The Journal of Open Source Software

libtrixi: an interface library for using Trixi.jl from C/C++/Fortran

Michael Schlottke-Lakemper [1,2¶], **Benedict Geihe** [3], **and Gregor J. Gassner** [3]

1 Applied and Computational Mathematics, RWTH Aachen University, Germany **2** High-Performance Computing Center Stuttgart (HLRS), University of Stuttgart, Germany **3** Department of Mathematics and Computer Science, University of Cologne, Germany ¶ Corresponding author

DOI: 10.xxxxxx/draft

Software

- Review
- Repository
- Archive

Editor: Open Journals

Reviewers:

- @openjournals

Submitted: 01 January 1970
Published: unpublished

Summary

The Julia programming language is able to natively call C and Fortran functions, which is widely used to utilize the functionality of existing, mature software libraries in Julia. In addition, Julia also provides an application programming interface (API) that allows calling Julia functions from C or Fortran programs. However, since the higher-level elements of the Julia language are not directly representable in C or Fortran, this direction of cross-language interoperability is much harder to realize and has not been used in practice so far.

With `libtrixi` (Schlottke-Lakemper et al., 2023) we thus present, to the best of our knowledge, the first software library to control complex Julia code from a main program written in a different language. Specifically, `libtrixi` provides an API to Trixi.jl (Schlottke-Lakemper, Winters, et al., 2021; Schlottke-Lakemper, Gassner, et al., 2021), a Julia package for adaptive numerical simulations of conservation laws. The API allows one to manage the entire simulation process, including setting up a simulation, advancing the numerical solution in time, and retrieving simulation data for further analysis. The main program may either be written in C/C++/Fortran or use any other language that can directly interact with the C/Fortran interface of `libtrixi`. With this approach, users can continue to use existing applications or legacy frameworks for the overall process control, while taking advantage of the modern, high-order discretization schemes implemented in Trixi.jl.

`Libtrixi` is developed and used as part of the research project "ADAPTEX" (see also section "Acknowledgments" below). Both `libtrixi` and Trixi.jl are available under the MIT license.

Statement of need

Numerical simulations of conservation laws are used to accurately predict many naturally occurring processes in various areas of physics, such as fluid flow, astrophysics, earth systems, or weather and climate modeling. These phenomena characteristically exhibit a broad range of spatial and temporal length scales, making it necessary to use finely resolved computational grids. Therefore, high-performance computing (HPC) techniques are required to render the numerical solution feasible on large-scale compute systems.

Consequently, many simulation tools are written in traditional HPC languages such as C, C++, or Fortran, e.g., deal.II (Arndt et al., 2021) or PETSc (Balay et al., 1997). These languages offer high computational performance, but often at the cost of being complex to learn and maintain. On the other hand, languages like Python, which are more amenable for rapid prototyping or less experienced users, are usually not fast enough to use without specialized

The Journal of Open Source Software

packages that utilize kernels written in a compiled language. For example, Python's well-known NumPy library (Harris et al., 2020) has its performance-critical code implemented in C.

The Julia programming language (Bezanson et al., 2017) aims to provide a new approach to scientific computing. It strives to combine convenience with performance by providing an accessible, high-level syntax together with fast, just-in-time-compiled execution (Churavy et al., 2022). Due to its native ability to call C or Fortran functions, in multi-language projects Julia often acts as a glue code between newly developed implementations written in Julia and existing libraries written in C/Fortran.

While there exist other numerical simulation codes in Julia, e.g., Gridap.jl (Badia & Verdugo, 2020) or Ferrite.jl (Carlsson et al., n.d.), none of them provide the ability to use them directly from another programming language. With `libtrixi`, we therefore enable new workflows by allowing scientists to connect established research codes to a modern numerical simulation package written in Julia. That is, a main program written in C/C++ or Fortran is able to execute a simulation set up in Julia with Trixi.jl without sacrifices in performance.

So far, this control direction, where a Julia package is managed from C/Fortran, has only been demonstrated in prototypes such as libcg[1] or libdiffusion[2]. To the best of our knowledge, however, it has not yet been employed in practical applications. Besides making Trixi.jl available to a wider scientific audience, `libtrixi` is thus a research project to investigate the efficacy of using Julia-based libraries in existing code environments, and can eventually serve as a blueprint for other similar efforts. Questions such as how to retain the flexibility of Julia while providing a traditional, fixed API, how to use system-local third-party libraries, or how to interact with Fortran are investigated and answered.

Technical overview

`Libtrixi` consists of three main parts:

- the Julia package LibTrixi.jl, which provides a traditional library API to Trixi.jl in Julia,
- a C API that exposes this Julia API as an ordinary, shared C library,
- Fortran bindings for the C API.

Figure 1 illustrates the general workflow: A main program written in C/C++/Fortran interacts with the public-facing C or Fortran API of `libtrixi`. The Fortran API is little more than a set of language bindings, with some extra code to handle the conversion between certain Fortran and C data types, e.g., strings or Boolean values. The C API is slightly more involved, providing functionality to initialize and eventually finalize the Julia runtime. During initialization, C function pointers are obtained from Julia to the relevant API functions implemented in LibTrixi.jl. Most of the C API then just forwards the API calls to these function pointers.

Finally, LibTrixi.jl is the actual library layer. Since Trixi.jl uses a composable design and relies heavily on Julia's type system, it is necessary to repackage this flexibility such that it can be exposed in a traditional API with static types. This is achieved by converting Trixi.jl's *elixirs*, which consist of all code to set up and run a simulation, into so-called *libelixirs*. These libelixirs are used to initialize a simulation state, which is then stored internally in LibTrixi.jl and assigned a unique integer. This integer handle is exposed in the C/Fortran API and facilitates all interaction between the main program and the actual simulation. It further allows controlling multiple independent simulations simultaneously.

[1] libcg, https://github.com/simonbyrne/libcg
[2] libdiffusion, https://github.com/omlins/libdiffusion

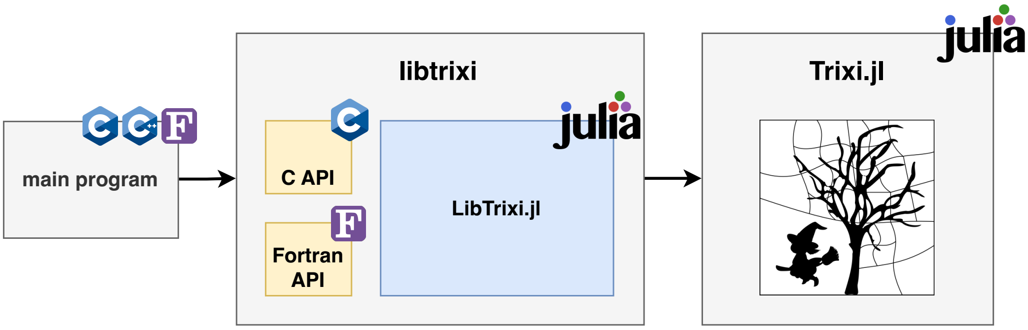

Figure 1: A main program implemented in C/C++/Fortran is able to interact with Trixi.jl via `libtrixi`.

As an alternative to the aforementioned translation layer written in C, which exposes the Julia API of LibTrixi.jl via function pointers, there exists experimental support in `libtrixi` for compiling LibTrixi.jl directly into a C library. This is achieved with PackageCompiler.jl[3], a Julia package that allows one to compile Julia packages directly into a shared C library or even a standalone executable. While other Julia packages exist that provide build options for PackageCompiler.jl, they typically use it to offer the Julia package as an executable and not as a library, e.g., Ribasim[4], Comonicon.jl[5], or SpmImage Tycoon (Riss, 2022).

In addition to the library itself, `libtrixi` comes with the tools necessary for a smooth setup and installation process. A custom shell script installs all required Julia dependencies and allows one to configure the use of system-local library dependencies, such as for the MPI library. A CMake[6]-based build system handles the build process of the C translation layer and the Fortran bindings. All parts of `libtrixi` are extensively tested using the built-in unit testing framework for Julia [7], GoogleTest[8] for the C API, and test-drive[9] for the Fortran bindings.

Acknowledgments

This project has benefited from funding by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) through the research unit FOR 5409 "SNuBIC" (project number 463312734), and through an DFG individual grant (project number 528753982).

This project has benefited from funding from the German Federal Ministry of Education and Research through the project grant "Adaptive earth system modeling with significantly reduced computation time for exascale supercomputers (ADAPTEX)" (funding id: 16ME0668K).

References

Arndt, D., Bangerth, W., Davydov, D., Heister, T., Heltai, L., Kronbichler, M., Maier, M., Pelteret, J.-P., Turcksin, B., & Wells, D. (2021). The deal.II finite element library: Design, features, and insights. *Computers & Mathematics with Applications*, *81*, 407–422. https://doi.org/10.1016/j.camwa.2020.02.022

Badia, S., & Verdugo, F. (2020). Gridap: An extensible finite element toolbox in julia. *Journal of Open Source Software*, *5*(52), 2520. https://doi.org/10.21105/joss.02520

[3]PackageCompiler.jl, https://github.com/JuliaLang/PackageCompiler.jl
[4]Ribasim, https://github.com/Deltares/Ribasim
[5]Comonicon.jl, https://github.com/comonicon/Comonicon.jl
[6]CMake, https://cmake.org
[7]Julia testing with `Test`, https://docs.julialang.org/en/v1/stdlib/Test/
[8]GoogleTest, https://google.github.io/googletest/
[9]test-drive, https://github.com/fortran-lang/test-drive

Balay, S., Gropp, W. D., McInnes, L. C., & Smith, B. F. (1997). Efficient management of parallelism in object oriented numerical software libraries. In E. Arge, A. M. Bruaset, & H. P. Langtangen (Eds.), *Modern software tools in scientific computing* (pp. 163–202). Birkhäuser Press. https://doi.org/10.1007/978-1-4612-1986-6_8

Bezanson, J., Edelman, A., Karpinski, S., & Shah, V. B. (2017). Julia: A fresh approach to numerical computing. *SIAM Review*, *59*(1), 65–98. https://doi.org/10.1137/141000671

Carlsson, K., Ekre, F., & Ferrite.jl contributors. (n.d.). *Ferrite.jl*. https://github.com/Ferrite-FEM/Ferrite.jl

Churavy, V., Godoy, W. F., Bauer, C., Ranocha, H., Schlottke-Lakemper, M., Räss, L., Blaschke, J., Giordano, M., Schnetter, E., Omlin, S., Vetter, J. S., & Edelman, A. (2022). *Bridging HPC communities through the julia programming language*. https://arxiv.org/abs/2211.02740

Harris, C. R., Millman, K. J., Walt, S. J. van der, Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N. J., Kern, R., Picus, M., Hoyer, S., Kerkwijk, M. H. van, Brett, M., Haldane, A., Fernández del Río, J., Wiebe, M., Peterson, P., … Oliphant, T. E. (2020). Array programming with NumPy. *Nature*, *585*, 357–362. https://doi.org/10.1038/s41586-020-2649-2

Riss, A. (2022). SpmImage tycoon: Organize and analyze scanning probe microscopy data. *Journal of Open Source Software*, *7*(77), 4644. https://doi.org/10.21105/joss.04644

Schlottke-Lakemper, M., Gassner, G. J., Ranocha, H., Winters, A. R., & Chan, J. (2021). *Trixi.jl: Adaptive high-order numerical simulations of hyperbolic PDEs in Julia*. https://github.com/trixi-framework/Trixi.jl. https://doi.org/10.5281/zenodo.3996439

Schlottke-Lakemper, M., Geihe, B., & Gassner, G. J. (2023). *Libtrixi: Interface library for using Trixi.jl from C/C++/Fortran*. https://github.com/trixi-framework/libtrixi. https://doi.org/10.5281/zenodo.8321803

Schlottke-Lakemper, M., Winters, A. R., Ranocha, H., & Gassner, G. J. (2021). A purely hyperbolic discontinuous Galerkin approach for self-gravitating gas dynamics. *Journal of Computational Physics*, *442*, 110467. https://doi.org/10.1016/j.jcp.2021.110467