use for a pair of fields:
predicate = FuzzyEquality(abs_tol=1e-6, rel_tol=1e-8)
```

```
result = comparator(predicate_selector=lambda _, __: predicate)

if not result and not result.domain_equality_check:
    print("Meshes not equal, retrying with sorted meshes...")
    result_fields = sort(result_fields)
    reference_fields = sort(reference_fields)
    result = FieldDataComparator(result_fields, reference_fields)(
        predicate_selector=lambda _, __: predicate
    )

print(f"Result is {bool(result)}")

# The result is a suite of field comparisons that we can loop over
# and print information on each comparison that was performed
for field_comp in result:
    print(f"Field name: {field_comp.name}")
    print(f"Status: {field_comp.status}")
    print(f"Predicate: {field_comp.predicate}")
    print(f"Report: {field_comp.report}")
```

Note that the API of `fieldcompare` also exposes a `MeshFieldsComparator` that can be used for fields defined on computational meshes. It automatically sorts the meshes in case they are not equal, making it possible to write a simple equality check for meshes using a custom predicate in a single instruction:

```
from fieldcompare.io import read
from fieldcompare.mesh import MeshFieldsComparator
from fieldcompare.predicates import FuzzyEquality

assert MeshFieldsComparator(
    source=read("test/data/test_mesh.vtu"),
    reference=read("test/data/test_mesh_permutated.vtu")
)(
    predicate_selector=lambda _, __: FuzzyEquality(
        abs_tol=1e-6, rel_tol=1e-8
    )
)
```

On the command line, the mesh is sorted by default (though this can be deactivated with a runtime flag). Results similar to the examples shown above can be obtained with the following commands:

```
# uses default tolerances
fieldcompare file test/data/test_mesh.vtu \
                  test/data/test_mesh_permutated.vtu

# specify tolerances
fieldcompare file test/data/test_mesh.vtu \
                  test/data/test_mesh_permutated.vtu \
                  --absolute-tolerance 1e-6 \
                  --relative-tolerance 1e-8

# specify the tolerances for a specific field only
fieldcompare file test/data/test_mesh.vtu \
                  test/data/test_mesh_permutated.vtu \
                  --absolute-tolerance function:1e-6 \
                  --relative-tolerance function:1e-8
```

The fuzzy details

Fuzziness is crucial when comparing fields of floating-point values, which are unlikely to be bit-wise identical to a reference solution when computed on different machines and/or after slight modifications to the code. In the code examples above, we have used the `FuzzyEquality` predicate of `fieldcompare`, which allows us define the absolute and relative tolerances to be used. However, specifying the tolerances is optional, so what are the defaults?

First of all, the `FuzzyEquality` predicate in `fieldcompare` evaluates to true if two given arrays have the same shape, and for each pair $(a,\ b)$ of scalar values in the arrays, the following condition holds:

$$|a - b| \leq max(\rho \cdot max(|a|, |b|), \epsilon).$$

Here, ρ and ϵ are the relative and absolute tolerance, respectively. Per default, $\epsilon = 0$, which means that each pair of scalars is tested by a relative criterion. The default relative tolerance depends on the data type and is chosen as the difference between 1.0 and the next smallest value larger than 1.0 as representable by the data type at hand (i.e., the unit of least precision). For 64-bit floating-point values following the IEEE-754 standard ("IEEE Standard for Binary Floating-Point Arithmetic" (1985)) this yields $\rho_{\text{default}} \approx 2.22 \cdot 10^{-16}$; that is, we require the values to match in ≈ 15 decimal digits. Generally, the tolerances should be carefully chosen for the context at hand, and the rather strict default values were selected to minimize the chances of false positives when using `fieldcompare` without any tolerance settings.

A common issue, in particular in numerical simulations, is that the values in a field may span several orders of magnitude, which may have a negative impact on the precision one can expect from the smaller values. As an example, consider a flow simulation with very high velocities in some parts of the domain, while in others the velocity is close to zero. A relative tolerance appropriate for the large velocities is unlikely to be suitable for the entire range of values.

For such scenarios, a suitable choice for the absolute tolerance ϵ comes into play, which can help avoid false negatives from comparing the velocities that are close to zero. According to the above formula, a switch to an absolute criterion occurs when the scaled relative tolerance falls below ϵ. In other words, ϵ defines a lower bound for the allowed difference between field values, which is illustrated in the figures below.

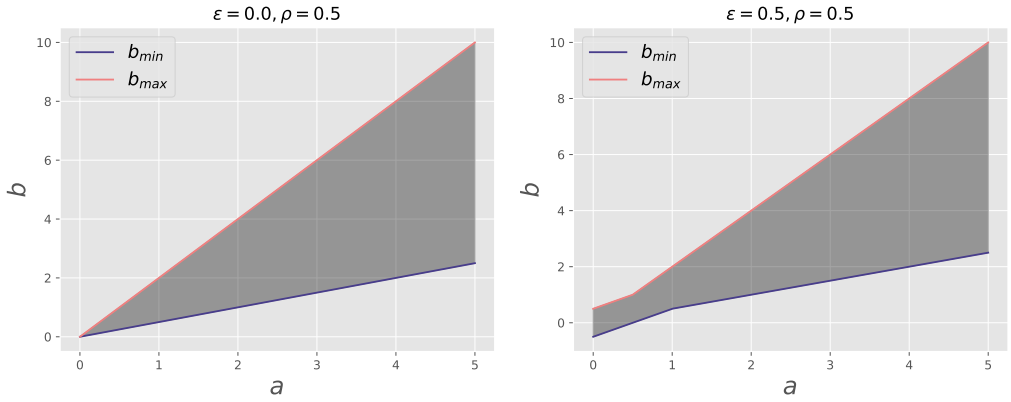

Figure 2: Visualization of the fuzzy equality check for different absolute and relative tolerances (chosen very high for illustrative purposes). For a given value a (x-axis), b_{min} and b_{max} visualize the lowest and highest values that evaluate fuzzy-equal to a, respectively. Consequently, all values that lie within this interval are considered fuzzy-equal to a (visualized by the dark grey area).

As can be seen, while for $\epsilon = 0$ the allowed difference between values goes down to zero as

$a \to 0$, a constant residual difference is allowed for small values of a in the case of $\epsilon > 0$. A suitable choice for ϵ depends on the fields to be compared, and when comparing a large number of fields, it can be cumbersome to define ϵ for all of them. We found that a useful heuristic is to define ϵ as a fraction of the maximum absolute value of both fields as an estimate for the precision that can be expected from the smaller values. Using the `fieldcompare` API, this can be achieved with the `ScaledTolerance` class, which is accepted by all interfaces receiving tolerances. A modified version of the previous example may look like this:

```
from fieldcompare.io import read
from fieldcompare.mesh import MeshFieldsComparator
from fieldcompare.predicates import FuzzyEquality, ScaledTolerance

assert MeshFieldsComparator(
    source=read("test/data/test_mesh.vtu"),
    reference=read("test/data/test_mesh_permutated.vtu")
)(
    predicate_selector=lambda _, __: FuzzyEquality(
        abs_tol=ScaledTolerance(base_tolerance=1e-12),
        rel_tol=1e-8
    )
)
```

With the above code, the absolute tolerance is computed for a pair of fields f_1 and f_2 via $\epsilon = max(mag(f_1), mag(f_2)) \cdot 10^{-12}$, where mag estimates the magnitude as the maximum absolute scalar value in a field. In the CLI, this functionality is exposed via the following syntax:

```
fieldcompare file test/data/test_mesh.vtu \
                  test/data/test_mesh_permutated.vtu \
                  -atol 1e-12*max
```

Fuzzy mesh comparison

As mentioned above, before the `FieldDataComparison` compares fields, it checks if the domains on which they are defined are *equal*. In the case of computational meshes, this check also has to be done in a fuzzy sense, since the point coordinates of the meshes are typically given as floating-point values. Moreover, the mesh sorting algorithms shown in the examples above also rely on fuzziness, and therefore, it is again crucial that suitable tolerances are defined. As a default, `fieldcompare` uses $\rho = 10^{-8}$ and $\epsilon = \tilde{x} * 10^{-8}$, where $\tilde{x}$ is the maximum occurring coordinate value in all points of the grid. Tolerances for domain equality checks can be set via the CLI as follows (again supporting the definition of scaled absolute tolerances):

```
fieldcompare file test/data/test_mesh.vtu \
                  test/data/test_mesh_permutated.vtu \
                  -atol domain:1e-12*max \
                  -rtol domain:1e-7
```

Programmatically, tolerances can be set directly on instances of a `Mesh`. As outlined before, the *domain* is a mesh when reading fields from mesh files, and the behavior of the above call to the CLI can be reproduced programmatically with the following code snippet:

```
from fieldcompare.io import read
from fieldcompare.mesh import MeshFieldsComparator
from fieldcompare.predicates import FuzzyEquality, ScaledTolerance

source = read("test/data/test_mesh.vtu")
reference = read("test/data/test_mesh_permutated.vtu")
source.domain.set_tolerances(
    abs_tol=ScaledTolerance(1e-12),
```

```
    rel_tol=1e-7
)
reference.domain.set_tolerances(
    abs_tol=ScaledTolerance(1e-12),
    rel_tol=1e-7
)
assert MeshFieldsComparator(source, reference)()
```

Applications

`fieldcompare` is currently used successfully by `VirtualFluids` (Kutscher et al., 2022) to continuously verify the results of the latest changes to the source code inside a continuous integration (CI) pipeline. To this end, `VirtualFluids` makes use of the command-line-interface of `fieldcompare`. Inside the CI pipeline, the trusted reference data is downloaded from a separate data source, in this case a Git repository. After that, the `fieldcompare` package is installed, and finally, a bash script compiles `VirtualFluids`, runs several simulations, and compares their result data with the respective reference files using the `fieldcompare` CLI. In case one of the test cases fails, the developers are notified by the continuous integration pipeline, therefore providing rapid feedback and allowing them to immediately start working on resolving the issue.

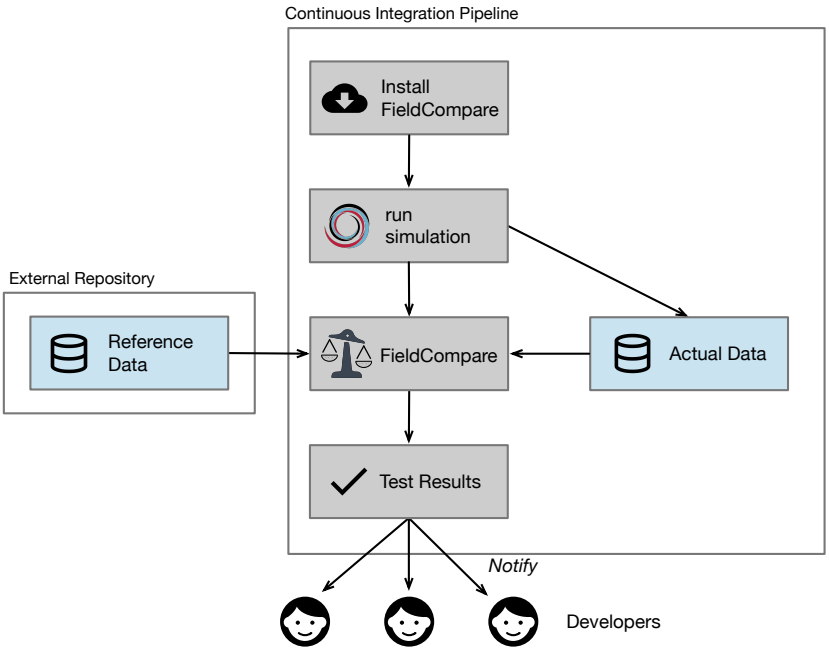

Figure 3: Overview of the VirtualFluids pipeline using fieldcompare

Another project in which `fieldcompare` is currently in use is `Argo` (Tolle & Maric, 2023), an `OpenFoam` (*OpenFOAM*, 2023) module for multiphase flow simulations. The continuous integration pipeline of `Argo` checks if the current state of the code is able to reproduce previously-published results by rerunning the cases of a paper (Tolle et al., 2022), fetching the published results from `Zenodo` (European Organization For Nuclear Research & OpenAIRE, 2013; Maric et al., 2021), and using `fieldcompare` to check for significant deviations.

The SURESOFT (Blech et al., 2022) project aims to establish a common usable methodology and infrastructure based on the concepts of continuous integration and containerization to approve the quality of research software, easing software delivery and ensuring long-term

sustainability and availability. One exemplary workflow of SURESOFT (Peters & Marcus, 2022) deploys a Singularity container on an HPC platform using `HPC-Rocket` (Marcus, 2022), runs a numerical simulation on the cluster, and validates the results with `fieldcompare`.

Finally, `DuMux` (Koch et al., 2021) uses the API of fieldcompare in its test suite, consisting of nearly 600 unit, integration, and system tests, the majority of which are regression tests. Calls to the `fieldcompare` API occur in the central test script of `DuMux`, where suitable default tolerances are defined, which are overridden by a few individual tests.

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. Additionally funding from the Deutsche Forschungsgemeinschaft (Project SURESOFT, LI 2970/1-1) and from the European Union's Horizon 2020 Research and Innovation programme under the Marie Skłodowska-Curie Actions Grant agreement No 801133 is gratefully acknowledged. Most of the icons in the presented figures have been obtained from Font Awesome by Dave Gandy - fontawesome.io

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

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

Blech, C., Dreyer, N., Friebel, M., Jacob, C., Shamil Jassim, M., Jehl, L., Kapitza, R., Krafczyk, M., Kürner, T., Langer, S. C., Linxweiler, J., Mahhouk, M., Marcus, S., Messadi, I., Peters, S., Pilawa, J.-M., Sreekumar, H. K., Strötgen, R., Stump, K., … Wolter, M. (2022). *SURESOFT: Towards sustainable research software*. https://doi.org/10.24355/dbbs.084-202210121528-0

European Organization For Nuclear Research, & OpenAIRE. (2013). *Zenodo*. CERN. https://doi.org/10.25495/7GXK-RD71

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

Gläser, D. (2022). GitHub action for FieldCompare. In *GitHub repository*. GitHub. https://github.com/dglaeser/fieldcompare-action

IEEE standard for binary floating-point arithmetic. (1985). *ANSI/IEEE Std 754-1985*, 1–20. https://doi.org/10.1109/IEEESTD.1985.82928

Kempf, D., & Koch, T. (2017). *System testing in scientific numerical software frameworks using the example of DUNE*. *5*, 151–168. https://doi.org/10.11588/ans.2017.1.27447

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

Kutscher, K., Schönherr, M., Geier, M., Marcus, S., Peters, S., Linxweiler, J., Krafczyk, M., & Wellmann, A. (2022). VirtualFluids. In *GitLab repository*. GitLab. https://git.rz.tu-bs.de/irmb/virtualfluids

Marcus, S. (2022). *hpc-rocket* (Version 0.3.1) [Computer software]. Zenodo. https://doi.org/10.5281/zenodo.7469695

Maric, T., Tolle, T., & Gruending, D. (2021). *Computing volume fractions and signed distances from arbitrary surfaces on unstructured meshes* (Version 2.0) [Data set]. Zenodo. https://doi.org/10.5281/zenodo.5603255

OpenFOAM. (2023). Website: https://www.openfoam.com/, code repository: https://develop.openfoam.com/Development/openfoam.

Peters, S., & Marcus, S. (2022). SURESOFT HPC workflow. In *GitLab repository*. GitLab. https://git.rz.tu-bs.de/soe.peters/suresoft-hpc-workflow

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*. Kitware.

Tolle, T., Gründing, D., Bothe, D., & Marić, T. (2022). triSurfaceImmersion: Computing volume fractions and signed distances from triangulated surfaces immersed in unstructured meshes. *Computer Physics Communications*, *273*, 108249. https://doi.org/10.1016/j.cpc.2021.108249

Tolle, T., & Maric, T. (2023). *argo*. Published on GitLab https://gitlab.com/leia-methods/argo and also accessible via Software Heritage Permalink.