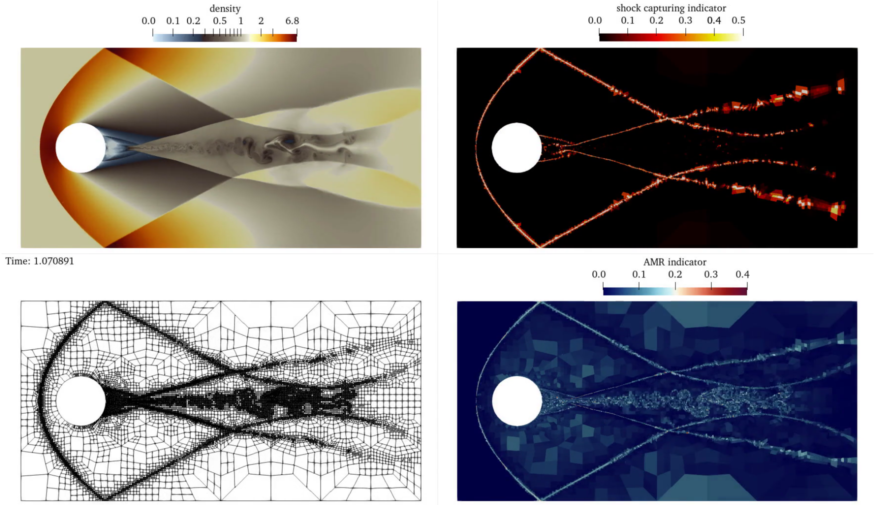
Why Julia?

Hendrik Ranocha

2023-10-09

Scientific computing: simulation of a Mach 3 flow with Trixi.jl[1]



Credit: Andrew R. Winters et al. with Trixi.jl

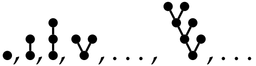
[1] R., Schlottke-Lakemper, Winters, Faulhaber, Chan and Gassner (2022); Schlottke-Lakemper, Winters, R. and Gassner (2021)

Numerical analysis: studying time integrators with BSeries.jl[2]

- Analysis of numerical integrators for $u'(t) = f\big(u(t)\big)$

$$u^{n+1} = B(a, \Delta t f, u^n) = a(\varnothing)u^n + \sum_{\tau \in T} \frac{h^{|\tau|}}{\sigma(\tau)} a(\tau)F(\tau)(u^n)$$

- Based on rooted trees and elementary differentials



$$f(u), f'\big(f(u)\big), f'\Big(f'\big(f(u)\big)\Big), f''\big(f(u), f(u)\big), \dots$$

	Mod. eq.	Mod. int.	Energy pres.
`pybs`	≈ 8.3 s	—	≈ 8.4 s
BSeries.jl	≈ 0.1 s	≈ 0.05 s	≈ 0.1 s

[2] Ketcheson and R. (2023)

Table of contents

Julia

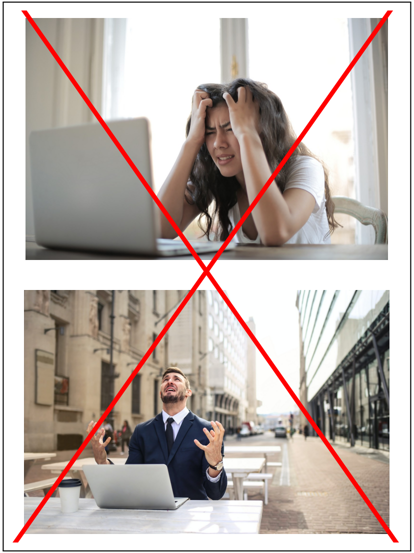
According to the official website `https://julialang.org`, Julia is ...

- free
- dynamic
- general
- composable
- fast
- reproducible

Julia encourages good software development practices!

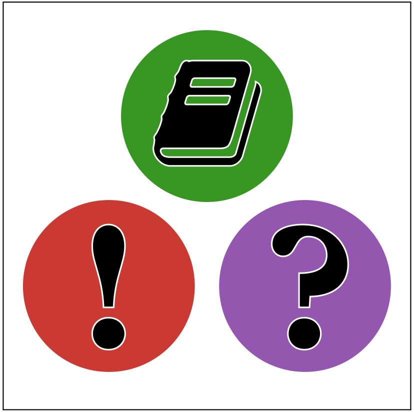
Testing

- Testing framework Test.jl in the standard library
- Continuous integration (CI) via GitHub actions
- Coverage reports via Coveralls.io and Codecov.io



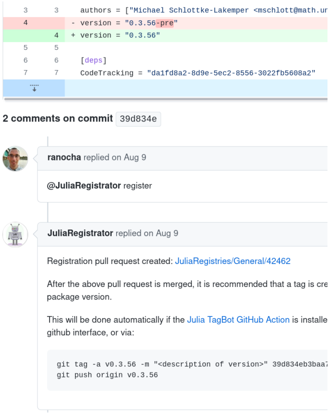
Documentation

- Docstrings
- Doctests and rendering with Documenter.jl
- GitHub actions and pages



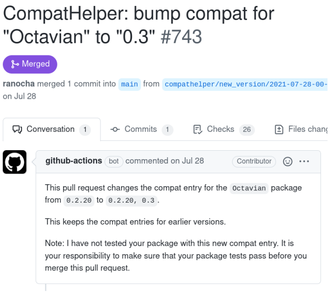
Streamlined release management & dependency tracking via GitHub

`JuliaRegistrator.jl`
Register new versions



`TagBot`
Tag registered versions

`CompatHelper.jl`
Keep dependencies up-to-date



Trixi.jl developers

Andrés Rueda-Ramírez‡, Cologne
methods for plasma simulation

Andrew Winters*, Linköping
shallow water equations
high-order curved meshes

Benedict Geihe‡, Cologne
GPU acceleration for HPC

Benjamin Bolm, Cologne
shock capturing methods

Daniel Bach‡, Cologne
coupled plasma simulations

Daniel Döhring‡, Aachen
time integration methods

David Knapp‡, Cologne
adaptive hybrid meshes
application: earth system modeling

Erik Faulhaber, Cologne
particle-based simulation methods

Gregor Gassner*, Cologne
shock-turbulence interaction

Hendrik Ranocha*, Hamburg
adaptive time integration schemes
structure-preserving methods

Jesse Chan*, Houston
entropy stable methods
simplex meshes

Johannes Markert‡, Cologne
adaptive hybrid meshes
application: tank sloshing

Lars Christmann‡, Aachen
high-performance computing
scientific machine learning

Michael Schlottke-Lakemper*, Aachen
adaptive multi-physics simulations
high-performance computing

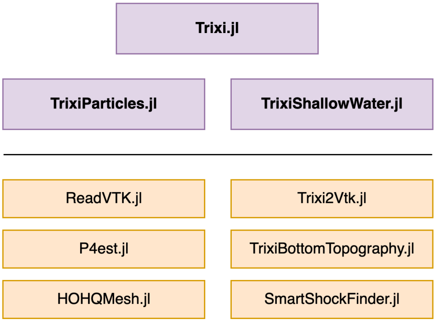
Niklas Neher‡, Stuttgart
particle-based simulation methods

Patrick Ersing, Linköping
shallow water equations
well-balanced schemes

Simon Candelaresi‡, Stuttgart
parallel multi-physics coupling
plasma physics

Sophia Schmickler, Cologne
scientific machine learning

5 principal developers*, 9 third-party funded scientists‡, 4 university-funded scientist



Bachelor/master theses
completed: 11
ongoing: 3

PhD theses
completed: 1
ongoing: 7

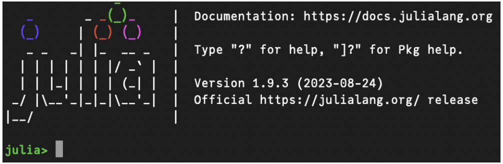
Interactivity and general programming

Interactivity

- Julia REPL (read-eval-print-loop)
- Visual studio code extension
- Jupyter
- Pluto.jl notebooks
- GUIs

General programming

- Rich type system
- N-dimensional arrays
- …



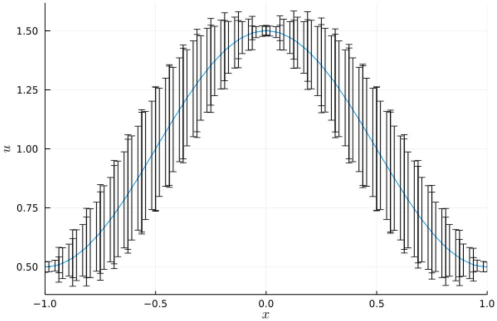
$\longrightarrow$ Pluto.jldemo

Example for composability: numerical simulations with uncertainty

Combine

- Trixi.jl: spatial semidiscretization
- OrdinaryDiffEq.jl: time integration
- Measurements.jl: values with uncertainty

  `velocity = 1.0 ± 0.1`
- Plots.jl: visualization

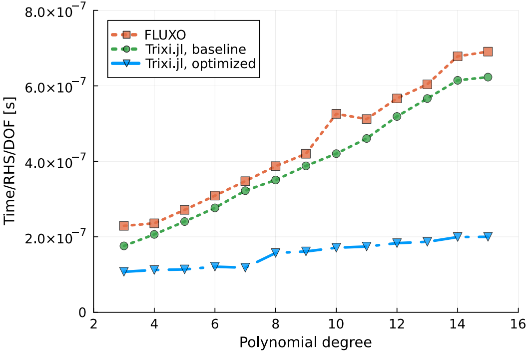


Credit: R. et al., documentation of Trixi.jl

Serial performance on par with Fortran

- 3D compressible Euler simulation (inviscid Taylor-Green vortex)
- Curved mesh, entropy-conservative fluxes
- Comparable performance as Fortran code FLUXO (same algorithms)

$\longrightarrow$ Performance depends on optimization effort



(lower is better)

arXiv: `2112.10517` | repro: `tinyurl.com/ecperf`

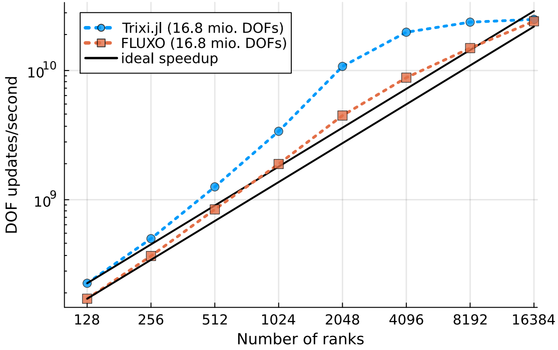
Parallel scalability experiments on JURECA

JURECA

- CPU cluster at Jülich Supercomputing Centre, Forschungszentrum Jülich
- 480 compute nodes
  - 2×AMD EPYC 7742, 2×64 cores, 2.25 GHz
  - 512 GB DDR4, 3200 MHz
  - 128 cores/node, 4 GB/core
- diskless nodes

Parallel scalability of Trixi.jl

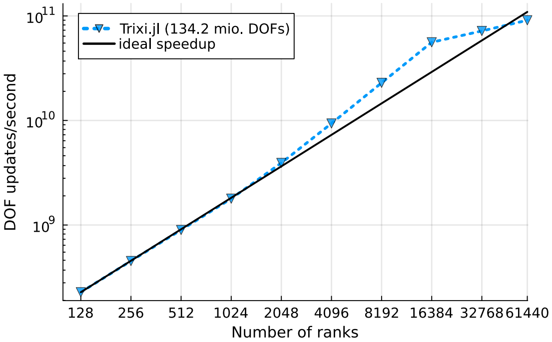
- Strong scaling experiment on JURECA (MPI only) with Julia v1.8.5
- 16.8 mio. degrees of freedom
- Good scalability to 16,384 MPI ranks
- Comparable performance as Fortran code FLUXO



(higher is better)

Parallel scalability to >50,000 MPI ranks

- Same setup as before, but 134.2 mio. DOFs
- Good scalability from 128 to 61,440 cores
- At full JURECA: 400× speedup, ∼34 elements/rank



(higher is better)

Reproducibility

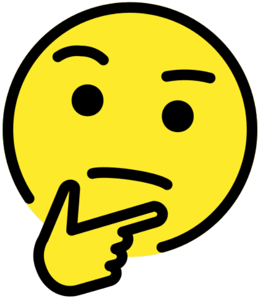
- Common interfaces allow splitting up tasks and enable code reuse
  - Depending on others can be scary
  - Dependency management and reproducibility infrastructure mitigate this
- Reproducibility is key in modern scientific computing
  - Dependency management is built into Julia
  - Binary dependencies are handled as well

“More than $70\,\%$ of researchers have tried and failed to reproduce another scientist’s experiments, and more than half have failed to reproduce their own experiments.”

— Baker, Nature 533, 2016

Why should I care about reproducibility in scientific computing?

- Scientific motivation: best practice/“expected”
- Legal motivation: my funding agency says so
- Moral motivation: public money, public X
- Personal motivation:
  Allow others to build upon my results
  (and cite me/collaborate with me)



Re: Why should I care about reproducibility?

"One of the strengths of this contribution is the accessibility it provides to the algorithms. Computational fluid dynamics packages often involve many underlying dependencies that can take several hours to download, configure, and compile . . . By using Julia . . . , the authors have significantly reduced this burden: I was able to (begin) reproducing their results within minutes."

— Anonymous Reviewer, ACM TOMS[3], 2022

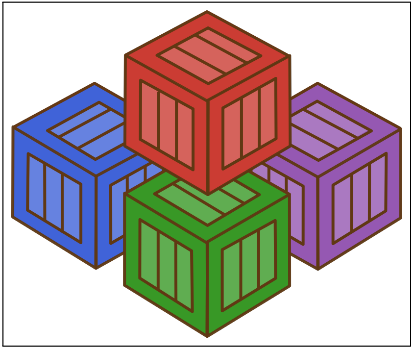
[3] R., Schlottke-Lakemper, Chan, Rueda-Ramírez, Winters, Hindenlang and Gassner (2021)

Packages

- Julia packages are Git repositories
- Package manager `Pkg` in the standard library

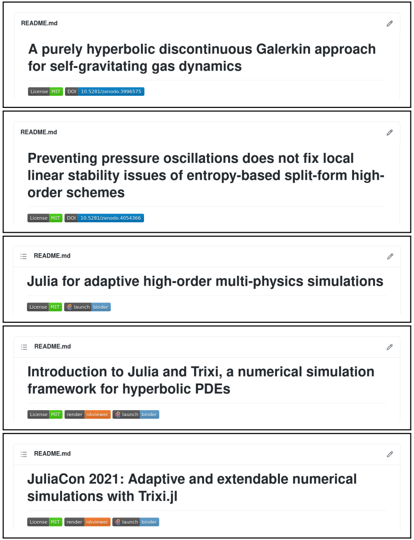
```
julia> using Pkg; Pkg.add("Trixi")
```

- General registry
- Semantic versioning



Projects

- Specify direct dependencies with versions
- Version control friendly version control (`Project.toml` and `Manifest.toml`)
- Excellent for reproducible science:
  - Paper #1: `https://git.io/JYBtP`
  - Paper #2: `https://git.io/JYBtA`
  - Paper #3: `https://git.io/JuEIO`
  - Talk #1: `https://git.io/JqnrE`
  - Talk #2: `https://git.io/JcLMy`
  - Talk #3: `https://git.io/JcL6G`
  - ...

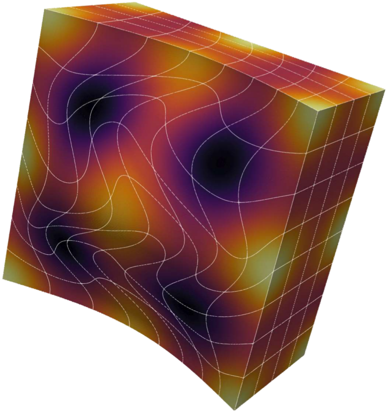


Binary dependencies

- Pre-compiled binaries bundled as packages ("JLL" packages)
- Install via regular package manager
- Natively call C/Fortran code from Julia

Example: adaptive meshes with p4est

- Wrapper package P4est.jl
- Auto-installs binaries with MPI support
- Works on Linux, macOS, Windows



Calling binaries is fast in Julia[4]

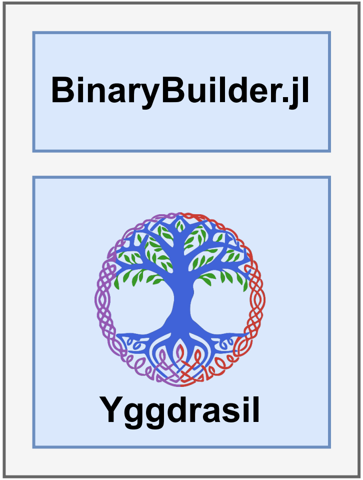
Function Signature	Pybind11		Julia's `ccall`		Speedup
`int fn0()`	132	± 14.9	2.34	± 1.24	$56\times$
`int fn1(int)`	217	± 20.9	2.35	± 1.33	$92\times$
`double fn2(int, double)`	232	± 11.7	2.32	± 0.189	$100\times$
`char* fn3(int, double, char*)`	267	± 28.9	6.27	± 0.396	$42\times$

Table: Round-trip times for calling C functions from Python and Julia in nanoseconds. The benchmark results were collected by using an Intel Core i7-1185G7 CPU running at 3.00 GHz with Julia version 1.7.1, Python version 3.8.10, and Pybind11 version 2.9.1.

[4] Churavy et al. (2022)

BinaryBuilder.jl: create Julia packages with binary artifacts

- BinaryBuilder.jl: automate building binaries for different targets
- Cross-compile locally for all Julia-supported hosts
  - Linux, macOS, Windows, FreeBSD
  - x86_64, i686, ARM
- Output: “JLL” package with binary artifacts
- Yggdrasil: central Julia repo for BB.jl recipes
  → automatically create and register JLLs



Workflow: binary dependency management in Julia

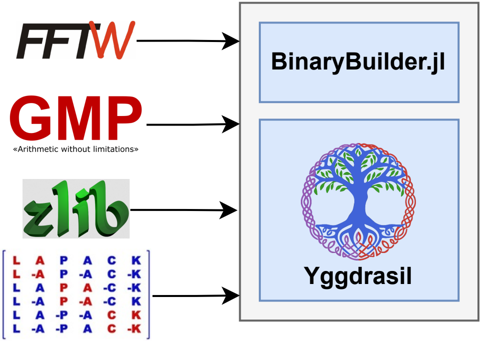
FFTW

GMP
«Arithmetic without limitations»

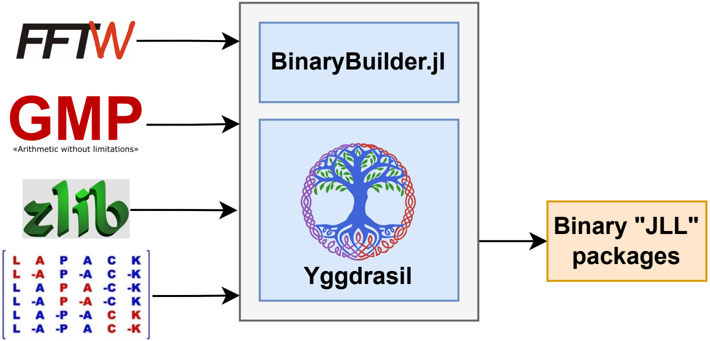
zlib

LAPACK

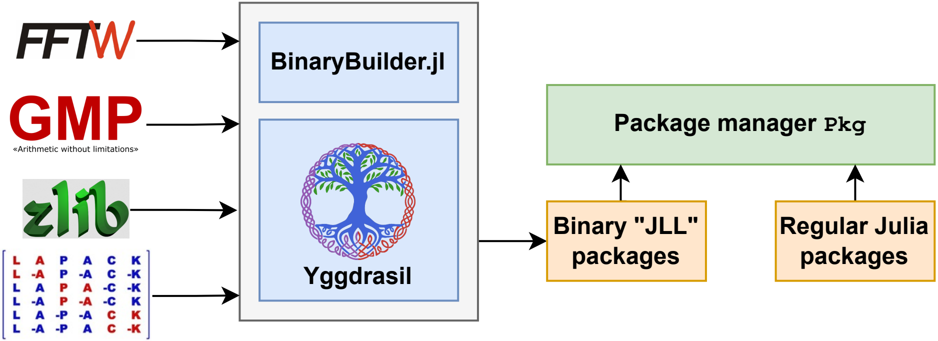
Workflow: binary dependency management in Julia



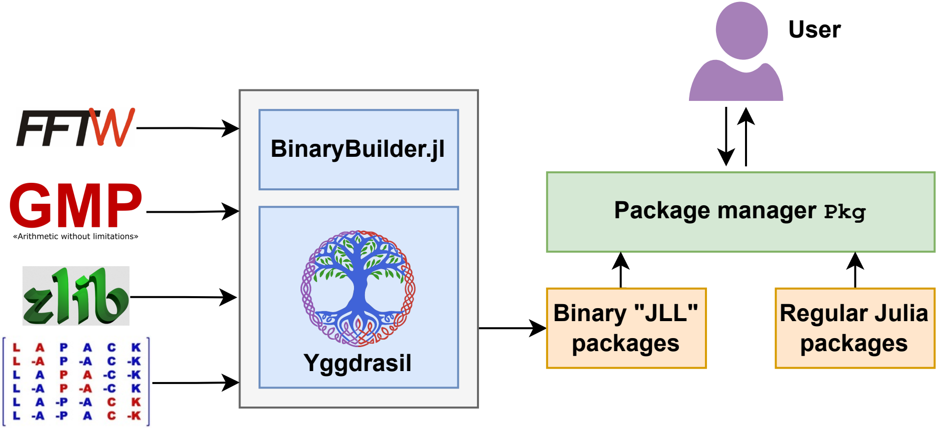
Workflow: binary dependency management in Julia



Workflow: binary dependency management in Julia



Workflow: binary dependency management in Julia



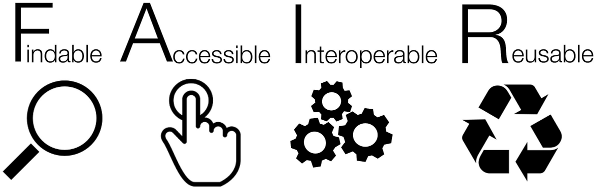
Caveat for HPC: using Julia with a system libraries

- HPC architectures often require using vendor-provided software, e.g., MPI, CUDA
- Need to replace MPI-enabled JLL binaries by system binaries (e.g., MPI.jl, HDF5.jl)
- May need to regenerate C bindings for libraries due to MPI ABI change

Remedies:

- Write/use wrapper packages with corresponding logic
- Have a look at MPItrampoline (`https://github.com/eschnett/MPItrampoline`)
- Document setup procedures for your users and yourself

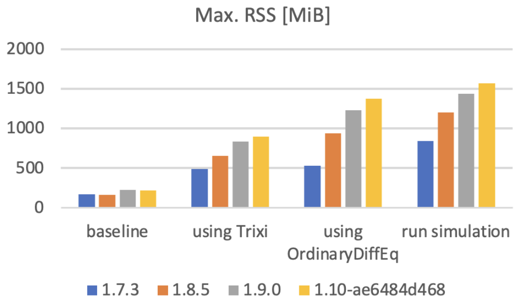
FAIR data/code principles



- **Findable**: general registry, JuliaHub, Discourse, Slack, Zulip
- **Accessible**: open source, package hosting on GitHub, Documenter.jl
- **Interoperable**: design around informal interfaces, duck typing
- **Reusable**: package structure and environments, semantic versioning

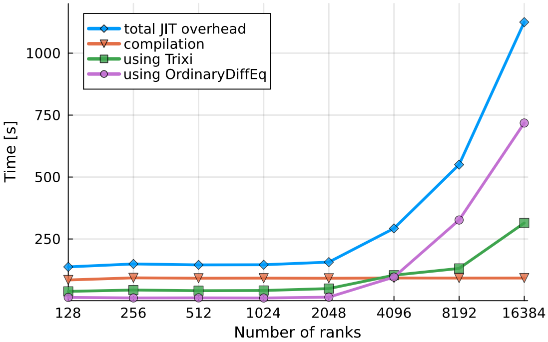
Memory usage in Julia

- Julia memory usage has been increasing since Julia v1.6
- Problematic for MPI-only parallel codes
- Less problematic for hybrid codes, e.g., MPI+threads or MPI+GPU



Startup latency: challenge for parallel execution

- Compilation time remains constant
- Loading time increases for >2000 MPI ranks
- Julia depot on parallel filesystem (GPFS)
- Parallel I/O becomes bottleneck



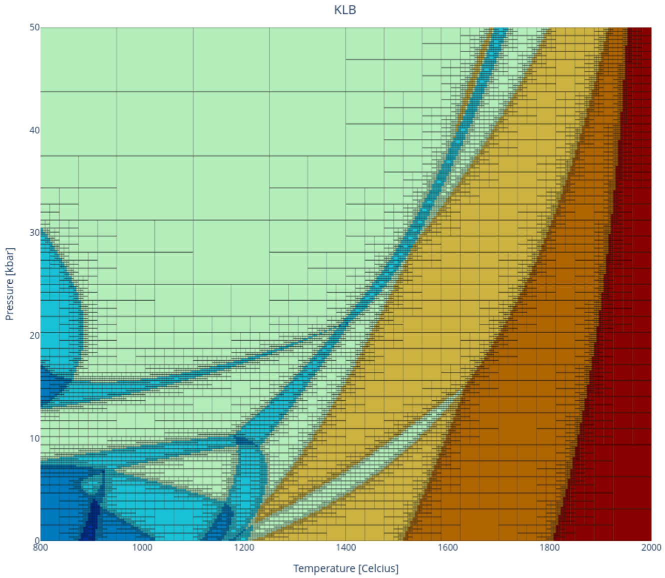
(lower is better)

Julia is promising for scientific computing

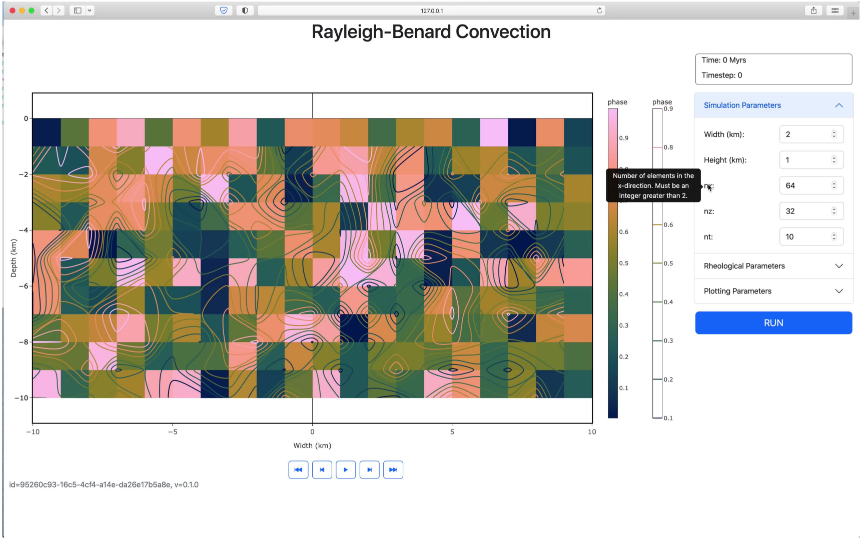
Julia hackathon in September

- C code MAGEMin (several years of development)
- Adaptive meshes with C/C++ code `t8code` wrapped in T8code.jl
- Parallelization via `Base.Threads`
- Coupling, GUI, and visualization in Julia

$\longrightarrow$ $1.5\times$ faster than previous MATLAB version (2 days)



Julia GUI and interface to LaMEM



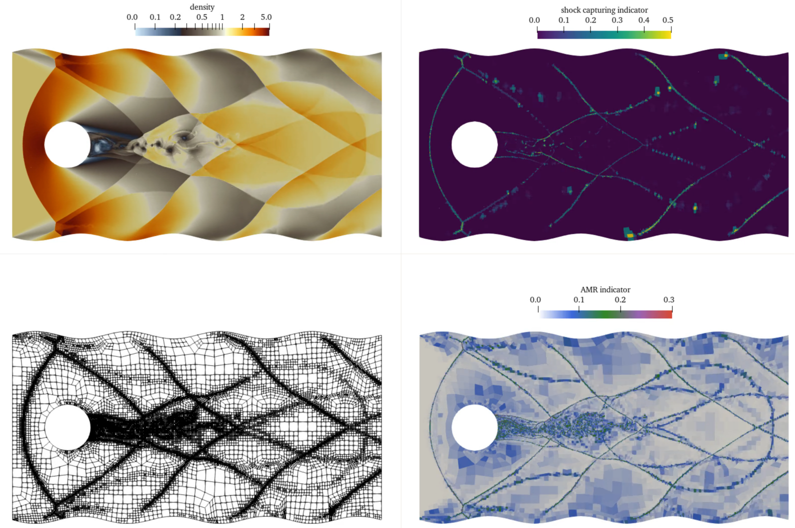
Credit: Boris Kaus et al.

Open source software Trixi.jl

- Adaptive high-order simulation framework for conservation laws (MIT license)
- Goals: usability, extensibility, performance
- Integration with Julia ecosphere:
  - OrdinaryDiffEq.jl: time integration
  - ForwardDiff.jl: automatic differentiation
  - Plots.jl, Makie.jl: plotting
  - LoopVectorization.jl: performance
  - Polyester.jl: multithreading
  - MPI.jl: distributed parallelism

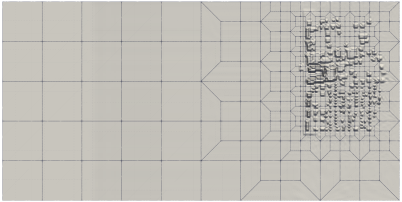
https://github.com/trixi-framework/Trixi.jl

Supersonic flow with curved, adaptive mesh



Credit: Andrew R. Winters et al. with Trixi.jl

Shallow water simulation of Seaside, Oregon, US



Credit: Andrew R. Winters, Sven Goldberg, Maximilian Bertrandt et al. with Trixi.jl

Summary

- Julia is promosing for scientific computing
  - from laptops to HPC systems
  - from experimental code to international collaborations
  - encouraging good research software engineering practices
- Julia is not perfect
  - but it’s actively developed
  - it requires some effort to use it well

Summary

- Julia is promosing for scientific computing
  - from laptops to HPC systems
  - from experimental code to international collaborations
  - encouraging good research software engineering practices
- Julia is not perfect
  - but it’s actively developed
  - it requires some effort to use it well

`https://ranocha.de`

Thank you

References I

Baker, M. (2016). “1,500 scientists lift the lid on reproducibility.” In: *Nature* 533, pp. 452–454. DOI: 10.1038/533452a.

Churavy, V., W. F. Godoy, C. Bauer, H. Ranocha, M. Schlottke-Lakemper, L. Räss, J. Blaschke, M. Giordano, E. Schnetter, S. Omlin, J. S. Vetter, and A. Edelman (Nov. 2022). *Bridging HPC Communities through the Julia Programming Language*. DOI: 10.48550/arXiv.2211.02740. arXiv: 2211.02740 [cs.DC].

Ketcheson, D. I. and H. Ranocha (June 2023). “Computing with B-series.” In: *ACM Transactions on Mathematical Software* 49.2. DOI: 10.1145/3573384. arXiv: 2111.11680 [math.NA].

Ranocha, H., M. Schlottke-Lakemper, J. Chan, A. M. Rueda-Ramírez, A. R. Winters, F. Hindenlang, and G. J. Gassner (Dec. 2021). *Efficient implementation of modern entropy stable and kinetic energy preserving discontinuous Galerkin methods for conservation laws*. Accepted in ACM Transactions on Mathematical Software (TOMS), 2023. arXiv: 2112.10517 [cs.MS].

Ranocha, H., M. Schlottke-Lakemper, A. R. Winters, E. Faulhaber, J. Chan, and G. J. Gassner (Jan. 2022). “Adaptive numerical simulations with Trixi.jl: A case study of Julia for scientific computing.” In: *Proceedings of the JuliaCon Conferences* 1.1, p. 77. DOI: 10.21105/jcon.00077. arXiv: 2108.06476 [cs.MS].

Schlottke-Lakemper, M., A. R. Winters, H. Ranocha, and G. J. Gassner (June 2021). “A purely hyperbolic discontinuous Galerkin approach for self-gravitating gas dynamics.” In: *Journal of Computational Physics* 442, p. 110467. DOI: 10.1016/j.jcp.2021.110467. arXiv: 2008.10593 [math.NA].