




# XCALibre.jl: an XPU unstructured finite volume Computational Fluid Dynamics library

Humberto Medina<sup>1</sup><sup>¶</sup>, Christopher D. Ellis<sup>1</sup>, Tom Mazin<sup>1</sup>, Oscar Osborn<sup>1</sup>, Timothy Ward<sup>1</sup>, Stephen Ambrose<sup>1</sup>, Svetlana Aleksandrova<sup>2</sup>, Benjamin Rothwell<sup>1</sup>, and Carol Eastwick<sup>1</sup>

<sup>1</sup> The University of Nottingham, UK <sup>2</sup> The University of Leicester, UK <sup>¶</sup> Corresponding author

DOI: [10.xxxxxx/draft](https://doi.org/10.xxxxxx/draft)

## Software

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

Editor: [Open Journals](#) 


## Reviewers:

- [@openjournals](#)

Submitted: 01 January 1970

Published: unpublished

## License

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

## Summary

Understanding the behaviour of fluid flow, such as air over a wing, oil lubrication in gas turbines, or cooling air flow in a combustor or turbine is crucial in many engineering applications, from designing aircraft and automotive components to optimising energy systems. Computational Fluid Dynamics (CFD) enables engineers to model real-world processes, optimise designs, and predict performance for a wide range of scenarios, and it has become a vital part of the modern engineering design process for creating efficient, safe, and sustainable designs. As engineers seek to develop and optimise new designs, particularly in fields where there is a drive to push the current state-of-the-art or physical limits of existing design solutions, often, new CFD methodologies or physical models are required. Therefore, extendable and flexible CFD frameworks are needed, for example, to allow seamless integration with machine learning models. In this paper, the features of the first release of the Julia package XCALibre.jl are presented. Designed with extensibility in mind, XCALibre.jl is aiming to facilitate the rapid prototyping of new fluid models and to easily integrate with Julia's powerful ecosystem, enabling access to optimisation libraries and machine learning frameworks to enhance its functionality and expand its application potential, whilst offering multi-threaded performance on CPUs and GPU acceleration.

## Statement of need

Given the importance of fluid flow simulation in engineering applications, it is not surprising that there is a wealth of CFD solvers available, both open-source and commercially available. Well established open-source codes include: OpenFOAM ([Foundation, 2024](#)), SU2 ([University, 2024](#)), CODE\_SATURN ([EDF, 2024](#)), Basilisk ([al., 2024](#)), etc. It is a testament to the open-source philosophy, and their developers, that some of these codes offer almost feature parity with commercial codes. However, established open-source and commercial codes have large codebases and, for performance reasons, have been implemented in statically compiled languages which makes it difficult to adapt and incorporate recent trends in scientific computing, for example, GPU computing and interfacing with machine learning frameworks to support the development of new models ([Ellis & Xia, 2023, 2024](#)). As a result, the research community has been actively developing new CFD codes, which is evident within the Julia ecosystem.

The Julia programming language offers a fresh approach to scientific computing, with the benefits of dynamism whilst retaining the performance of statically typed languages thanks to its just-in-time compilation approach (using LLVM compiler technology). Thus, Julia makes it easy to prototype and test new ideas whilst producing performant machine code. This simplicity-performance dualism has resulted in a remarkable growth in its ecosystem offering for scientific computing, which includes state-of-the-art packages for solving differential equations

e.g. `DifferentialEquations.jl` (Rackauckas & Nie, 2017), building machine learning models such as `Flux.jl` (Innes, 2018), `Knet.jl` (Yuret & collaborators, 2024) and `Lux.jl` (Pal, 2023), optimisation frameworks e.g. `JUMP.jl` (Lubin et al., 2023), automatic differentiation, such as `Enzyme.jl` (Moses & Churavy, 2020), etc. Likewise, excellent CFD packages have also been developed, e.g. `Oceananigans.jl` (Ramadhan et al., 2020), which provides tools for ocean modelling, `Trixi.jl` (Schlottke-Lakemper et al., 2021) which provides high-order solvers using the Discontinuous Galerkin method, and `Waterlilly.jl` (Weymouth & Font, 2024), which implements the immersed boundary method on structured grids using a staggered finite volume method. To complement and extend the CFD offering of the Julia ecosystem, `XCALibre.jl` provides a composable framework which enables the aforementioned strengths of the Julia programming language to be realised in an unstructured finite volume solver with XPU capabilities. The package is intended primarily for researchers and students, as well as engineers, who are interested in CFD applications using the built-in solvers. Additionally, `XCALibre.jl` offers developers a user-friendly and computationally efficient framework for prototyping cutting-edge CFD solvers or methodologies, which can be tested in complex geometries relevant to a wide range of engineering applications.

## Key features

The main features available in the latest release (version 0.4) are highlighted here. Users are also encouraged to explore the latest version of [the user guide](#) where the public API and current features are documented.

- **XPU computation** `XCALibre.jl` is implemented using `KernelAbstractions.jl` which allows it to support both multi-threaded CPU and GPU calculations.
- **Unstructured grids and formats** `XCALibre.jl` is implemented to support unstructured meshes using the Finite Volume method for equation discretisation. Thus, arbitrary polyhedral cells are supported, enabling the representation and simulation of complex geometries. `XCALibre.jl` provides mesh conversion functions to import externally generated grids. Currently, the Ideas (`unv`) and OpenFOAM mesh formats can be used. The `.unv` mesh format supports both 2D and 3D grids (note that the `.unv` format only supports prisms, tetrahedral, and hexahedral cells). The OpenFOAM mesh format can be used for 3D simulations (this format has no cell restrictions and supports arbitrary polyhedral cells).
- **Flow solvers** Steady and transient solvers are available, which use the SIMPLE and PISO algorithms for steady and transient simulations, respectively. These solvers support simulation of both incompressible and weakly compressible fluids (using a sensible energy model).
- **Turbulence models** RANS and LES turbulence models are supported. RANS models available in the current release include: the standard Wilcox  $k - \omega$  model (Wilcox, 1988) and the transitional  $k - \omega$  LKE model (Medina et al., 2018). For LES simulations the classic Smagorinsky model (Smagorinsky, 1963) is available.
- **VTK simulation output** Simulation results are written to `vtk` files for 2D cases and `vtu` for 3D simulations. This allows to perform simulation post-processing in ParaView, which is the leading open-source project for scientific visualisation.
- **Linear solvers and discretisation schemes** Users are able to select from a growing range of pre-defined discretisation schemes, e.g. Upwind, Linear and LUST for discretising divergence terms. By design, the choice of discretisation strategy is made on a term-by-term basis, offering great flexibility. Users must also select and configure the linear solvers used to solve the discretised equations. Linear solvers are provided by `Krylov.jl` (Montoisson & Orban, 2023) and reexported in `XCALibre.jl` for convenience (please refer to the user guide for details on exported solvers).

## 91 Examples

92 Users are referred to the documentation where examples for using XCALibre.jl are provided,  
93 including advanced examples showing how it is possible to integrate the Julia ecosystem to  
94 extend the functionality in XCALibre.jl, with examples that include flow optimisation, and  
95 integration with the Flux.jl machine learning framework.

### 96 Verification: laminar flow over a backward facing step

97 The use of XCALibre.jl is now illustrated using simple backward-facing step configuration  
98 with 4 boundaries as depicted in figure 1. The flow will be considered as incompressible and  
99 laminar. The :wall and :top boundaries will be considered as solid wall boundaries. The  
100 inflow velocity is  $1.5 \text{ m/s}$ , and the outlet boundary is set up as a pressure outlet (i.e. a  
101 Dirichlet condition with a reference value of  $0 \text{ Pa}$ ). Notice that this case uses a structured  
102 grid for simplicity, however, in XCALibre.jl the grid connectivity information is unstructured  
103 and complex geometries can be used. Here the simulation is set up to run on the CPU.  
104 The number of CPU threads to be used can be specified when launching the Julia process,  
105 e.g. `julia --threads 4`. The steps needed to carry out the simulation on GPUs can be found  
106 in the [documentation](#).

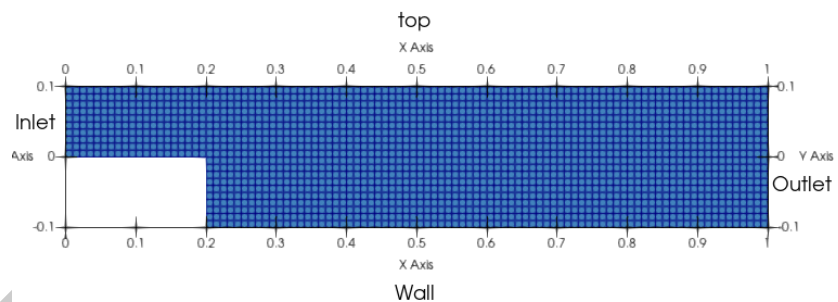


Figure 1: Computational domain

107 The corresponding simulation setup is shown below:

using XCALibre

```
# Get path to mesh file
grids_dir = pkgdir(XCALibre, "examples/0_GRIDS")
grid = "backwardFacingStep_10mm.unv"
mesh_file = joinpath(grids_dir, grid)
```

```
# Convert and load mesh
mesh = UNV2D_mesh(mesh_file, scale=0.001)
```

```
# Define flow variables & do checks
velocity = [1.5, 0.0, 0.0]; nu = 1e-3; H = 0.1
Re = velocity[1]*H/nu # check Reynolds number
```

```
# Define models
model = Physics(
    time = Steady(),
    fluid = Fluid{Incompressible}(nu = nu),
    turbulence = RANS{Laminar}(),
    energy = Energy{Isothermal}(),
```

```

domain = mesh
)

# Assign boundary conditions
@assign! model.momentum.U (
    Dirichlet(:inlet, velocity),
    Neumann(:outlet, 0.0),
    Wall(:wall, [0.0, 0.0, 0.0]),
    Wall(:top, [0.0, 0.0, 0.0]),
)

@assign! model.momentum.p (
    Neumann(:inlet, 0.0),
    Dirichlet(:outlet, 0.0),
    Neumann(:wall, 0.0),
    Neumann(:top, 0.0)
)

# Specify discretisation schemes
schemes = (
    U = set_schemes(divergence = Linear),
    p = set_schemes() # no input provided (will use defaults)
)

# Configuration: linear solvers
solvers = (
    U = set_solver(
        model.momentum.U;
        solver      = BicgstabSolver,
        preconditioner = Jacobi(),
        convergence = 1e-7,
        relax       = 0.7,
        rtol        = 1e-4,
        atol        = 1e-10
    ),
    p = set_solver(
        model.momentum.p;
        solver      = CgSolver,
        preconditioner = Jacobi(),
        convergence = 1e-7,
        relax       = 0.7,
        rtol        = 1e-4,
        atol        = 1e-10
    )
)

# Configuration: runtime and hardware information
runtime = set_runtime(iterations=2000, time_step=1, write_interval=2000)
hardware = set_hardware(backend=CPU(), workgroup=1024)

# Configuration: build Configuration object
config = Configuration(
    solvers=solvers, schemes=schemes, runtime=runtime, hardware=hardware)

# Initialise fields

```

```
initialise!(model.momentum.U, velocity)
initialise!(model.momentum.p, 0.0)
```

```
# Run simulation
residuals = run!(model, config);
```

108 The velocity and pressure results are verified with those obtained using OpenFOAM (Foundation,  
109 2024) and shown in figure 2, showing that they are in excellent agreement.

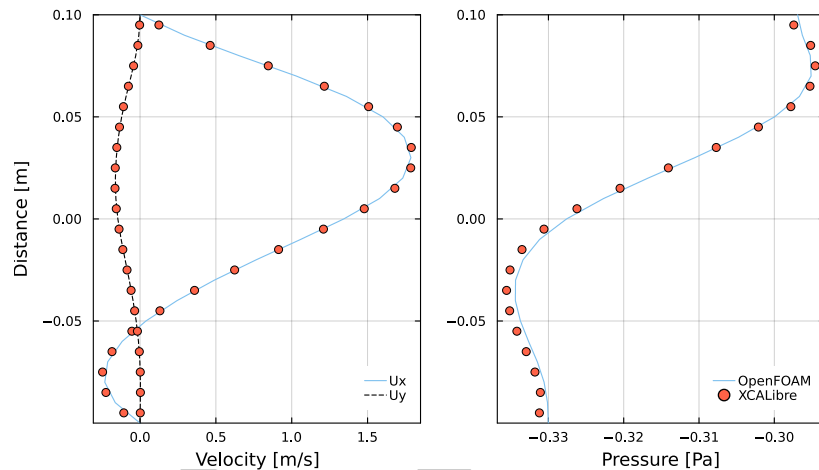


Figure 2: Comparision with OpenFOAM

## References

- 110 al., S. P. et. (2024). Basilisk. *Basilisk*. <http://basilisk.fr/>
- 112 EDF. (2024). CODE\_SATURN. *GitHub Repository*. [https://github.com/code-saturne/code\\_](https://github.com/code-saturne/code_saturne)  
113 [saturne](https://github.com/code-saturne/code_saturne)
- 114 Ellis, C., & Xia, H. (2023). Data-driven turbulence anisotropy in film and effusion cooling  
115 flows. *Physics of Fluids*, 35. <https://doi.org/10.1063/5.0166685>
- 116 Ellis, C., & Xia, H. (2024). LES informed data-driven models for RANS simulations of  
117 single-hole cooling flows. *International Journal of Heat and Mass Transfer*, 235. <https://doi.org/10.1016/j.ijheatmasstransfer.2024.126150>
- 118
- 119 Foundation, O. (2024). Open field operation and manipulation. *GitHub Repository*. <https://github.com/OpenFOAM>
- 120
- 121 Innes, M. (2018). Flux: Elegant machine learning with julia. *Journal of Open Source Software*.  
122 <https://doi.org/10.21105/joss.00602>
- 123 Lubin, M., Dowson, O., Dias Garcia, J., Huchette, J., Legat, B., & Vielma, J. P. (2023).  
124 JuMP 1.0: Recent improvements to a modeling language for mathematical optimiza-  
125 tion. *Mathematical Programming Computation*, 15, 581–589. [https://doi.org/10.1007/](https://doi.org/10.1007/s12532-023-00239-3)  
126 [s12532-023-00239-3](https://doi.org/10.1007/s12532-023-00239-3)
- 127 Medina, H., Beechook, A., Fadhila, H., Aleksandrova, S., & Benjamin, S. (2018). A novel  
128 laminar kinetic energy model for the prediction of pretransitional velocity fluctuations and  
129 boundary layer transition. *International Journal of Heat and Fluid Flow*, 69, 150–163.  
130 [https://doi.org/https://doi.org/10.1016/j.ijheatfluidflow.2017.12.008](https://doi.org/10.1016/j.ijheatfluidflow.2017.12.008)
- 131 Montoisson, A., & Orban, D. (2023). Krylov.jl: A Julia basket of hand-picked Krylov methods.  
132 *Journal of Open Source Software*, 8(89), 5187. <https://doi.org/10.21105/joss.05187>

- 133 Moses, W., & Churavy, V. (2020). Instead of rewriting foreign code for machine learning,  
134 automatically synthesize fast gradients. In H. Larochelle, M. Ranzato, R. Hadsell, M. F.  
135 Balcan, & H. Lin (Eds.), *Advances in neural information processing systems* (Vol. 33, pp.  
136 12472–12485). Curran Associates, Inc. [https://proceedings.neurips.cc/paper/2020/file/  
137 9332c513ef44b682e9347822c2e457ac-Paper.pdf](https://proceedings.neurips.cc/paper/2020/file/9332c513ef44b682e9347822c2e457ac-Paper.pdf)
- 138 Pal, A. (2023). *Lux: Explicit parameterization of deep neural networks in julia* (Version v0.5.0).  
139 Zenodo. <https://doi.org/10.5281/zenodo.7808904>
- 140 Rackauckas, C., & Nie, Q. (2017). Differentialequations.jl—a performant and feature-rich  
141 ecosystem for solving differential equations in julia. *Journal of Open Research Software*,  
142 5(1), 15.
- 143 Ramadhan, A., Wagner, G. L., Hill, C., Campin, J.-M., Churavy, V., Besard, T., Souza,  
144 A., Edelman, A., Ferrari, R., & Marshall, J. (2020). Oceananigans.jl: Fast and friendly  
145 geophysical fluid dynamics on GPUs. *Journal of Open Source Software*, 5(53), 2018.  
146 <https://doi.org/10.21105/joss.02018>
- 147 Schlottke-Lakemper, M., Winters, A. R., Ranocha, H., & Gassner, G. J. (2021). A purely  
148 hyperbolic discontinuous Galerkin approach for self-gravitating gas dynamics. *Journal of  
149 Computational Physics*, 442, 110467. <https://doi.org/10.1016/j.jcp.2021.110467>
- 150 Smagorinsky, J. (1963). GENERAL CIRCULATION EXPERIMENTS WITH THE PRIMITIVE  
151 EQUATIONS: I. THE BASIC EXPERIMENT. *Monthly Weather Review*, 91(3), 99–164.  
152 [https://doi.org/10.1175/1520-0493\(1963\)091%3C0099:GCEWTP%3E2.3.CO;2](https://doi.org/10.1175/1520-0493(1963)091%3C0099:GCEWTP%3E2.3.CO;2)
- 153 University, S. (2024). SU2. *GitHub Repository*. <https://github.com/su2code/SU2>
- 154 Weymouth, G. D., & Font, B. (2024). *WaterLily.jl: A differentiable and backend-agnostic julia  
155 solver to simulate incompressible viscous flow and dynamic bodies*. [https://arxiv.org/abs/  
156 2407.16032](https://arxiv.org/abs/2407.16032)
- 157 Wilcox, D. C. (1988). Reassessment of the scale-determining equation for advanced turbulence  
158 models. *AIAA Journal*, 26(11), 1299–1310. <https://doi.org/10.2514/3.10041>
- 159 Yuret, D., & collaborators. (2024). Knet. *GitHub Repository*. [https://github.com/  
160 denizyuret/Knet.jl](https://github.com/denizyuret/Knet.jl)