The Journal of Open Source Software

Kinetic.jl: A portable finite volume toolbox for scientific and neural computing

Tianbai Xiao[1]

1 Karlsruhe Institute of Technology, 76131 Karlsruhe, Germany

DOI: 10.21105/joss.03060

Software

- Review
- Repository
- Archive

Editor: Patrick Diehl

Reviewers:

- @rdeits
- @jarvist

Submitted: 06 January 2021
Published: 15 June 2021

Summary

Kinetic.jl is a lightweight finite volume toolbox written in the Julia programming language for the study of computational physics and scientific machine learning. It is an open-source project hosted on GitHub and distributed under the MIT license. The main module consists of KitBase.jl for basic physics and KitML.jl for neural dynamics. The function library provides a rich set of numerical fluxes and source terms for differential and integral equations. Any advection-diffusion type mechanical or neural equation can be set up and solved within the framework. Machine learning methods can be seamlessly integrated to build data-driven closure models and accelerate the calculation of nonlinear terms. The package is designed to balance programming flexibility for scientific research, algorithmic efficiency for applications, the simplicity for educational usage.

Statement of need

A physical system can perform a wonderfully diverse set of acts on different characteristic scales. It is challenging to propose a universal theory that can be applied to describing multi-scale physical evolutions quantitatively. For example, particle transport can be depicted statistically by fluid mechanics at a macroscopic level (Batchelor, 2000), but needs to be followed in more detail by the kinetic theory of gases at the molecular mean free path scale (Chapman et al., 1990). With rapidly advancing computing power, the finite volume method (FVM) provides a prevalent method to conduct direct numerical simulations based on first physical principles.

Most existing FVM libraries, e.g., OpenFOAM (Jasak et al., 2007), are dedicated to solving the Euler and the Navier-Stokes equations. Very limited work has been done for phase-field models (Krause et al., 2021; Zhu et al., 2017). Since classical fluid dynamics basically requires an one-shot simulation process from initial to final solution fields, these libraries are mostly written in compiled languages (C/C++ and Fortran). Such approaches enjoy good execution efficiency but sacrifice the flexibility of secondary development. This makes it cumbersome to integrate existing numerical solvers with scientific machine learning (SciML) packages, as interactive programming is becoming a mainstream practice in data science. This also causes considerable difficulties to general or educational users who are not familiar with the package in configuring environments and compiling binaries.

One compromise can be made by using a combination of static and dynamic languages (Clawpack Development Team, 2020), where the high-level front-ends and the low-level computational back-ends are split. This methodology benefits general users, while researchers still need to work on the back-end if a new feature is required. The so-called two-language problem introduces additional tradeoffs in both development and execution. For example, a two-tiered system brings unavoidable challenges for type domain transition and memory management.

Special attention needs to be paid on optimizing the high-level codes, e.g., the vectorization of massive computation part, which can be unnatural in a physical simulation and might generate additional temporary objects. In addition, interfacing between layers may add significant overhead and makes whole-program optimization much more difficult (Bezanson et al., 2012). Unlike these packages, Kinetic.jl is built upon the Julia programming language (Bezanson et al., 2017), which is dynamically typed and designed for high performance computing for a broad range of devices. Based on type inference and multiple dispatch, it is a promising choice to solve the two-language problem.

Kinetic.jl focuses on the theoretical and numerical studies of many-particle systems of gases, photons, plasmas, neutrons, etc. (Xiao et al., 2017, 2020) A hierarchy of abstractions is implemented in the library. At the highest level, it is feasible to model and simulate a fluid dynamic problem within ten lines of code. At the lowest level, we designed methods for general numbers and arrays so that it is possible to cooperate with existing packages in Julia ecosystem. For example, Flux.jl (Innes et al., 2018) can be used to create and train scientific machine learning models. Innovations of the package are:

- 100% Julia stack that encounters no two-language problem
- Comprehensive support for kinetic theory and phase-space equations
- Lightweight design to ensure the flexibility for secondary development
- Close coupling with scientific machine learning

KitBase.jl

The main module of Kinetic.jl is split into two pieces to reduce the just-in-time (JIT) compilation time for domain specific applications. The basic physical laws and finite volume method are implemented in KitBase.jl. It provides a variety of solvers for the Boltzmann equation, Maxwell's equations, advection-diffusion equation, Burgers' equation, Euler and Navier-Stokes equations, etc. Different parallel computing techniques are provided, e.g., multi-threading, distributed computing, and CUDA programming.

In the following, we present an illustrative example of solving a lid-driven cavity problem with the Boltzmann equation. Two initialization methods, i.e., configuration text and Julia script, are available for setting up the solver. With the configuration file `config.toml` set as below,

```
# setup
matter = gas # material
case = cavity # case
space = 2d2f2v # phase
flux = kfvs # flux function
collision = bgk # intermolecular collision
nSpecies = 1 # number of species
interpOrder = 2 # interpolation order of accuracy
limiter = vanleer # limiter function
boundary = maxwell # boundary condition
cfl = 0.8 # CFL number
maxTime = 5.0 # maximal simulation time

# physical space
x0 = 0.0 # starting point in x
x1 = 1.0 # ending point in x
nx = 45 # number of cells in x
```

```
y0 = 0.0 # starting point in y
y1 = 1.0 # ending point in y
ny = 45 # number of cells in y
pMeshType = uniform # mesh type
nxg = 0 # number of ghost cell in x
nyg = 0 # number of ghost cell in y

# velocity space
umin = -5.0 # starting point in u
umax = 5.0 # ending point in u
nu = 28 # number of cells in u
vmin = -5.0 # starting point in v
vmax = 5.0 # ending point in v
nv = 28 # number of cells in v
vMeshType = rectangle # mesh type
nug = 0 # number of ghost cell in u
nvg = 0 # number of ghost cell in v

# gas property
knudsen = 0.075 # Knudsen number
mach = 0.0 # Mach number
prandtl = 1.0 # Prandtl number
inK = 1.0 # molecular inner degree of freedom
omega = 0.72 # viscosity index of hard-sphere gas
alphaRef = 1.0 # viscosity index of hard-sphere gas in reference state
omegaRef = 0.5 # viscosity index of hard-sphere gas ub reference state

# boundary condition
uLid = 0.15 # U-velocity of moving wall
vLid = 0.0 # V-velocity of moving wall
tLid = 1.0 # temperature of wall
```

we can execute the following codes

```
using Kinetic
set, ctr, xface, yface, t = initialize("config.toml")
t = solve!(set, ctr, xface, yface, t)
plot_contour(set, ctr)
```

In the above codes, the computational setup is stored in `set`. The solutions over control volumes are represented in an array `ctr`, while `xface` and `yface` record the interface fluxes along x and y directions. In this example, the structured mesh is generated automatically by Kinetic.jl, while a non-structured mesh file can also be imported and used for computation. The result is visualized with built-in function `plot_contour`, which presents the distributions of gas density, velocity, and temperature inside the cavity.

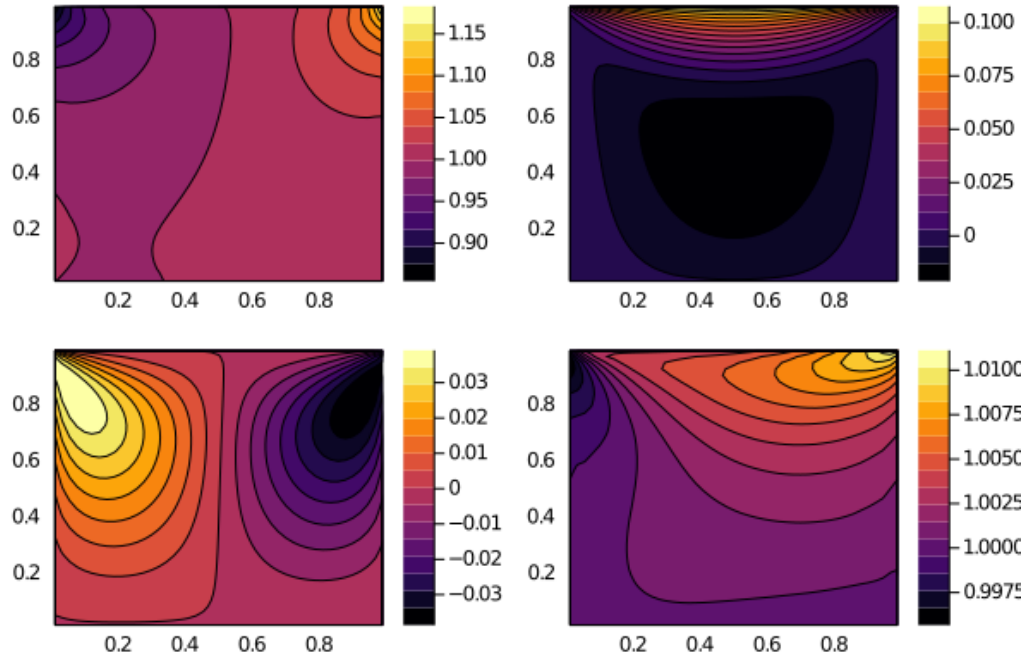

Fig. 1: macroscopic variables in the lid-driven cavity (top left: density, top right: U-velocity, bottom left: V-velocity, bottom right: temperature).

KitML.jl

Machine learning has increasing momentum in scientific computing. Given the nonlinear structure of differential and integral equations, it is promising to incorporate the universal function approximators from machine learning surrogate models into the governing equations and achieve a better balance between efficiency and accuracy. In KitML.jl, we implement strategies to construct hybrid mechanical-neural differential operators and form structure-preserving data-driven closure models. The detailed background can be found in Xiao & Frank (2020).

Extension

Numerical simulations of nonlinear models and differential equations are essentially connected with supercomputers and high-performance computing (HPC). Considering that some existing hardware architecture, e.g., Sunway TaihuLight with Chinese-designed SW26010 processors, only provides optimization for specific languages, we have developed an accompanying package KitFort.jl. This is not a default component of Kinetic.jl but can be manually imported. In addition, a wrapper, kineticpy, has been built to locate structures and methods from the Python ecosystem.

Acknowledgements

The current work is funded by the Alexander von Humboldt Foundation (Ref3.5-CHN-1210132-HFST-P).

References

Batchelor, G. K. (2000). *An introduction to fluid dynamics*. Cambridge University Press. https://doi.org/10.1017/CBO9780511800955

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

Bezanson, J., Karpinski, S., Shah, V. B., & Edelman, A. (2012). Julia: A fast dynamic language for technical computing. *arXiv Preprint arXiv:1209.5145*.

Chapman, S., Cowling, T. G., & Burnett, D. (1990). *The mathematical theory of non-uniform gases: An account of the kinetic theory of viscosity, thermal conduction and diffusion in gases*. Cambridge University Press. https://doi.org/10.2307/3609795

Clawpack Development Team. (2020). *Clawpack software*. https://doi.org/10.5281/zenodo.4025432

Innes, M., Saba, E., Fischer, K., Gandhi, D., Rudilosso, M. C., Joy, N. M., Karmali, T., Pal, A., & Shah, V. (2018). Fashionable modelling with Flux. *CoRR*, *abs/1811.01457*. https://arxiv.org/abs/1811.01457

Jasak, H., Jemcov, A., Tukovic, Z., & others. (2007). OpenFOAM: A C++ library for complex physics simulations. *International Workshop on Coupled Methods in Numerical Dynamics*, *1000*, 1–20.

Krause, M. J., Kummerländer, A., Avis, S. J., Kusumaatmaja, H., Dapelo, D., Klemens, F., Gaedtke, M., Hafen, N., Mink, A., Trunk, R., & others. (2021). OpenLB—open source lattice Boltzmann code. *Computers & Mathematics with Applications*, *81*, 258–288. https://doi.org/10.1016/j.camwa.2020.04.033

Xiao, T., Cai, Q., & Xu, K. (2017). A well-balanced unified gas-kinetic scheme for multiscale flow transport under gravitational field. *Journal of Computational Physics*, *332*, 475–491. https://doi.org/10.1016/j.jcp.2016.12.022

Xiao, T., & Frank, M. (2020). *Using neural networks to accelerate the solution of the Boltzmann equation*. http://arxiv.org/abs/2010.13649

Xiao, T., Liu, C., Xu, K., & Cai, Q. (2020). A velocity-space adaptive unified gas kinetic scheme for continuum and rarefied flows. *Journal of Computational Physics*, *415*, 109535. https://doi.org/10.1016/j.jcp.2020.109535

Zhu, L., Chen, S., & Guo, Z. (2017). dugksFoam: An open source OpenFOAM solver for the Boltzmann model equation. *Computer Physics Communications*, *213*, 155–164. https://doi.org/10.1016/j.cpc.2016.11.010