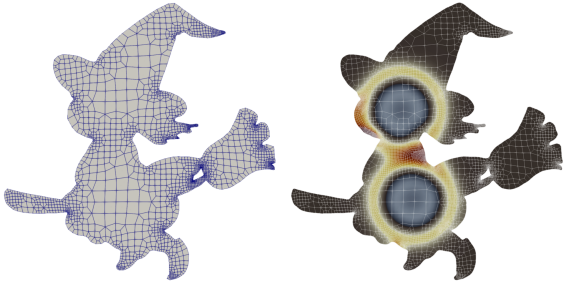
From Mesh Generation to Adaptive Simulation: A Journey in Julia

Andrew R. Winters

JuliaCon 2022, 27$^{\text{th}}$ - 29$^{\text{th}}$ July 2022

Linköping University, Sweden



Acknowledgements

HOHQMesh developer

- David Kopriva

Trixi.jl: Main developers

- Jesse Chan
- Gregor Gassner
- Hendrik Ranocha
- Michael Schlottke-Lakemper

Trixi.jl: Mesh adaptivity and shock capturing

- Erik Faulhaber
- Christof Czernik
- Andrés Rueda-Ramírez

Common workflow in computational fluid dynamics

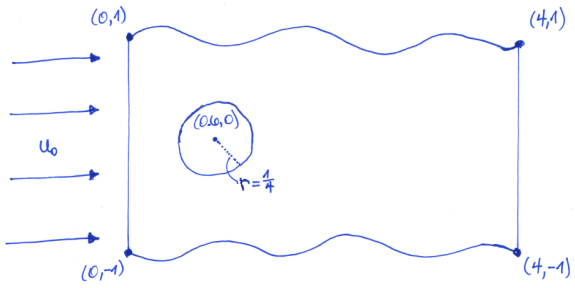
1. Model physical phenomena with partial differential equations (PDEs).
2. Divide the physical domain into a mesh.
3. Approximate spatial and temporal derivatives.
4. Simulate the flow.
5. Visualize simulation results.

`This talk demonstrates a Julia toolchain for this workflow.`

- Reproducibility repository for this talk:

  `https://github.com/trixi-framework/talk-2022-juliacon_toolchain`

Example: Supersonic flow over a cylinder

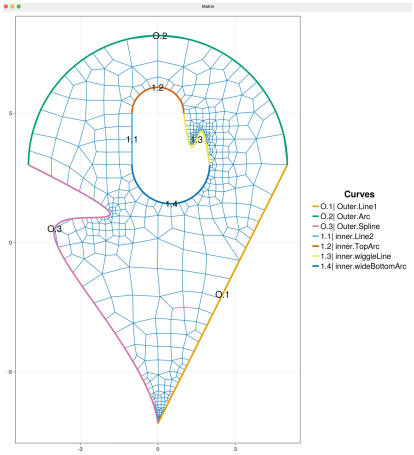


- Compressible Euler equations in 2D
- Nonlinear hyperbolic PDEs
- Complex behavior: shocks and vortices
- Curvilinear domain
- Supersonic inflow/outflow on left/right
- Walls along sinusoidal top/bottom and the cylinder

Where do meshes come from?

HOHQMesh.jl

- Julia frontend to Fortran-based High Order Hex-Quad Mesher (HOHQMesh)
- Creates quad (2D) or hex (3D) meshes with high-order boundary information
- Interactive functionality to create and visualize 2D boundary model prior to meshing
- Uses Makie to plot interactive model



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

Installing HOHQMesh

- Package manager Pkg handles everything!
- Install HOHQMesh + interactive tools

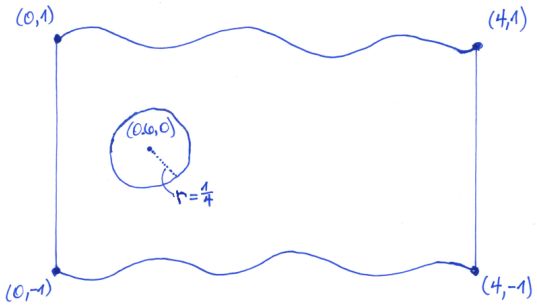
```
julia> import Pkg
julia> Pkg.add("HOHQMesh")
```

- Visualization of interactive tools requires Makie, e.g., the Makie backend

```
julia> Pkg.add("GLMakie")
```

- Ready to interactively generate unstructured quadrilateral meshes!

Example: Supersonic flow over a cylinder



Key features of the curved domain:

- Two straight sided boundaries
- Two sinusoidal boundaries
- Circular inner boundary
- Mesh refinement for wake behind cylinder

Using interactive HOHQMesh

- Load HOHQMesh and visualization capability

```
julia> using HOHQMesh, GLMakie
```

- Build a HOHQMesh model `cylinder_flow` for this domain

```
julia> cylinder_flow = newProject("cylinder_sine_walls", "out")
```

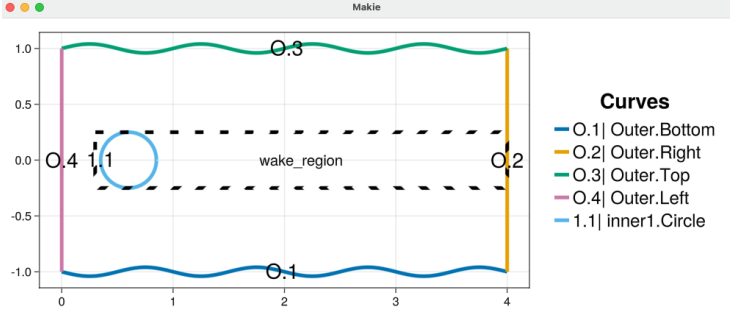
- Create and add inner circular boundary into `cylinder_flow`

```
julia> cylinder = newCircularArcCurve("Circle", [0.6, 0.0, 0.0],  0.25,
                                      0.0, 360.0,  "degrees")
julia> addCurveToInnerBoundary!(cylinder_flow, cylinder, "inner1")
```

Using interactive HOHQMesh

- Add outer boundary curves as straight lines or with parametric equations
- Manually add refinement in wake region behind the cylinder

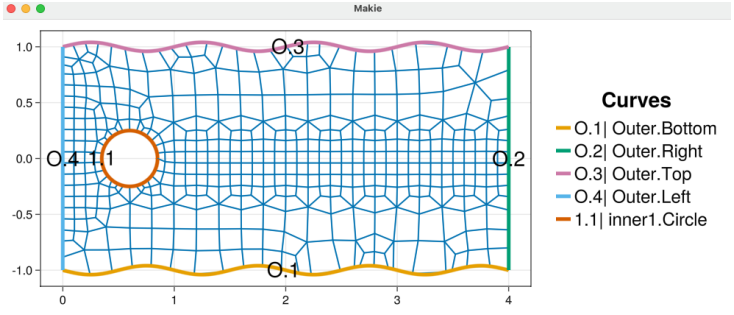
```
julia> refine_line = newRefinementLine("wake_region", "smooth",
                                       [0.3,0.0,0.0], [4.0,0.0,0.0], 0.1, 0.25)
julia> addRefinementRegion!(cylinder_flow, refine_line)
```



Result of interactive HOHQMesh

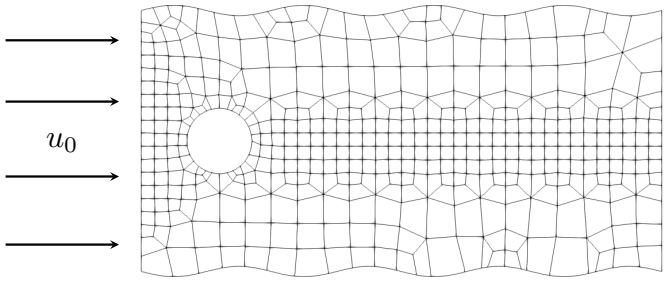
- Lay a `BackgroundGrid` and we are ready to generate a quad mesh

```
julia> addBackgroundGrid!(cylinder_flow, [0.4, 0.4, 0.0])
julia> generate_mesh(cylinder_flow)
```



- `generate_mesh` call saves mesh file to the directory `"out"`

Example: Supersonic flow over a cylinder



Numerical approximation of non-linear physical model needs:

- High-order accuracy on unstructured, curved meshes
- Shock capturing to handle complex transient behavior
- Solution-adaptive refinement of the mesh

How to approximate the solution?

Trixi.jl

- Adaptive numerical simulation framework for hyperbolic PDEs
- Robust, high-order accurate discretizations in space and time
- Supports 1D, 2D, or 3D setups
- Integrated into the Julia ecosystem:
  - `OrdinaryDiffEq.jl`: time integration
  - `P4est.jl`: adaptive mesh management
  - `Plots.jl`, `Makie.jl`: plotting
  - `LoopVectorization.jl`: performance

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

Installing and using Trixi

- Install Trixi + dependencies

```
julia> Pkg.add(["Trixi", "OrdinaryDiffEq"])
julia> using OridinaryDiffEq
julia> using Trixi
```

- Ready to run adaptive simulations, e.g., on unstructured quad meshes!
- But how?!

- Simulation setup done with an `elixir` file
- Combines components from the Trixi library to create and solve a discretization

Mixing an elixir: Physical setup

- Setup 2D compressible Euler equations with adiabatic constant $\gamma = 1.4$

```
equations = CompressibleEulerEquations2D(1.4)
```

- Trixi is easily extendable! Create an initial condition function for Mach 2 flow

```
function initial_condition_mach2_flow(x,t,equations::CompressibleEulerEquations2D)
  # set the freestream flow parameters
  rho = 1.4
  v1  = 2.0
  v2  = 0.0
  p   = 1.0
  return prim2cons(SVector(rho, v1, v2, p), equations)
end
```

- Also create functions for `boundary_condition_supersonic_inflow` and `boundary_condition_supersonic_outflow`

Mixing an elixir: Unstructured mesh

- Read in the quadrilateral mesh file created with HOHQMesh

```
mesh_file = joinpath(@__DIR__, "out", "cylinder_sine_walls.inp")
mesh = P4estMesh{2}(mesh_file)
```

- `P4estMesh` type allows adaptive refinement
- Create a `Dictionary` to assign the boundary conditions
- Boundary names match those given from the HOHQMesh model

```
boundary_conditions = Dict( :Bottom => boundary_condition_slip_wall,
                            :Circle => boundary_condition_slip_wall,
                            :Top    => boundary_condition_slip_wall,
                            :Right  => boundary_condition_supersonic_outflow,
                            :Left   => boundary_condition_supersonic_inflow  )
```

Mixing an elixir: Spatial dicretizations

- Shock indicator factor α tracks shocks as the solution evolves
- Blend low-order and high-order methods with a special discretization type

```
polydeg = 4; basis = LobattoLegendreBasis(polydeg)
shock_indicator = IndicatorHennemannGassner(equations, basis,
                                            alpha_max=0.5, alpha_min=0.001,
                                            alpha_smooth=true,
                                            variable=density_pressure)
solver = DGSEM(polydeg, surface_flux=surface_flux,
               volume_integral=VolumeIntegralShockCapturingHG(shock_indicator)
```

- Assemble semidiscretization and setup an ODE problem
- Unlocks the horsepower of OrdinaryDiffEq

```
semi = SemidiscretizationHyperbolic(mesh, equations, initial_condition, solver,
                                    boundary_conditions)
tspan = (0.0, 2.25)
ode = semidiscretize(semi, tspan)
```

Mixing an elixir: Callbacks

- Many useful features are implemented as callbacks, e.g.,
  - Explicit time step computation (CFL-based time stepping)
  - Saving solution information to a file
  - Adaptive mesh refinement (AMR)
  - Positivity preservation
- Callbacks are passed to the ODE solver to perform these tasks during execution

```
amr_indicator = IndicatorLöhner(semi, variable=Trixi.density)

amr_controller = ControllerThreeLevel(semi, amr_indicator,
                                      base_level=0,
                                      med_level=3, med_threshold=0.05,
                                      max_level=5, max_threshold=0.1)

amr_callback = AMRCallback(semi, amr_controller,
                           interval=1,
                           adapt_initial_condition=true,
                           adapt_initial_condition_only_refine=true)
```

Mixing an elixir: Time integration & running a simulation

- Integrate in time using SSPRK with error-based time stepping
- Positivity preservation done in the `stage_limiter!`
- Pass `callbacks` set with AMR and saving the solution to a file

```
sol = solve(ode, SSPRK43(stage_limiter!),
            save_everystep=false, callback=callbacks);
```

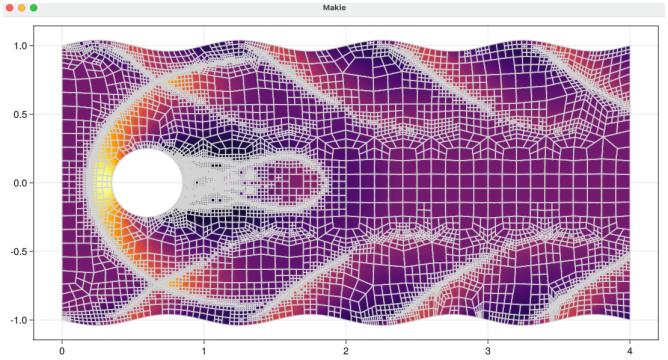
- Collect everything in an elixir file
- Run from the REPL with the `trixi_include` command

```
julia> elixir_file = joinpath(@__DIR__, "elixir_euler_mach2_cylinder.jl")
julia> trixi_include(elixir_file, tspan=(0.0, 0.5))
```

Immediate visualization with Makie

- Plots or Makie can visualize Trixi solution data directly in the REPL

```
julia> pd = PlotData2D(sol);
julia> plot(pd["rho"])
```



- Looks good but we are missing some information...

Postprocessing tool for Trixi

```
julia> Pkg.add("Trixi2Vtk")
```

- Converts HDF5 output to VTK files usable with visualization software, like ParaView
- Postprocess all solution files created by the `SaveSolution` callback

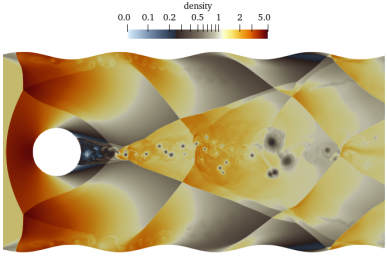
```
julia> using Trixi2Vtk
julia> trixi2vtk("out/solution*", output_directory="plot_files")
```

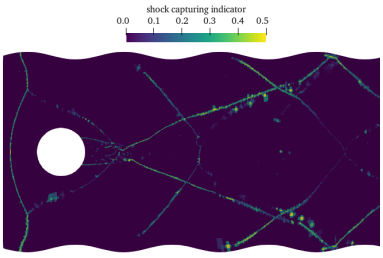
- Re-interpolates solution data for improved visualization quality
- Creates auxiliary files that hold cell-based data, like the shock and AMR indicators

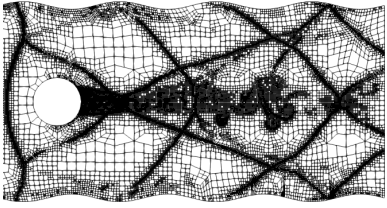
https://github.com/trixi-framework/Trixi2Vtk.jl

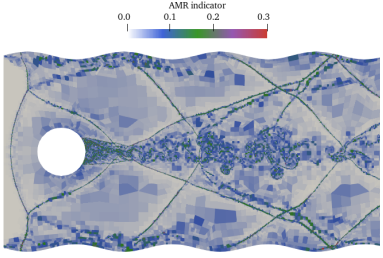
https://github.com/trixi-framework/talk-2022-juliacon_toolchain

Visualization from ParaView





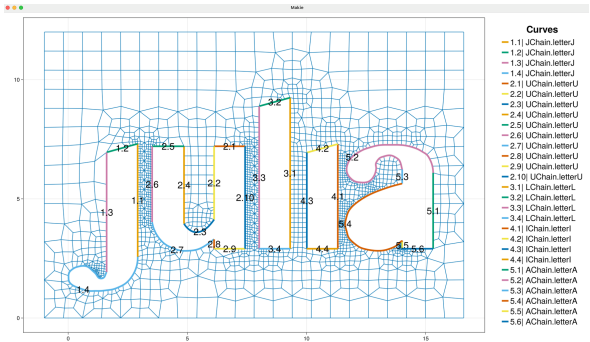




Conclusions and outlook

Presented a Julia toolchain for

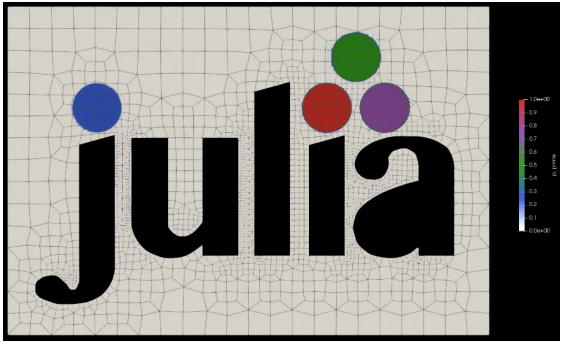
- Unstructured meshes via HOHQMesh
- Adaptive simulations via Trixi
- Visualization via Makie or postprocessing via Trixi2Vtk
- Future: Extend interactive HOHQMesh tools for hex meshes



Conclusions and outlook

Presented a Julia toolchain for

- Unstructured meshes via HOHQMesh
- Adaptive simulations via Trixi
- Visualization via Makie or postprocessing via Trixi2Vtk
- Future: Extend interactive HOHQMesh tools for hex meshes



Thank you for your interest!

Join us on Trixi's Slack (with a #hohqmesh channel)
`https://github.com/trixi-framework/Trixi.jl`

For more results check out Trixi's YouTube channel
`https://www.youtube.com/channel/UCpd92vU2HjjTPup-AIN0pkg/videos`