Julia for adaptive high-order multi-physics simulations

Michael Schlottke-Lakemper

27th January 2021

Department of Mathematics and Computer Science,
University of Cologne

Acknowledgments

- Andrew Winters, Linköping University, Sweden (website)
  - especially coupled Euler-gravity simulations
- Hendrik Ranocha, KAUST, Saudia Arabia (website)
- Gregor Gassner, University of Cologne, Germany (website)

Back in January 2020. . .

Back in January 2020...

- FLUXO: fast, parallel Fortran 3D-DGSEM code for curvilinear compressible Euler & MHD simulations (`github.com/project-fluxo/fluxo`)

Back in January 2020…

- FLUXO: fast, parallel Fortran 3D-DGSEM code for curvilinear compressible Euler & MHD simulations (`github.com/project-fluxo/fluxo`)

- Many one-trick ponies: codes with a singular purpose, often unusable for anything else, discarded after use



Images: ©MJ Boswell (top), ©Clive Williams (bottom)

What should we do? Write more code!

- Plan: use hackathon to write new simulation framework
  - should be useful to scientists and students
  - extensible and fast
- But we want even more:
  - easy to use for inexperienced users
  - hassle-free toolchain (= installation & postprocessing)
  - potential for HPC (maybe)
- Main question: Which language to use? Fortran? C++? ~~Python?~~ Julia?

What is Julia?

According to the official website Julia is . . .

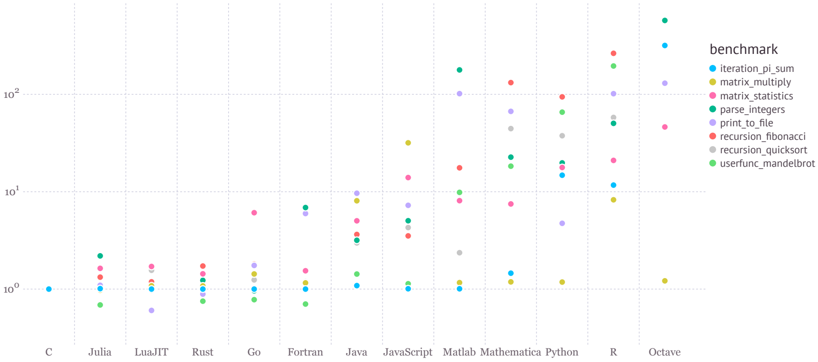
- fast
- dynamic
- reproducible
- composable
- general
- open source

`https://github.com/JuliaLang/julia-logo-graphics`

Source: `https://julialang.org`

Julia is *fast*

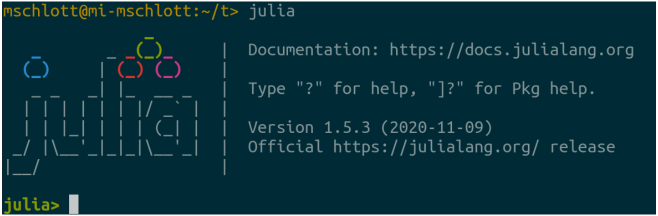
- "Just-Ahead-Of-Time" (JAOT) compilation from Julia to machine code
- Facilitated by LLVM infrastructure
- Built-in support for parallelism (GPUs, shared memory, distributed)



https://julialang.org/benchmarks/

Julia is *dynamic*

- Dynamic type system
- Automatic memory management (garbage collection)
- Focus on interactive use



Julia's interactive *read-eval-print-loop* (REPL)

Julia is *reproducible*

- Create reproducible software environments across platforms
- Built-in package manager `Pkg` handles dependencies and versioning
- Automatic provisioning of pre-built binaries

Project.toml

```
name = "Trixi"
uuid = "a7f1ee26-1774-49b1-8366-f1abc58fbfcb"
authors = [...]
version = "0.3.9-pre"

[deps]
EllipsisNotation = "da5c29d0-fa7d-589e-88eb-ea29b0a81949"
LinearMaps = "7a12625a-238d-50fd-b39a-03d52299707e"

[compat]
EllipsisNotation = "1.0"
LinearMaps = "2.7, 3.0"
```

Which language should we use?

Battle of Languages

C++	Fortran	Julia
Pro 1. Auf allen ernsthaften Maschinen installiert 2. Sehr schnell 3. Ausdrucksstark: komplexe Operationen lassen sich kompakt darstellen 4. Gute Standardbibliothek 5. Gute Kompatibilität zu externen Libraries 6. Statisch kompiliert -> Compiler hilft bei Fehlersuche	**Pro** 1. Auf allen ernsthaften Maschinen installiert 2. Sehr schnell 3. Einfache Syntax -> wenig Fehlerpotenzial 4. Alle in Gruppe sind “Experten” 5. Statisch kompiliert -> Compiler hilft bei Fehlersuche	**Pro** 1. “Sexy” für Studenten 2. Im Kern einfach zu erlernen (Matlab-ähnlich) 3. Weniger “boilerplate” Code (vs. Fortran) 4. Gut für schnelles Prototyping 5. Einfach zu installieren (auch Laptop) 6. Große Paket-Bibliothek -> viel Funktionalität in Julia-only 7. Neuheitswert (ggf. auch wissenschaftlich?) 8. Bei Erfolg: echtes Alleinstellungsmerkmal 9. Gute (nachgesagte) hybride Parallelisierung
Con 1. Kann kein Student 2. Mittelmäßig sexy für Studenten 3. Nur ein “Experte” in der Gruppe 4. Viele Wege, etwas falsch zu machen 5. Nicht memory-safe	**Con** 1. Kann kein Student 2. Super unsexy für Studenten 3. Sehr viel Boilerplate 4. Mittelmäßige Portierbarkeit 5. Schlechtes Interfacing mit C Bibliotheken (Wrapper notwendig)	**Con** 1. Kann kein Student 2. Kein Experte in der Gruppe 3. Zwingt (!) bestimmte, ungewohnte Programmierparadigmen zu nutzen 4. (Sehr) langsamer Startup von sogar kleinen Programmen, da immer erst kompiliert werden muss 5. Programmierfehler erst zur Laufzeit sichtbar 6. Unausgereifte Toolchain: z.B. Fehlermeldungen nicht so hilfreich 7. Kleine Community, wenige Experten greifbar für uns 8. Schlechtere Unterstützung auf ernsthaften Maschinen (wenig Erfahrung) 9. Viele Feinheiten, die nicht auf den ersten Blick offensichtlich sind 10. Mit Garbage Collection etc. sind a priori Performanceabschätzungen schwieriger

`LanguageBattle.gdoc`

Which language should we use?

Battle of Languages

C++	Fortran	Julia
Pro 1. Auf allen ernsthaften Maschinen installiert 2. Sehr schnell 3. Ausdrucksstark: komplexe Operationen lassen sich kompakt darstellen 4. Gute Standardbibliothek 5. Gute Kompatibilität zu externen Libraries 6. Statisch kompiliert -> Compiler hilft bei Fehlersuche	**Pro** 1. Auf allen ernsthaften Maschinen installiert 2. Sehr schnell 3. Einfache Syntax -> wenig Fehlerpotenzial 4. Alle in Gruppe sind "Experten" 5. Statisch kompiliert -> Compiler hilft bei Fehlersuche	**Pro** 1. "Sexy" für Studenten 2. Im Kern einfach zu erlernen (Matlab-ähnlich) 3. Weniger "boilerplate" Code (vs. Fortran) 4. Gut für schnelles Prototyping 5. Einfach zu installieren (auch Laptop) 6. Große Paket-Bibliothek -> viel Funktionalität in Julia-only 7. Neuheitswert (ggf. auch wissenschaftlich?) 8. Bei Erfolg: echtes Alleinstellungsmerkmal 9. Gute (nachgesagte) hybride Parallelisierung
C[illegible] 1. [illegible] Student 2. Mittelmäßig sexy für Studenten 3. Nur ein "Experte" in der Gruppe 4. Viele Wege, etwas falsch zu machen 5. Nicht memory-safe	**Con** 1. Kan[illegible] Student 2. Super unsexy für Studenten 3. Sehr viel Boilerplate 4. Mittelmäßige Portierbarkeit 5. Schlechtes Interfacing mit C Bibliotheken (Wrapper notwendig)	**C**[illegible] 1. [illegible] Student 2. Kein Experte in der Gruppe 3. Zwingt (!) bestimmte, ungewohnte Programmierparadigmen zu nutzen 4. (Sehr) langsamer Startup von sogar kleinen Programmen, da immer erst kompiliert werden muss 5. Programmierfehler erst zur Laufzeit sichtbar 6. Unausgereifte Toolchain: z.B. Fehlermeldungen nicht so hilfreich 7. Kleine Community, wenige Experten greifbar für uns 8. Schlechtere Unterstützung auf ernsthaften Maschinen (wenig Erfahrung) 9. Viele Feinheiten, die nicht auf den ersten Blick offensichtlich sind 10. Mit Garbage Collection etc. sind a priori Performanceabschätzungen schwieriger

too complicated

too unsexy

maybe interesting!

LanguageBattle.gdoc

Outline of this talk

1. Should we use Julia?
2. Hyperbolic self-gravitating gas dynamics
3. Julia in practice: Trixi.jl
   - Live demonstration
4. Evaluating Julia for scientific computing
5. Conclusions and outlook

Hyperbolic self-gravitating gas dynamics

Goal: Approximate self-gravitating gas dynamics

- Compressible Euler equations for hydrodynamics

$$\frac{\partial}{\partial t}\begin{bmatrix}\rho\\ \rho v_1\\ \rho v_2\\ E\end{bmatrix}+\frac{\partial}{\partial x}\begin{bmatrix}\rho v_1\\ \rho v_1^2+p\\ \rho v_1 v_2\\ (E+p)v_1\end{bmatrix}+\frac{\partial}{\partial y}\begin{bmatrix}\rho v_2\\ \rho v_1 v_2\\ \rho v_2^2+p\\ (E+p)v_2\end{bmatrix}=\begin{bmatrix}0\\ -\rho\phi_x\\ -\rho\phi_y\\ -(\vec{v}\cdot\vec{\nabla}\phi)\rho\end{bmatrix}$$

- Newtonian potential equation for gravitation

$$-\vec{\nabla}^2\phi=-4\pi G\rho$$

- PDE for hydrodynamics is hyperbolic whereas gravity is elliptic

- Coupling of the two equations entirely through source terms

There is so much potential...

- Let's manipulate the Poisson equation in 2D

$$-\nu\vec{\nabla}^2 u = f$$

- Potential u is the steady state solution of a parabolic equation

$$u_t - \nu\vec{\nabla}^2 u = f$$

- Introduce variables $\vec{\nabla} u = (q_1\,,\, q_2)^T$ to have a parabolic system

$$\begin{aligned} u_t - \nu[q_1]_x - \nu[q_2]_y &= f \\ u_x &= q_1 \\ u_y &= q_2 \end{aligned}$$

There is so much potential...

- Diffusion equation has paradox of instant propogation
- Idea of Cattaneo, introduce (small) time scale T_r

$$
\begin{aligned}
u_t - \nu[q_1]_x - \nu[q_2]_y &= f \\
T_r[q_1]_t - u_x &= -q_1 \\
T_r[q_2]_t - u_y &= -q_2
\end{aligned}
$$

to correct unphysical behavior

- Yields the hyperbolic diffusion equations

$$
\frac{\partial}{\partial t}\begin{bmatrix} u \\ q_1 \\ q_2 \end{bmatrix} + \frac{\partial}{\partial x}\begin{bmatrix} -\nu q_1 \\ -u/T_r \\ 0 \end{bmatrix} + \frac{\partial}{\partial y}\begin{bmatrix} -\nu q_2 \\ 0 \\ -u/T_r \end{bmatrix} = \begin{bmatrix} f(x,y) \\ -q_1/T_r \\ -q_2/T_r \end{bmatrix}
$$

Numerical solver for hyperbolic conservation laws

- Select discontinuous Galerkin (DG) to approximate solution of general conservation law system

$$\mathbf{u}_t + \mathbf{f}_x(\mathbf{u}) = \mathbf{s}(\mathbf{u})$$

- Divide spatial domain into elements, map to reference element
- Multiply by test function and integrate-by-parts (IBP) to obtain weak form $\rightarrow$ optionally use IBP again to get strong form
- Approximate solution, fluxes, sources with nodal polynomials
- Resolve discontinuities at element boundaries by numerical flux
- Approximate integrals with Gauss-type quadrature
- Interpolation and quadrature nodes are collocated

Numerical approximation: Discontinuous Galerkin

- Gives an ODE to integrate in time and update the approximate solution in each element

$$\boxed{\frac{\mathrm{d}\mathbf{U}_j}{\mathrm{d}t} = -\frac{2}{\Delta x_i}\left\{\frac{\delta_{jN}}{\omega_N}\left[\mathbf{F}^* - \mathbf{F}_N\right] - \frac{\delta_{j0}}{\omega_0}\left[\mathbf{F}^* - \mathbf{F}_0\right] + \sum_{m=0}^{N}\mathcal{D}_{jm}\mathbf{F}_m\right\} + \mathbf{S}_j}$$

- Here $j = 0, \ldots, N$ and $\mathcal{D}_{jm} = \ell'_m(\xi_j)$ is the polynomial derivative matrix

- Use explicit time integration via Runge-Kutta (RK) methods

- Stable DG time step has the form

$$\Delta t = \frac{\mathtt{CFL}}{N+1}\frac{\Delta x}{|\lambda_{\max}|}$$

with adjustable CFL constant

Verify high-order DG method for hyperbolic diffusion

- Integrate in time with "standard" five-stage, four-order low-storage RK method of Carpenter & Kennedy
- Take $\texttt{CFL} = 0.5$ such that spatial errors dominate
- Threshold to define steady state taken as $\texttt{tol} = 10^{-10}$
- Domain $\Omega = [0,1]^2$ with known Poisson solution

$$u(x,y) = 2 + 2\cos(\pi x)\sin(2\pi y) \qquad f(x,y) = 10\pi^2\cos(\pi x)\sin(2\pi y)$$

- Boundary conditions: Dirichlet in x-direction, periodic in y-direction

Convergence of hyperbolic diffusion

- $N = 3$

K	$L^2(u)$	$L^2(q_1)$	$L^2(q_2)$
4^2	3.15E-03	1.24E-02	2.19E-02
8^2	2.26E-04	8.83E-04	1.50E-03
16^2	1.50E-05	5.51E-05	9.68E-05
32^2	9.65E-07	3.32E-06	6.14E-06
avg. EOC	3.89	3.96	3.93

- $N = 4$

K	$L^2(u)$	$L^2(q_1)$	$L^2(q_2)$
4^2	2.51E-04	8.81E-04	1.63E-03
8^2	8.52E-06	2.88E-05	5.45E-05
16^2	2.77E-07	9.12E-07	1.76E-06
32^2	8.85E-09	2.85E-08	5.60E-08
avg. EOC	4.93	4.97	4.94

- Discrete L^2 errors computed on uniform Cartesian meshes of increasing resolution
- Demonstrate high-order accuracy of two polynomial orders for potential u and its gradient

Revisit equations of self-gravitating gas dynamics

- Compressible Euler equations

$$\frac{\partial}{\partial t}\begin{bmatrix}\rho\\ \rho v_1\\ \rho v_2\\ E\end{bmatrix}+\frac{\partial}{\partial x}\begin{bmatrix}\rho v_1\\ \rho v_1^2+p\\ \rho v_1 v_2\\ (E+p)v_1\end{bmatrix}+\frac{\partial}{\partial y}\begin{bmatrix}\rho v_2\\ \rho v_1 v_2\\ \rho v_2^2+p\\ (E+p)v_2\end{bmatrix}=\begin{bmatrix}0\\ -\rho q_1\\ -\rho q_2\\ -(v_1 q_1+v_2 q_2)\rho\end{bmatrix}$$

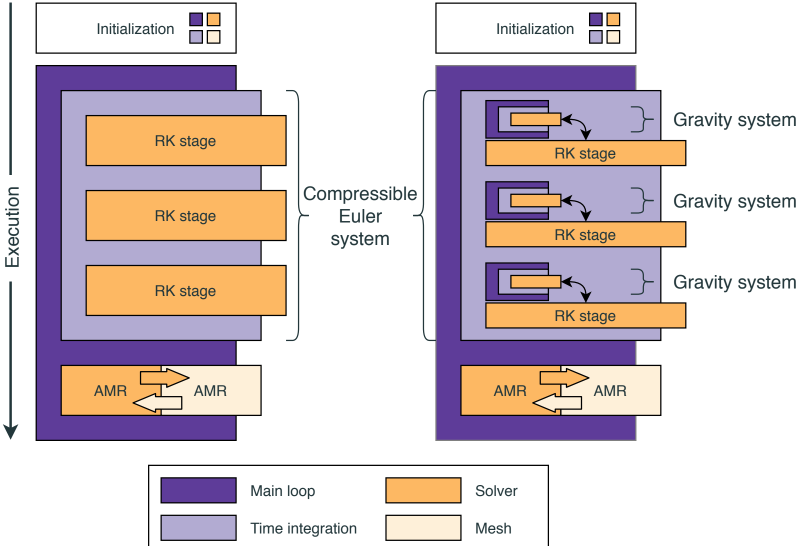
- Recast gravitational potential equation

$$\frac{\partial}{\partial t}\begin{bmatrix}\phi\\ q_1\\ q_2\end{bmatrix}+\frac{\partial}{\partial x}\begin{bmatrix}-q_1\\ -\phi/T_r\\ 0\end{bmatrix}+\frac{\partial}{\partial y}\begin{bmatrix}-q_2\\ 0\\ -\phi/T_r\end{bmatrix}=\begin{bmatrix}-4\pi G\rho\\ -q_1/T_r\\ -q_2/T_r\end{bmatrix}$$

- PDEs for dynamics are both hyperbolic and coupled via source terms

Schematic of volumetric coupling

- Single-physics simulation
- Multi-physics simulation



Creating a multi-physics simulation

- **Instantiate two DG solvers**: One for hydrodynamics, one for gravity
- Independent and only "talk" through the source terms
- For simplicity assume the two solvers share a mesh
- Compressible Euler solver can have other features, shock capturing or AMR
- **Use different time integrators**:
  - Compressible Euler uses five-stage, fourth-order low-storage RK scheme of Carpenter & Kennedy
  - Gravity uses five-stage, second-order low-storage RK scheme optimized to allow large explicit time steps
  - Two adjustable coefficients for time step selection: $\texttt{CFL}_{\mathrm{Eu}} \in (0,1]$, $\texttt{CFL}_{\mathrm{Gr}}$

Verify high-order DG method for coupled problem

- Take $\texttt{CFL}_{\mathrm{Eu}} = \texttt{CFL}_{\mathrm{Gr}} = 0.5$ such that spatial errors dominate
- Threshold to define steady state gravity is $\texttt{tol} = 10^{-10}$
- Domain $\Omega = [0, 2]^2$ with manufactured solution

  $$\rho = 2 + \frac{1}{10}\sin(\pi(x + y - t)) \quad v_1 = v_2 = 1 \quad p = \frac{1}{\pi}\rho^2 \quad \phi = -\frac{2}{\pi}(\rho - 2)$$

  Gravitational constant $G = 1$
- Boundary conditions: Periodic in all directions
- Introduces additional residual terms added to the right-hand-side

Convergence of coupled self-gravity problem

- $N = 3$

K	$L^2(\rho)$	$L^2(\rho v_1)$	$L^2(\rho v_2)$	$L^2(E)$	$L^2(\phi)$	$L^2(q_1)$	$L^2(q_2)$
4^2	4.37E-04	4.69E-04	4.69E-04	9.72E-04	1.64E-04	8.33E-04	8.33E-04
8^2	2.43E-05	2.60E-05	2.60E-05	5.09E-05	9.90E-06	5.65E-05	5.65E-05
16^2	1.06E-06	1.37E-06	1.37E-06	2.65E-06	6.63E-07	3.77E-06	3.77E-06
32^2	4.73E-08	8.03E-08	8.03E-08	1.56E-07	4.33E-08	2.44E-07	2.44E-07
avg. EOC	4.39	4.17	4.17	4.20	3.96	3.91	3.91

- $N = 4$

K	$L^2(\rho)$	$L^2(\rho v_1)$	$L^2(\rho v_2)$	$L^2(E)$	$L^2(\phi)$	$L^2(q_1)$	$L^2(q_2)$
4^2	3.50E-05	3.38E-05	3.38E-05	6.59E-05	1.15E-05	6.31E-05	6.31E-05
8^2	7.99E-07	9.00E-07	9.00E-07	1.71E-06	3.74E-07	2.11E-06	2.11E-06
16^2	1.95E-08	2.49E-08	2.49E-08	4.78E-08	1.23E-08	6.95E-08	6.95E-08
32^2	5.31E-10	7.73E-10	7.73E-10	1.44E-09	4.03E-10	2.25E-09	2.25E-09
avg. EOC	5.34	5.14	5.14	5.16	4.93	4.93	4.93

- Retain high-order accuracy for all variables in the coupled system

A more physical self-gravitating setup

- Jeans instability models perturbations and interactions between a gas cloud and gravity
- Consider a background state in centimeter-gram-second (CGS) units

$$\rho_0 = 1.5 \cdot 10^7 \text{ [g cm}^{-3}\text{]} \qquad p_0 = 1.5 \cdot 10^7 \text{ [dyn cm}^{-2}\text{]}$$

- Perturb a particular mode $\vec{k}$ with small amplitude δ_0

$$\rho = \rho_0 \left(1 + \delta_0 \cos(\vec{k} \cdot \vec{x})\right) \qquad \vec{v} = \vec{0} \qquad p = p_0 \left(1 + \delta_0 \gamma \cos(\vec{k} \cdot \vec{x})\right)$$

- Gravitational field responds accordingly

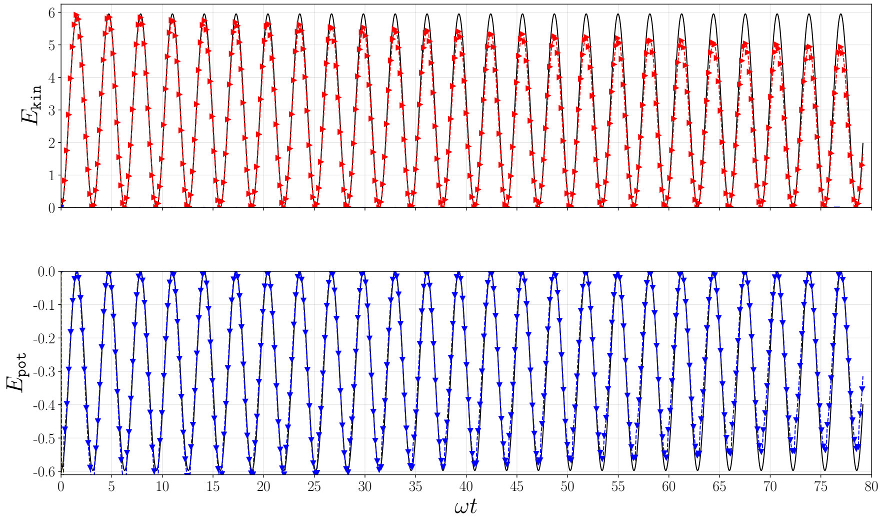
$$-\vec{\nabla}^2 \phi = -4\pi G(\rho - \rho_0) \qquad G = 6.674 \cdot 10^{-8} \text{ [cm}^3 \text{ g}^{-1} \text{ s}^{-2}\text{]}$$

A more physical self-gravitating setup (continued)

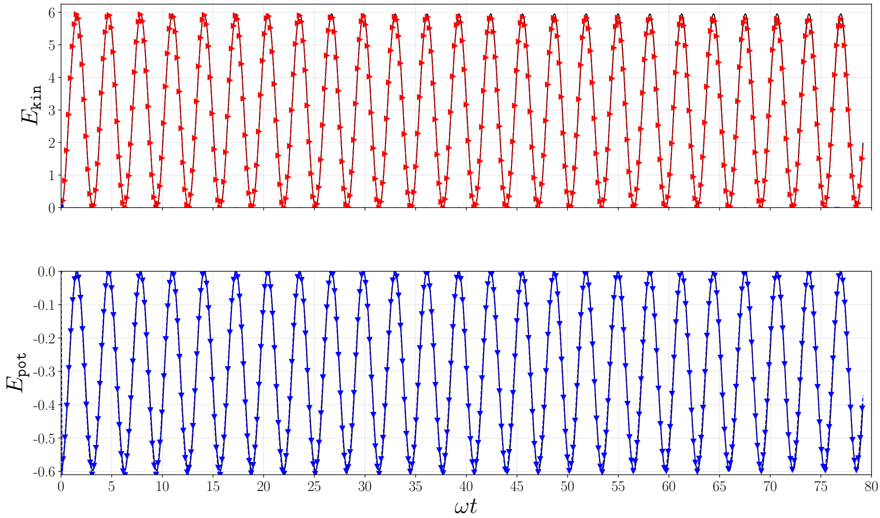
- Domain $\Omega = [0, 1]^2$ with periodic boundary conditions
- Take $\texttt{CFL}_{\mathrm{Eu}} = 0.5$ and $\texttt{CFL}_{\mathrm{Gr}} = 1.2$
- Uniform 16×16 Cartesian mesh with polynomial order $N = 3$
- Threshold to define steady state taken as $\texttt{tol} = 10^{-4}$
- Offers a convenient bridge to examine numerics: More physically relevant but still has analytical expressions for energy evolution

$$E_{\mathtt{kin}} = \int_\Omega \frac{\rho}{2}(v_1^2 + v_2^2)\,\mathrm{d}\Omega \qquad E_{\mathtt{int}} = \int_\Omega \frac{p}{\gamma - 1}\,\mathrm{d}\Omega \qquad E_{\mathtt{pot}} = \int_\Omega \rho\phi\,\mathrm{d}\Omega$$

Energy evolution of Jeans instability (couple every RK step)

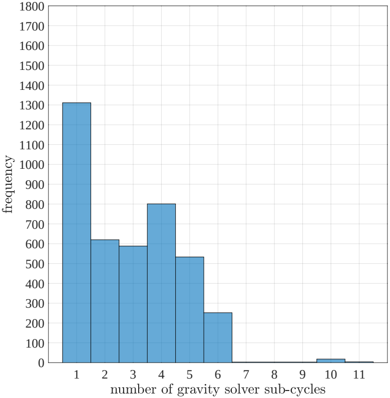


Energy evolution of Jeans instability (couple every RK stage)



Visualize cost of gravity solver for Jeans instability

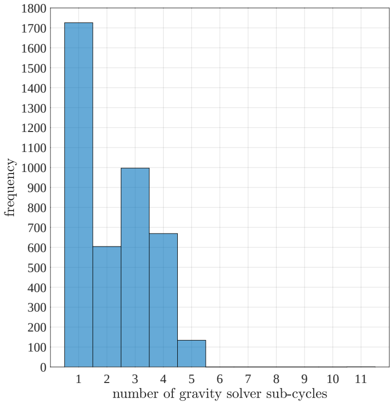
- Gravity sub-cycle: Assembly and evolution of hyperbolic gravity system by one complete time step



- Explicit time integration with low storage RK
- **Compressible Euler** solver uses five-stage, fourth order scheme of Carpenter & Kennedy $\mathtt{CFL}_{\mathrm{Eu}} = 0.5$
- **Hyperbolic gravity** uses the same time integration scheme as compressible Euler $\mathtt{CFL}_{\mathrm{Gr}} = 0.8$

Visualize cost of gravity solver for Jeans instability

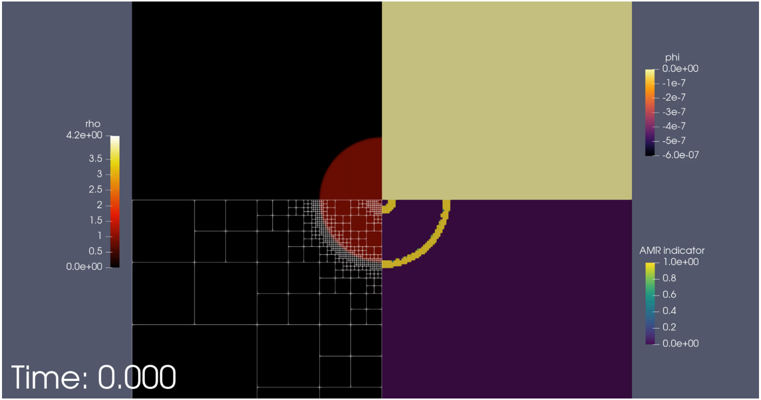
- Gravity sub-cycle: Assembly and evolution of hyperbolic gravity system by one complete time step



- Explicit time integration with low storage RK
- **Compressible Euler** solver uses five-stage, fourth order scheme of Carpenter & Kennedy
  $\mathtt{CFL}_{\mathrm{Eu}} = 0.5$
- **Hyperbolic gravity** uses five-stage, second order RK scheme optimized to take larger times steps
  $\mathtt{CFL}_{\mathrm{Gr}} = 1.2$

Bring everything together

- Sedov explosion problem with self-gravity
- Localize explosion to occur within a dense disc of radius one
- Contains strong shocks and necessitates AMR

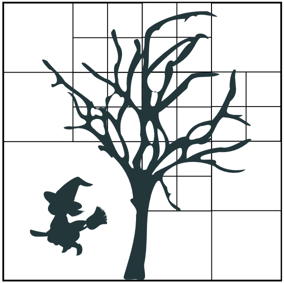


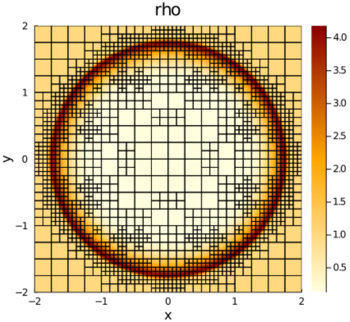
Schlottke-Lakemper, Winters, Ranocha, Gassner, JPC, 2020 (submitted). `arXiv:2008.10593`

Julia in practice: Trixi.jl

Trixi.jl: A tree-based numerical simulation framework for (hyperbolic) PDEs

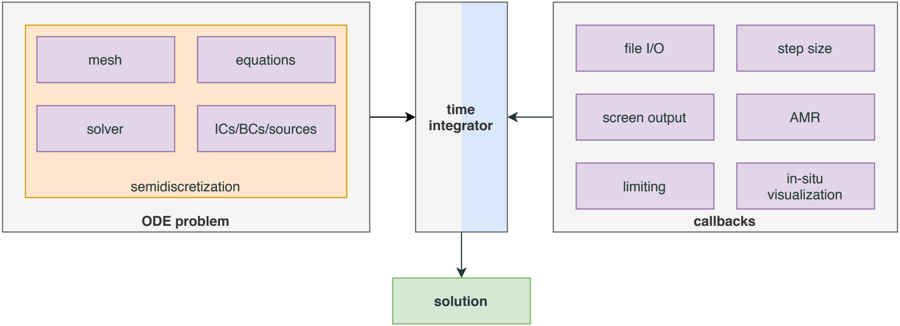
- Adaptive hierarchical quadtree/octree grids
- Nodal discontinuous Galerkin spectral element methods
- Explicit time integration with SciML's `OrdinaryDiffEq.jl`
- Multiple governing equations
- Available at `github.com/trixi-framework/Trixi.jl` (open source)





Launch your Binders! launch binder

Overall structure of Trixi

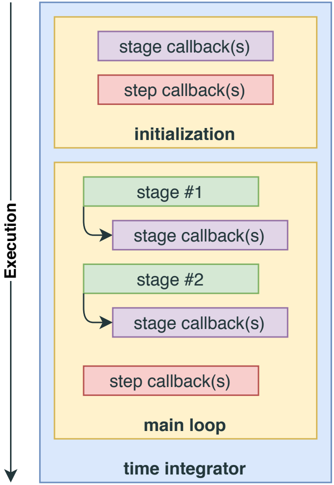


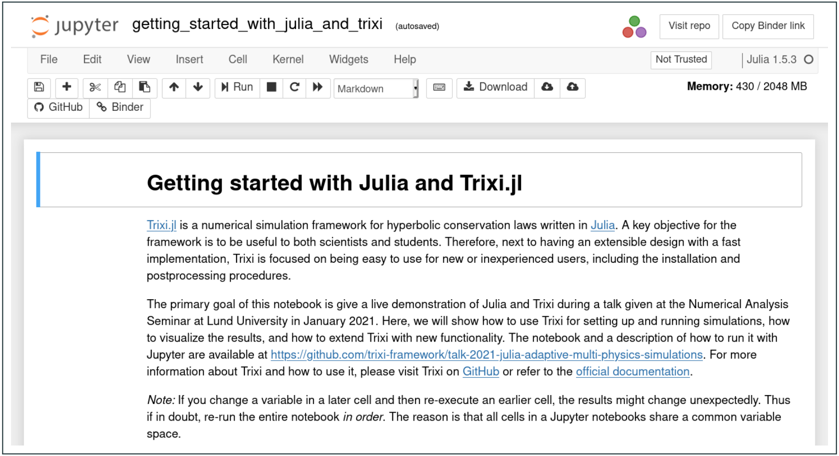
Guiding principles for the framework:

- Modular architecture ("*Trixi as a library*")
- Only read code that you actually use
- Implement first, ask questions later

Extended functionality in callbacks

- Callbacks provide additional functionality
  - file I/O
  - step size
  - adaptive mesh refinement
  - limiting
  - in-situ visualization
  - ...
- Flexible: combine callbacks for applications
- Extensible: easily add new features





Getting started with Julia and Trixi.jl

Trixi.jl is a numerical simulation framework for hyperbolic conservation laws written in Julia. A key objective for the framework is to be useful to both scientists and students. Therefore, next to having an extensible design with a fast implementation, Trixi is focused on being easy to use for new or inexperienced users, including the installation and postprocessing procedures.

The primary goal of this notebook is give a live demonstration of Julia and Trixi during a talk given at the Numerical Analysis Seminar at Lund University in January 2021. Here, we will show how to use Trixi for setting up and running simulations, how to visualize the results, and how to extend Trixi with new functionality. The notebook and a description of how to run it with Jupyter are available at https://github.com/trixi-framework/talk-2021-julia-adaptive-multi-physics-simulations. For more information about Trixi and how to use it, please visit Trixi on GitHub or refer to the official documentation.

Note: If you change a variable in a later cell and then re-execute an earlier cell, the results might change unexpectedly. Thus if in doubt, re-run the entire notebook *in order*. The reason is that all cells in a Jupyter notebooks share a common variable space.

Evaluating Julia for scientific computing

Evaluating Julia for scientific computing

- What about performance?
- What about ease of use?
- What about reproducibility?
- What about composability?
- What else is there?

What about *performance*?

- Serial performance is good
  - FLUXO vs. FLUXO-in-Julia-v0.6: within factor of $2 - 3\times$
  - FLUXO vs. Trixi: Trixi can be faster (caveat: no metric terms)
- Requires new performance intuition (when coming from C/C++/Fortran)
  - use many small functions (function barriers)
  - type instabilities $\to$ Python-like performance
  - benchmarking new code is a must
- Parallel performance for simulation science? The verdict is still out...
  - no true MPI HPC codes out there (yet)
  - possibly first large-scale project: `https://github.com/CliMA/ClimateMachine.jl`
  - petascale showcase (`Celeste.jl`) is *not* a simulation

Just-ahead-of-time compilation has its quirks – and perks!

- Very long startup times: only usable with REPL (caching)
  - Time to first result in Trixi: $\sim$20 seconds
  - Time to second result: 60 *milli*seconds
- Compilation is serial

Just-ahead-of-time compilation has its quirks – and perks!

- Very long startup times: only usable with REPL (caching)
  - Time to first result in Trixi: ∼20 seconds
  - Time to second result: 60 *milli*seconds
- Compilation is serial
- Kernels can be written as in C/C++/Fortran (loops are fast!)
- Extremely high degree of function specialization
  (compare to fully templated C++)

Just-ahead-of-time compilation has its quirks – and perks!

- Very long startup times: only usable with REPL (caching)
  - Time to first result in Trixi: ∼20 seconds
  - Time to second result: 60 *milli*seconds
- Compilation is serial
- Kernels can be written as in C/C++/Fortran (loops are fast!)
- Extremely high degree of function specialization (compare to fully templated C++)

ClangJIT: Enhancing C++ with Just-in-Time Compilation

Hal Finkel
Lead, Compiler Technology and Programming Languages
Leadership Computing Facility
Argonne National Laboratory
Lemont, IL, USA
hfinkel@anl.gov

David Poliakoff
Lawrence Livermore National Laboratory
Livermore, CA, USA
poliakoff1@llnl.gov

David F. Richards
Lawrence Livermore National Laboratory
Livermore, CA, USA
richards12@llnl.gov

ABSTRACT

The C++ programming language is not only a keystone of the high-performance-computing ecosystem but has proven to be a

body of C++ code, but critically, defer the generation and optimization of template specializations until runtime using a relatively-natural extension to the core C++ programming language.

arXiv:1904.08555

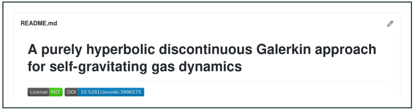
What about *ease of use*?

- Very easy to get up and running (see reproducibility)
- Code can be very simple (MATLAB-like) if desired
  - but be careful about unwanted allocations (type instabilities)
- Lack of a mature toolchain
  - Plotting is only OK
  - No widespread IDE support (Julia plugin for VS Code)
  - Only one debugger
- No surprise: truly fast code looks virtually the same everywhere

What about *reproducibility*?

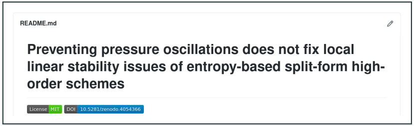
```julia
import Pkg
Pkg.activate()
Pkg.instantiate()

using Trixi # ... and enjoy!
```

- Provisioning reproducible compute environments is straightforward
- Two files provide all relevant information (`Project.toml`, `Manifest.toml`)
- Recreation with only a few lines of code
- Excellent for reproducible science: paper 1, paper 2, this talk's repo


README.md

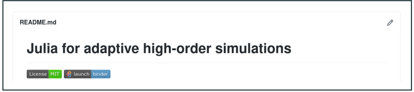
A purely hyperbolic discontinuous Galerkin approach for self-gravitating gas dynamics

License MIT DOI 10.5281/zenodo.3996575


README.md

Preventing pressure oscillations does not fix local linear stability issues of entropy-based split-form high-order schemes

License MIT DOI 10.5281/zenodo.4054366


README.md

Julia for adaptive high-order simulations

License MIT launch binder

What about *composability*?

- Multiple dispatch: “function overloading on runtime types”
- No difference between standard library code, package code, own code
- Only implement what you need → rapid prototyping
- Increased code reuse invites collaboration

Adapted from Trixi

```
calc_volint(solver::DGSEM, volint_type::WeakForm, equations)
calc_volint(solver::DGSEM, volint_type::StrongForm, equations)
calc_volint(solver::DGSEM, volint_type::StrongForm, equations::MHD)
```

What else is there?

- Julia community is focused on data science, not simulation science
- Different notion of "HPC" than computional science community
  - $\rightarrow$ think big data, not necessarily exascale computing

Conclusions and outlook

Conclusions and outlook: hyperbolic gravity

- Compute solution to elliptic problem via hyperbolic framework
  - Verified experimental order of convergence
  - High-order gradient computations
- Multi-physics coupling for flow-gravity simulations works
  - Reuse hyperbolic schemes without modifications
  - Supports adaptive mesh refinement
- Next up: speed up gravity solver, add more physics

Conclusions and outlook: Julia for scientific computing

- Performance is good, but predictability not so much
  - "no free lunch" for C/C++/Fortran-like performance
  - Requires new performance intuition
  - No verdict on large-scale MPI parallelization yet
- Ad-hoc compilation and multiple dispatch can be great assets
  - Facilitates rapid prototyping
  - More code sharing and code reuse
- Ease of use
  - Minimal setup time for new users
  - Invites collaboration
  - Great for reproducibility of scientific findings
- Next up: fully parallelize Trixi and scale to 10,000+ cores

Thank you for your interest!

Are there any questions?