The Journal of Open Source Software

XCALibre.jl: A general-purpose unstructured finite volume Computational Fluid Dynamics library

Humberto Medina[1¶], **Chris Ellis**[1], **Tom Mazin**[1], **Oscar Osborne**[1], **Timothy Ward**[1], **Stephen Ambrose**[1], **Svetlana Aleksandrova**[2], **Benjamin Rothwell**[1], and **Carol Eastwick**[1]

1 The University of Nottingham, UK **2** The University of Leicester, UK ¶ Corresponding author

DOI: 10.xxxxxx/draft

Software

- Review
- Repository
- Archive

Editor: Open Journals

Reviewers:

- @openjournals

Submitted: 01 January 1970
Published: unpublished

Summary

Understanding the behaviour of fluid flow, such as air over a wing, water in a pipeline, or fuel in an engine is crucial in many engineering applications, from designing aircraft and automotive components to optimising energy systems, etc. Computational Fluid Dynamics (CFD) enables engineers to model real-world conditions, optimise designs, and predict performance under 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 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, SU2, CODE_SATURN, Gerris, 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, the more feature-rich open-source 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, which is also the case for commercial codes (to a larger extent due to their closed source nature where interfaces to code internals can be quite rigid – although thanks to access to more resources commercial codes have been steadily ported to work on GPUs). 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 machine code that is performant. 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 (`DifferentialEquations.jl`), building machine learning models (`Flux.jl`, `Knet.jl` and `Lux.jl`), optimisation frameworks (`JUMP.jl`, XXX and XXX, and more), automatic differentiation (), etc. Likewise, excellent CFD packages have also been developed, most notoriously: `Oceananigans.jl`, which provides tools for ocean modelling, `Trixi.jl` which provides high-order for solvers using the Discontinuous Garlekin method, and `Waterlilly.jl` which implements the immerse boundary method on structured grids using a staggered finite volume method. In this context, `XCALibre.jl` aims to complement and extend the Julia ecosystem by providing a cell-centred and unstructured finite volume general-purpose CFD framework for the simulation of both incompressible and weakly compressible flows. The package is intended primarily for researchers and students, as well as engineers, who are interested in CFD applications using the built-in solvers or those who seek a user-friendly framework for developing new CFD solvers or methodologies.

Key features

A brief summary of the main features available in the first public release (version `0.3.x`) are highlighted. 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 load externally generated grid. 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 `OpenOAM` mesh format can be used for 3D simulations (the mesh `OpenFOAM` mesh format has no cell restrictions and support 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 (add ref) and the transitional $k - \omega LKE$ model (add ref). For LES simulations the classic `Smagorinsky` model 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** in `XCALibre.jl` 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` (REF) and reexported in `XCALibre.jl` for convenience (please refer to the user guide for details on exported solvers).

Example: laminar flow over a backward facing step

quick description of flow and domain

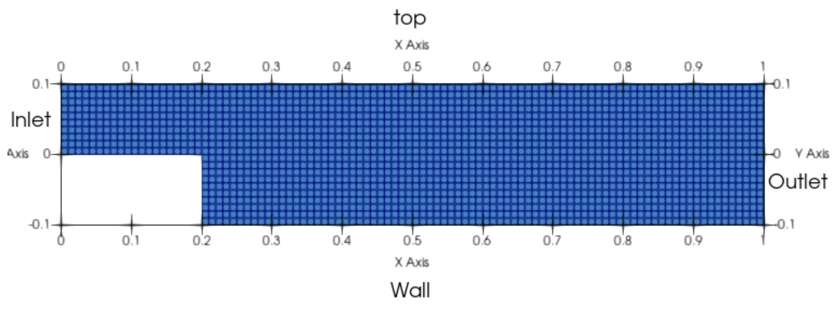

Figure 1: My figure

```julia
using XCALibre

grids_dir = pkgdir(XCALibre, "examples/0_GRIDS")
grid = "backwardFacingStep_10mm.unv"
mesh_file = joinpath(grids_dir, grid)

mesh = UNV2D_mesh(mesh_file, scale=0.001)

hardware = set_hardware(backend=CPU(), workgroup=4)

velocity = [1.5, 0.0, 0.0]
nu = 1e-3
Re = velocity[1]*0.1/nu

model = Physics(
    time = Steady(),
    fluid = Fluid{Incompressible}(nu = nu),
    turbulence = RANS{Laminar}(),
    energy = Energy{Isothermal}(),
    domain = mesh
    )

@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)
)

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

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
    )
)

runtime = set_runtime(iterations=2000, time_step=1, write_interval=2000)

config = Configuration(
solvers=solvers, schemes=schemes, runtime=runtime, hardware=hardware)

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

residuals = run!(model, config);
```

Quick narrative of results

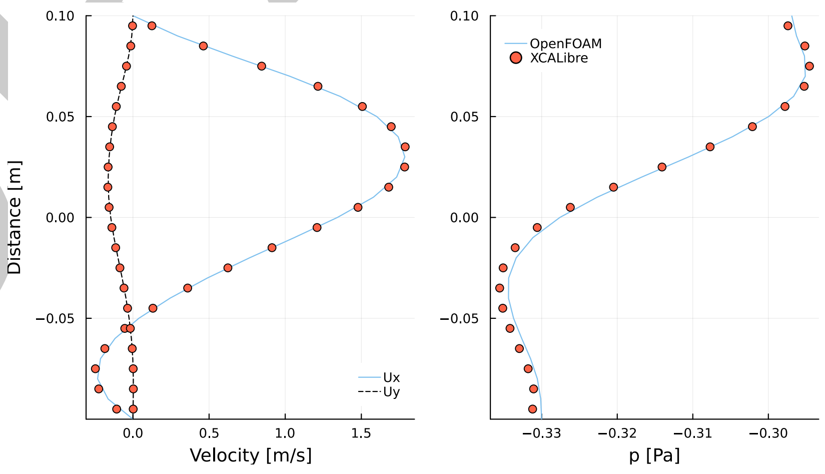

Figure 2: My figure

Other examples

Quick text to point to advanced examples in the documentation

Acknowledgements

We acknowledge contributions from Brigitta Sipocz, Syrtis Major, and Semyeong Oh, and support from Kathryn Johnston during the genesis of this project.

References