




# Gridap: An extensible Finite Element toolbox in Julia

Santiago Badia<sup>1</sup> and Francesc Verdugo<sup>2</sup>

<sup>1</sup> School of Mathematics, Monash University, Clayton, Victoria, 3800, Australia. <sup>2</sup> Centre Internacional de Mètodes Numèrics en Enginyeria, Esteve Terrades 5, E-08860 Castelldefels, Spain.

DOI: [10.21105/joss.02520](https://doi.org/10.21105/joss.02520)

## Software

- [Review](#) 
- [Repository](#) 
- [Archive](#) 

Editor: [Kevin M. Moerman](#) 

## Reviewers:

- [@PetrKryslUCSD](#)
- [@TeroFrondeius](#)
- [@KristofferC](#)

Submitted: 10 July 2020

Published: 11 August 2020

## License

Authors of papers retain copyright and release the work under a Creative Commons Attribution 4.0 International License ([CC BY 4.0](https://creativecommons.org/licenses/by/4.0/)).

## Summary

Gridap is a new Finite Element (FE) framework, exclusively written in the Julia programming language, for the numerical simulation of a wide range of mathematical models governed by partial differential equations (PDEs). The library provides a feature-rich set of discretization techniques, including continuous and discontinuous FE methods with Lagrangian, Raviart-Thomas, or Nédélec interpolations, and supports a wide range of problem types including linear, nonlinear, single-field, and multi-field PDEs (see (Badia, Martín, & Principe, 2018, Section 3) for a detailed presentation of the mathematical abstractions behind the implementation of these FE methods). Gridap is designed to help application experts to easily simulate real-world problems, to help researchers improve productivity when developing new FE-related techniques, and also for its usage in numerical PDE courses.

The main motivation behind Gridap is to find an improved balance between computational performance, user-experience, and work-flow productivity when working with FE libraries. Previous FE frameworks, e.g., FEniCS (Alnæs et al., 2015) or Deal.II (Bangerth, Hartmann, & Kanschat, 2007) usually provides a high-level user front-end to facilitate the use of the library and a computational back-end to achieve performance. The user front-end is usually programmable in an interpreted language like Python, whereas the computational back-end is usually coded in a compiled language like C/C++ or Fortran. Users can benefit from the high-level front-end (i.e., for rapid prototyping) and simultaneously enjoy the performance of the compiled back-end. This approach reaches a compromise between performance and productivity when the back-end provides all the functionality required by the user. However, it does not satisfactorily address the needs of researchers on numerical methods willing to extend the library with new techniques or features. These extensions usually need to be done at the level of the computational back-end for performance reasons. Thus, the researcher is forced to develop a new code in a compiled language like C/C++ instead of benefiting from the productivity of scripting languages like Python, incurring serious productivity losses. In order to overcome this limitation, Gridap is fully implemented in the Julia programming language (Bezanson, Edelman, Karpinski, & Shah, 2017). Julia combines the performance of compiled languages with the productivity of interpreted ones by using type inference and just-in-time compilation to generate fast code. As a result, there is no need to use two different languages to write low-level performance code and high-level user interfaces. In addition, writing a FE library in Julia also allows one to leverage the feature-rich ecosystem of Julia libraries and exploit its excellent package manager. It permits a seamless coupling of Gridap with application-specific libraries, like optimization (Dunning, Huchette, & Lubin, 2017), an approximation of ordinary differential equations (Rackauckas & Nie, 2017), or data science (Innes, 2018), which can certainly boost the capabilities of a FE solver.

Another major feature of Gridap is that it is not a simple Julia translation of a standard object-oriented FE code. There are other FE libraries written in Julia that have been inspired by standard FE frameworks, see, e.g., JuAFEM (Carlsson, n.d.), whose interface resembles Deal.II. In contrast, Gridap adopts a novel software design that allows one to manipulate

different types of data associated with the cells of the computational mesh in a convenient way. For instance, one can build an object representing the elemental matrices for all cells in the mesh using high-level API calls, without explicitly writing any for-loop. These objects representing data for all cells of the mesh are usually *lazy*, meaning that the underlying data is never stored for all cells in the mesh simultaneously. Instead, the value for a specific cell is computed on-the-fly when needed, which certainly reduces memory requirements. This software design allows the library developers to hide assembly loops and other core computations from the user-code, leading to a very compact, user-friendly, syntax, while providing a high degree of flexibility for users to define their own weak forms. A Poisson or Stokes problem can be solved with Gridap in 10-20 lines of code, as this example for the Poisson equation shows:

```
using Gridap
# Manufactured solutions
u(x) = x[1]^2 + x[2]
f(x) = -Δ(u)(x); g(x) = u(x)
# FE mesh (aka discrete model)
pmin = Point(0,0,0); pmax = Point(1,1,1)
cells=(8,8,8); order = 1
model = CartesianDiscreteModel(pmin, pmax, cells)
# FE Spaces
V0 = TestFESpace(model=model, reffe=:Lagrangian,
    valuetype=Float64, order=order,
    conformity=:H1, dirichlet_tags="boundary")
Ug = TrialFESpace(V0, g)
# Weak form
a(u,v) = (u) (v); l(v) = v*f
trian_Ω = Triangulation(model)
quad_Ω = CellQuadrature(trian_Ω, 2*order)
t_Ω = AffineFETerm(a,l,trian_Ω,quad_Ω)
# FE Problem and solution
op = AffineFEOperator(Ug,V0,t_Ω)
uh = solve(op)
# Output for visualization
writevtk(trian_Ω,"results",
    cellfields=["uh"=>uh,"grad_uh"=>(uh)])
```

Other FE packages like FEniCS also achieve such compact user interfaces, but in contrast to Gridap, they are based on a sophisticated compiler of variational forms (Kirby & Logg, 2006), which generates, compiles and links a specialized C++ back-end for the problem at hand. One of the limitations of this approach is that the form compiler is a rigid system that is not designed to be extended by average users.

Gridap is an open-source project hosted at Github and distributed under an MIT license. The source code for Gridap has been archived to Zenodo with the linked DOI [10.5281/zenodo.3934468](https://doi.org/10.5281/zenodo.3934468).

## References

- Alnæs, M., Blechta, J., Hake, J., Johansson, A., Kehlet, B., Logg, A., Richardson, C., et al. (2015). The FEniCS Project Version 1.5. *The FEniCS Project Version 1.5*, 3(100), 9–23. doi:[10.11588/ans.2015.100.20553](https://doi.org/10.11588/ans.2015.100.20553)
- Badia, S., Martín, A. F., & Principe, J. (2018). FEMPAR: An Object-Oriented Parallel Finite Element Framework. *Archives of Computational Methods in Engineering*, 25(2), 195–271. doi:[10.1007/s11831-017-9244-1](https://doi.org/10.1007/s11831-017-9244-1)

- Bangerth, W., Hartmann, R., & Kanschat, G. (2007). Deal.II –A general-purpose object-oriented finite element library. *ACM Transactions on Mathematical Software*, 33(4). doi:[10.1145/1268776.1268779](https://doi.org/10.1145/1268776.1268779)
- Bezanson, J., Edelman, A., Karpinski, S., & Shah, V. B. (2017). Julia: A fresh approach to numerical computing. *SIAM Review*, 59(1), 65–98. doi:[10.1137/141000671](https://doi.org/10.1137/141000671)
- Carlsson, K. (n.d.). JuaFEM git repository. Retrieved from <https://github.com/KristofferC/JuAFEM.jl>
- Dunning, I., Huchette, J., & Lubin, M. (2017). JuMP: A modeling language for mathematical optimization. *SIAM Review*, 59(2), 295–320. doi:[10.1137/15M1020575](https://doi.org/10.1137/15M1020575)
- Innes, M. (2018). Flux: Elegant machine learning with Julia. *Journal of Open Source Software*, 3(25), 602. doi:[10.21105/joss.00602](https://doi.org/10.21105/joss.00602)
- Kirby, R. C., & Logg, A. (2006). A compiler for variational forms. *ACM Transactions on Mathematical Software*, 32(3), 417–444. doi:[10.1145/1163641.1163644](https://doi.org/10.1145/1163641.1163644)
- Rackauckas, C., & Nie, Q. (2017). DifferentialEquations.jl – A Performant and Feature-Rich Ecosystem for Solving Differential Equations in Julia. *Journal of Open Research Software*, 5(1). doi:[10.5334/jors.151](https://doi.org/10.5334/jors.151)