Technische Universität Braunschweig

IAM
INSTITUT FÜR ANGEWANDTE MECHANIK

How simple should using `Ferrite.jl` get?

Knut Andreas Meyer

Complete FE program for solving the nonlinear, time-dependent problem

$$c(u)\dot{u} - [k\,\nabla u]\cdot\nabla = h, \quad c(u) = c_0 + au$$

```julia
using Ferrite, FESolvers, FerriteProblems, FerriteAssembly, FerriteViz, CairoMakie
import FerriteAssembly.ExampleElements: WeakForm
grid = generate_grid(Quadrilateral, (10,10)); dΩ = union(getfaceset.((grid,), ("left","top"))...)
ip = Lagrange{RefQuadrilateral,1}()
dh = close!(add!(DofHandler(grid), :u, ip))
ch = close!(add!(ConstraintHandler(dh), Dirichlet(:u, dΩ, Returns(0.0))));
cv = CellValues(QuadratureRule{RefQuadrilateral}(2), ip)
material = WeakForm((δu, ∇δu, u, ∇u, u_dot, _) -> δu*((1+10u)*u_dot - 1) + (∇δu ⋅ ∇u))
problem = FerriteProblem(FEDefinition(DomainSpec(dh, material, cv); ch))
solver = QuasiStaticSolver(;nlsolver=NewtonSolver(), timestepper=FixedTimeStepper(;num_steps=10, Δt=0.1))
solve_problem!(problem, solver)
FerriteViz.solutionplot(dh, FESolvers.getunknowns(problem))
```

FerriteCon2023, Bochum, 2023-10-06

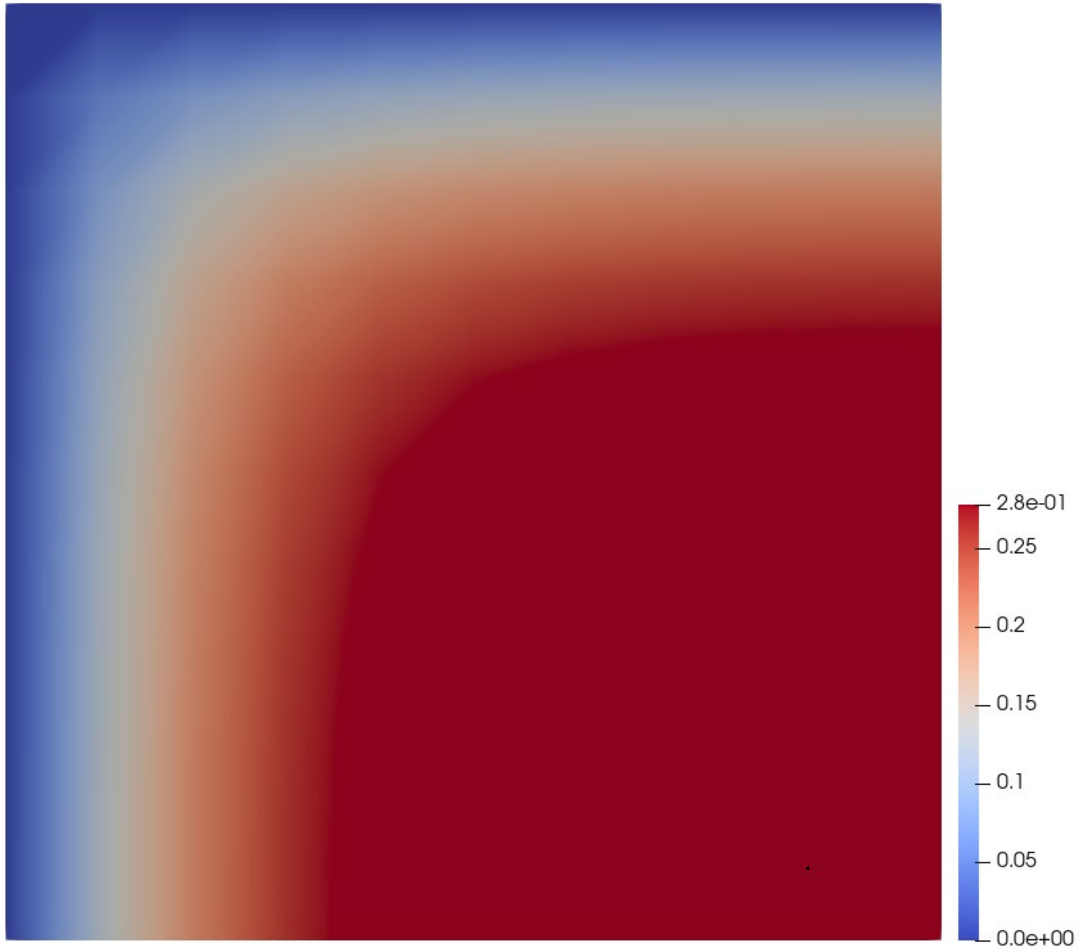
Introduction

FE program for solving the nonlinear, time-dependent problem

$$c(u)\dot{u} - [k\,\nabla u]\cdot\nabla = h, \quad c(u) = c_0 + au$$

```
using Ferrite, FerriteViz

function setup()
    grid = generate_grid(Quadrilateral, (10,10)); dΩ =
union(getfaceset.((grid,), ("left","top"))...)
    ip = Lagrange{RefQuadrilateral,1}()
    dh = close!(add!(DofHandler(grid), :u, ip))
    ch = close!(add!(ConstraintHandler(dh), Dirichlet(:u, dΩ, Returns(0.0))));
    cv = CellValues(QuadratureRule{RefQuadrilateral}(2), ip)
    return dh, ch, cv
end

function element_routine!(Ke, re, ae, ae_old, Δt, cv)
    for q_point in 1:getnquadpoints(cv)
        dΩ = getdetJdV(cv, q_point)
        u = function_value(cv, q_point, ae)
        uold = function_value(cv, q_point, ae_old)
        ∇u = function_gradient(cv, q_point, ae)
        udot = (u-uold)/Δt
        for i in 1:getnbasefunctions(cv)
```

```
using Ferrite, FESolvers, FerriteProblems, FerriteAssembly, FerriteViz, CairoMakie
import FerriteAssembly.ExampleElements: WeakForm
grid = generate_grid(Quadrilateral, (10,10)); dΩ = union(getfaceset.((grid,), ("left","top"))...)
ip = Lagrange{RefQuadrilateral,1}()
dh = close!(add!(DofHandler(grid), :u, ip))
ch = close!(add!(ConstraintHandler(dh), Dirichlet(:u, dΩ, Returns(0.0))));
cv = CellValues(QuadratureRule{RefQuadrilateral}(2), ip)
material = WeakForm((δu, ∇δu, u, ∇u, u_dot, args...) -> δu*((1+10u)*u_dot - 1) + 1.0*(∇δu ⋅ ∇u))
problem = FerriteProblem(FEDefinition(DomainSpec(dh, material, cv); ch))
solver = QuasiStaticSolver(;nlsolver=NewtonSolver(), timestepper=FixedTimeStepper(;num_steps=10, Δt=0.1))
solve_problem!(problem, solver)
FerriteViz.solutionplot(dh, FESolvers.getunknowns(problem))
```



Introduction

Complete FE program for solving the nonlinear, time-dependent weak form

$$c(u)\dot{u} - [k \, \nabla u] \cdot \nabla = h, \quad c(u) = c_0 + au$$

```
# Ferrite.jl syntax
grid = generate_grid(Quadrilateral, (10,10)); dΩ = union(getfaceset.((grid,), ("left","top"))...)
ip = Lagrange{RefQuadrilateral,1}()
dh = close!(add!(DofHandler(grid), :u, ip))
ch = close!(add!(ConstraintHandler(dh), Dirichlet(:u, dΩ, Returns(0.0))));
cv = CellValues(QuadratureRule{RefQuadrilateral}(2), ip)

# FerriteAssembly.jl
material = WeakForm((δu, ∇δu, u, ∇u, u_dot, args...) -> δu*((1+10u)*u_dot - 1) + 1.0*(∇δu ⋅ ∇u))
domainspec = DomainSpec(dh, material, cv)

# FerriteProblems.jl
problem = FerriteProblem(FEDefinition(domainspec; ch))

# FESolvers.jl
solver = QuasiStaticSolver(;nlsolver=NewtonSolver(), timestepper=FixedTimeStepper(;num_steps=10, Δt=0.1))
solve_problem!(problem, solver)

# FerriteViz.jl
FerriteViz.solutionplot(dh, FESolvers.getunknowns(problem))
```

Outline

- **Introduction to the "Ecosystem" built on top of Ferrite.jl**
  - FerriteAssembly.jl
  - FESolvers.jl
  - FerriteProblems.jl
- **Examples from research and fun**
  - Multi-field problems: Frost damage in concrete
  - Highly nonlinear problems: Partially saturated porous media
  - Phase-field damage with @lijas's IGA.jl
- **Conclusions and outlook**
  - Challenges in the design
  - When to use?

FerriteAssembly.jl

```
work!(worker,        domainbuffer; a, aold)
```

Current and old degrees of freedom

What
- Assembler
- Integrator

Where
- Cell domain
- Face domain
- Interface domain

User to define:

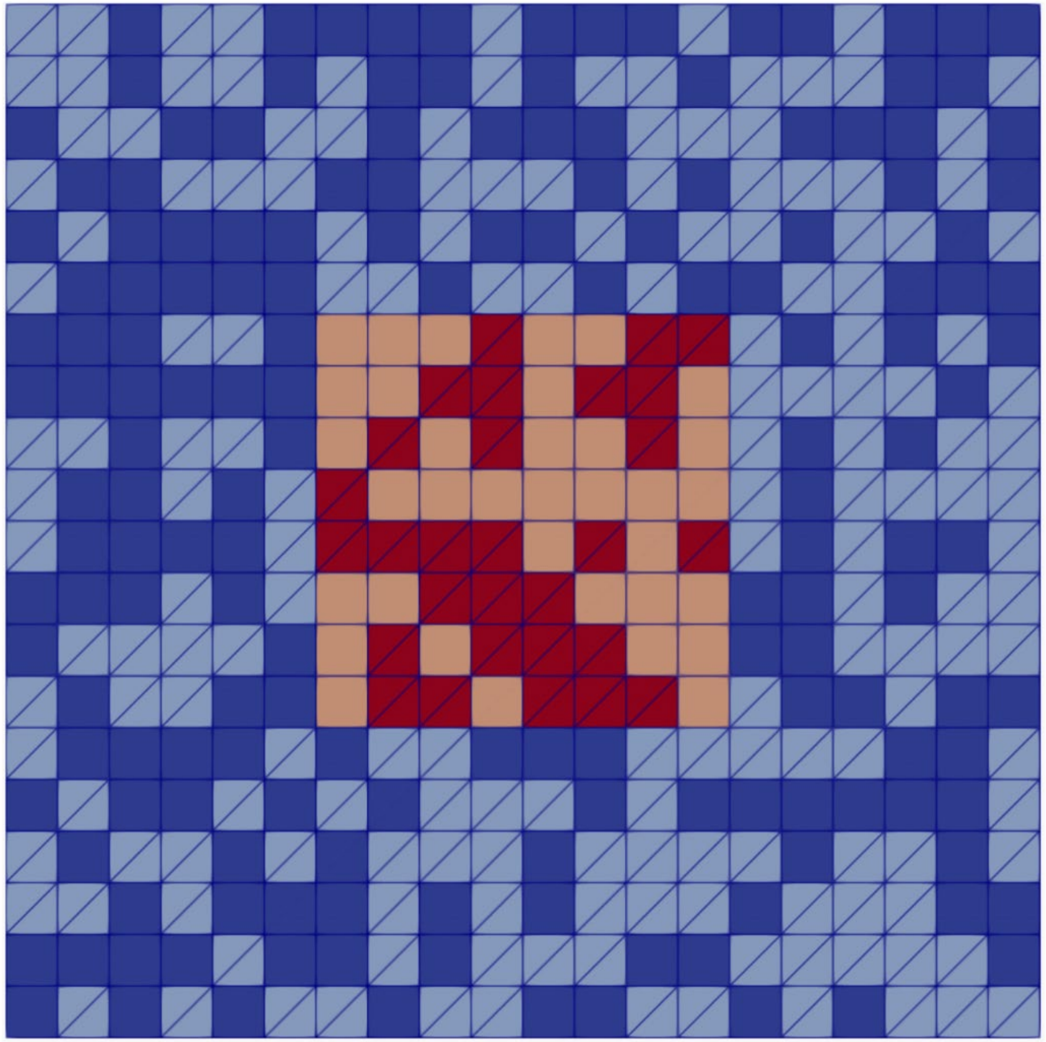
```
# Assemble: Cell domain
element_routine!(Ke, re, cell_state, ae, material, cellvalues, cellbuffer)
element_residual!(re, cell_state, ae, material, cellvalues, cellbuffer)

# Assemble: Face domain
face_routine!(Ke, re, ae, material, facevalues, facebuffer)
face_residual!(re, ae, material, facevalues, facebuffer)

# Integrate: Cell and face domain
integrate_cell!(val, cell_state, ae, material, cv, cellbuffer)
integrate_face!(val, ae, material, cv, facebuffer)
```

FerriteAssembly.jl

What constitutes a domain?

- Same SubDofHandler
  - Same fields
  - Same interpolations
  - Same cell type
- Same FEValues
  - Same quadrature rule
- Same “material” type (i.e. same weak form)

● Inclusion (tet)	`sdh_tet,`	`inclusion_material`
● Inclusion (quad)	`sdh_quad,`	`inclusion_material`
● Matrix (tet)	`sdh_tet,`	`matrix_material`
● Matrix (quad)	`sdh_quad,`	`matrix_material`



FerriteAssembly.jl

How is a domain defined?

```julia
# Create single DomainSpec
ds = DomainSpec(sdh::[Sub]DofHandler, material, fe_values; [set], [user_data],
                kwargs...)

# Create single domain buffer
domainbuffer = setup_domainbuffer(ds; kwargs...)
```

```julia
face_residual!(re, ae, material,
    facevalues, facebuffer)
```

```julia
struct DomainBuffer{I,B,S,SDH<:SubDofHandler}
    set::Vector{I}
    itembuffer::B
    states::Dict{Int,S}   # Indexed by cellid
    old_states::Dict{Int,S}
    sdh::SDH
end
```

```julia
element_routine!(Ke, re, cell_state, ae, material, cellvalues, cellbuffer)
```

FerriteAssembly.jl

How to define buffers for multiple domains?

```
# Create single DomainSpec
ds = DomainSpec(sdh::[Sub]DofHandler, material, fe_values; [set], [user_data],
                kwargs...)

# Create single domain buffer
domainbuffer = setup_domainbuffer(ds; kwargs...)

# Create multiple domain buffers
domainbuffers = setup_domainbuffers(Dict(
    "domain 1"=>DomainSpec(...),
    "domain 2"=>DomainSpec(...));
    kwargs...)
```

```
DomainBuffer -> Dict{String, <:DomainBuffer}
```

FerriteAssembly.jl

How to make things multithreaded?

```
# Create single DomainSpec
ds = DomainSpec(sdh::[Sub]DofHandler, material, fe_values; [set], [user_data],
                kwargs...)

# Create single domain buffer
domainbuffer = setup_domainbuffer(ds; kwargs...)

# Create multiple domain buffers
domainbuffers = setup_domainbuffers(Dict(
    "domain 1"=>DomainSpec(...),
    "domain 2"=>DomainSpec(...));
    kwargs...)
```

```
DomainBuffer -> Dict{String, <:DomainBuffer}
```

FerriteAssembly.jl

How to make things multithreaded?

```
# Create single DomainSpec
ds = DomainSpec(sdh::[Sub]DofHandler, material, fe_values; [set], [user_data],
                kwargs...)

# Create single domain buffer
domainbuffer = setup_domainbuffer(ds; threading=true, kwargs...)

# Create multiple domain buffers
domainbuffers = setup_domainbuffers(Dict(
    "domain 1"=>DomainSpec(...),
    "domain 2"=>DomainSpec(...));
    threading=true, kwargs...)
```

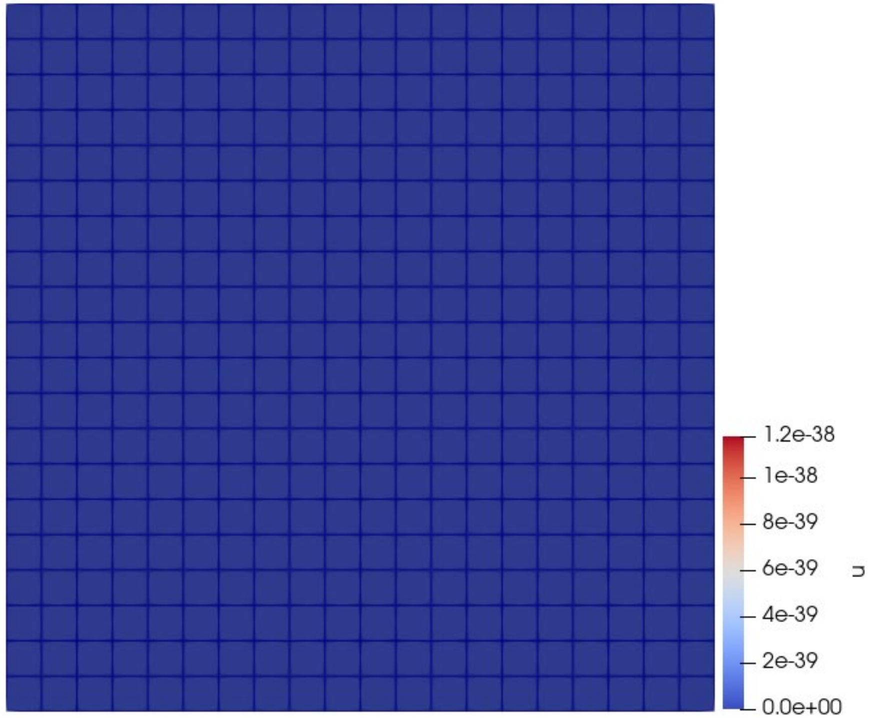
```
DomainBuffer -> ThreadedDomainBuffer
```

```
Dict{String, <:DomainBuffer} -> Dict{String, <:ThreadedDomainBuffer}
```

FerriteAssembly.jl

Let's do some work!

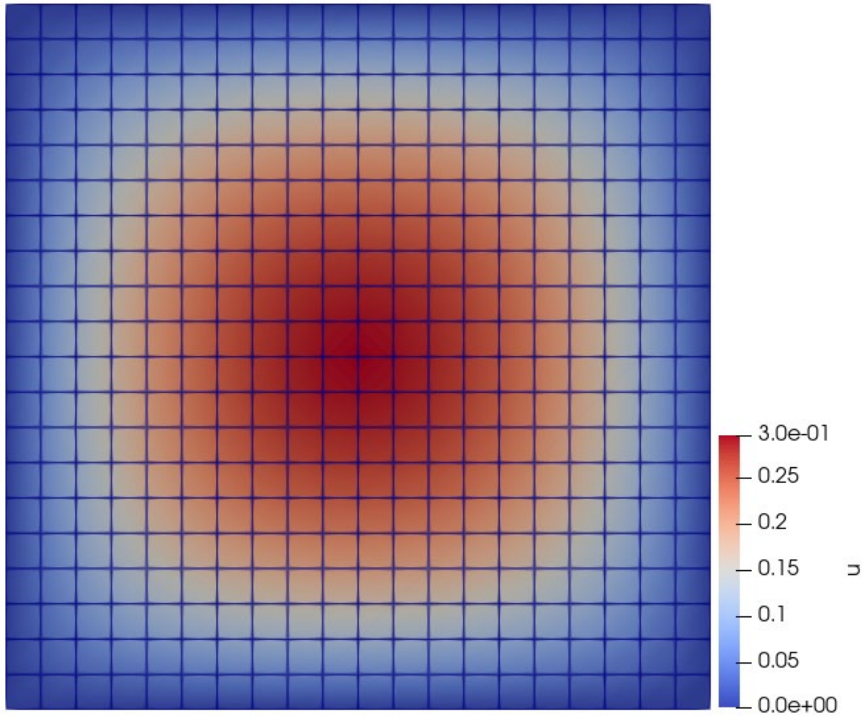
```julia
using Ferrite, FerriteAssembly
import FerriteAssembly.ExampleElements: StationaryFourier
# Ferrite.jl setup
grid = generate_grid(Quadrilateral, (20, 20));
dΩ = union(getfaceset.((grid,), ("left", "top", "bottom", "right"))...)
ip = Lagrange{RefQuadrilateral,1}()
dh = close!(add!(DofHandler(grid), :u, ip))
ch = close!(add!(ConstraintHandler(dh), Dirichlet(:u, dΩ, Returns(0.0))));
cv = CellValues(QuadratureRule{RefQuadrilateral}(2), ip)
K = create_sparsity_pattern(dh)
a, r = [zeros(ndofs(dh)) for _ in 1:2]

# FerriteAssembly.jl
material = StationaryFourier(#=k=#1.0)
domainbuffer = setup_domainbuffer(DomainSpec(dh, material, cv))
assembler = start_assemble(K, r)
work!(assembler, domainbuffer; a=a)

apply!(K, r, ch)
a .-= K\r
```



FerriteAssembly.jl

Let's do some work!

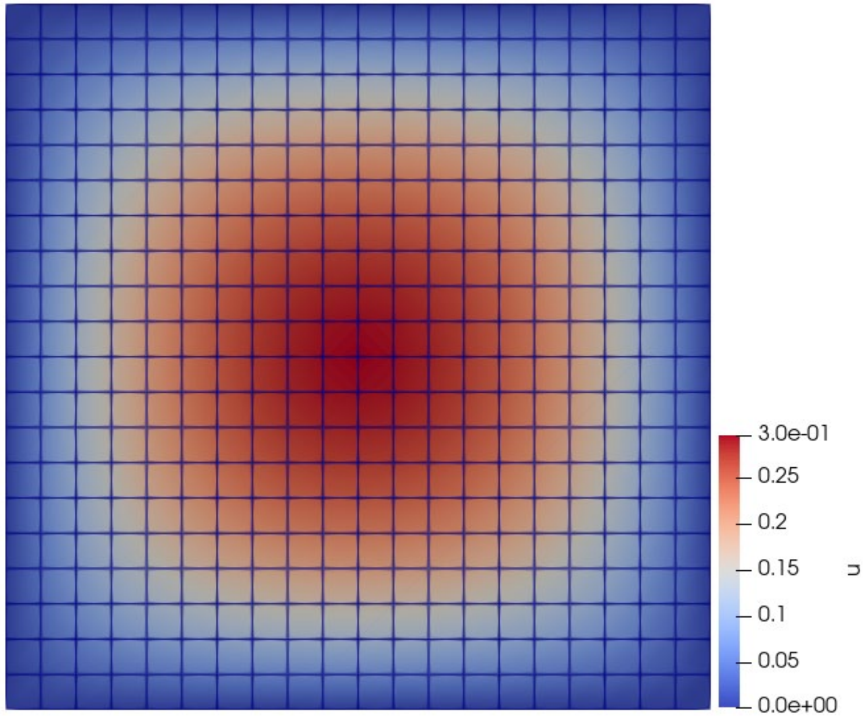
```
using Ferrite, FerriteAssembly
import FerriteAssembly.ExampleElements: StationaryFourier
# Ferrite.jl setup
grid = generate_grid(Quadrilateral, (20, 20));
dΩ = union(getfaceset.((grid,), ("left", "top", "bottom", "right"))...)
ip = Lagrange{RefQuadrilateral,1}()
dh = close!(add!(DofHandler(grid), :u, ip))
ch = close!(add!(ConstraintHandler(dh), Dirichlet(:u, dΩ, Returns(0.0))));
cv = CellValues(QuadratureRule{RefQuadrilateral}(2), ip)
K = create_sparsity_pattern(dh)
a, r = [zeros(ndofs(dh)) for _ in 1:2]

# FerriteAssembly.jl
material = StationaryFourier(#=k=#1.0)
domainbuffer = setup_domainbuffer(DomainSpec(dh, material, cv))
assembler = start_assemble(K, r)
work!(assembler, domainbuffer; a=a)
lh = LoadHandler(dh)
add!(lh, BodyLoad(:u, #=qr_order=# 2, Returns(-1.0)))
apply!(r, lh, 0.0)
apply!(K, r, ch)
a .-= K\r
```



FerriteAssembly.jl

Not only for assembling

```julia
using Ferrite, FerriteAssembly
import FerriteAssembly.ExampleElements: StationaryFourier
# Ferrite.jl setup
grid = generate_grid(Quadrilateral, (20, 20));
dΩ = union(getfaceset.((grid,), ("left", "top", "bottom", "right"))...)
ip = Lagrange{RefQuadrilateral,1}()
dh = close!(add!(DofHandler(grid), :u, ip))
ch = close!(add!(ConstraintHandler(dh), Dirichlet(:u, dΩ, Returns(0.0))));
cv = CellValues(QuadratureRule{RefQuadrilateral}(2), ip)
K = create_sparsity_pattern(dh)
a, r = [zeros(ndofs(dh)) for _ in 1:2]

# FerriteAssembly.jl
material = StationaryFourier(#=k=#1.0)
domainbuffer = setup_domainbuffer(DomainSpec(dh, material, cv))
assembler = start_assemble(K, r)
work!(assembler, domainbuffer; a=a)
lh = LoadHandler(dh)
add!(lh, BodyLoad(:u, #=qr_order=# 2, Returns(-1.0)))
apply!(r, lh, 0.0)
apply!(K, r, ch)
a .-= K\r
```




```julia
integrator = SimpleIntegrator((u, ∇u, s)->(1, u),
                              #=initial_value=#(0.0, 0.0))
work!(integrator, domainbuffer; a=a)
area = integrator.val[1]
average_temperature = integrator.val[2]/area
@show average_temperature
```

```
average_temperature = 0.14005406375299906
```

$$\text{Average temperature, } \bar{T} = \frac{\int_\Omega T \; \mathrm{d}\Omega}{\int_\Omega 1 \; \mathrm{d}\Omega}$$

FerriteAssembly.jl

Fast automatic differentiation

```
# Assemble: Cell domain
element_residual!(re, state, ae, material, cellvalues, cellbuffer)
```

```
Assemble 100x100 heat equation

Standard AD:           11.371 ms (60000 allocations: 9.16 MiB)
Analytical tangent:     4.609 ms (    0 allocations: 0 bytes)
Special buffer for AD: 4.721 ms (    0 allocations: 0 bytes)
```

```
# Create single domain buffer
domainbuffer = setup_domainbuffer(ds; autodiffbuffer=true,
kwargs...)
```

FerriteAssembly.jl

Search docs (Ctrl + /)

Home

- Documentation structure

Learning by doing

- Tutorials
  - Heat Equation
  - Viscoelasticity with state variables
  - Multiple fields
  - Multiple materials
- How-to
  - Threaded assembly
  - Automatic differentiation
  - Local constraint application
  - Robin boundary conditions
  - Volume integration
  - Surface integration

Reference

```
# Summary of FerriteAssembly.jl

1. Define your "material" type

2. Define your low-level routine
   element_routine!, face_residual!, integrate_cell!, etc.

3. Standard Ferrite.jl setup, dh, ch, fe_values, etc.

4. db = setup_domainbuffer(DomainSpec(...); kwargs...)

5. worker = start_assemble(...) # (for example)

6. work!(worker, db; kwargs....)
```

```
assembler = KeReAssembler(K, r; ch, apply_zero=true)
```

FESolvers.jl

Your problem – your way

Define your problem type

```
struct MyProblem{...}
    ...
end
problem = MyProblem(...)
```

Overload a set of functions from FESolvers, e.g.

```
FESolvers.update_problem!(problem, Δx, update_spec)
FESolvers.getjacobian(problem)
FESolvers.postprocess!(problem, step, solver)
```

Define time stepper and nonlinear solver

```
solver = QuasiStaticSolver(;
    nlsolver=NewtonSolver(),
    timestepper=FixedTimeStepper(;num_steps=10, Δt=0.1))
```

Solve the problem

```
solve_problem!(problem, solver)
```

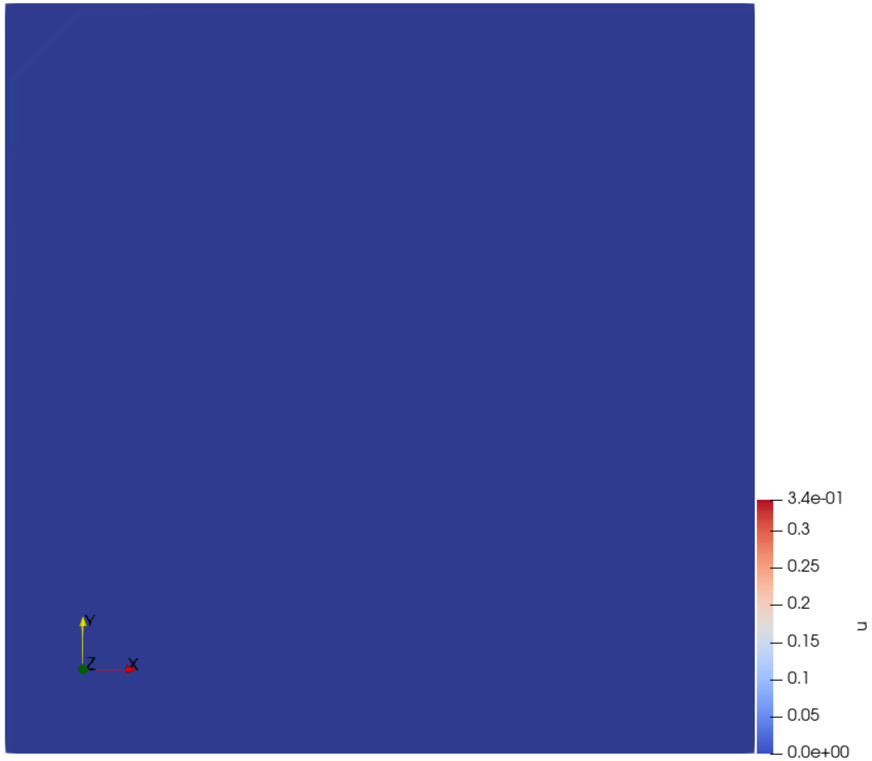
FerriteProblems.jl

A Ferrite.jl problem for FESolvers.jl
- building on FerriteAssembly.jl

```
# Ferrite.jl syntax
grid = generate_grid(Quadrilateral, (10,10)); dΩ = union(getfaceset.((grid,), ("left","top"))...)
ip = Lagrange{RefQuadrilateral,1}()
dh = close!(add!(DofHandler(grid), :u, ip))
ch = close!(add!(ConstraintHandler(dh), Dirichlet(:u, dΩ, Returns(0.0))));
cv = CellValues(QuadratureRule{RefQuadrilateral}(2), ip)

# FerriteAssembly.jl
material = WeakForm((δu, ∇δu, u, ∇u, u_dot, args...) -> δu*((1+10u)*u_dot - 1) + 1.0*(∇δu ⋅ ∇u))
domainspec = DomainSpec(dh, material, cv)

# FerriteProblems.jl
problem = FerriteProblem(FEDefinition(domainspec; ch))

# FESolvers.jl
solver = QuasiStaticSolver(;nlsolver=NewtonSolver(), timestepper=FixedTimeStepper(;num_steps=10, Δt=0.1))
solve_problem!(problem, solver)
```

FerriteProblems.jl

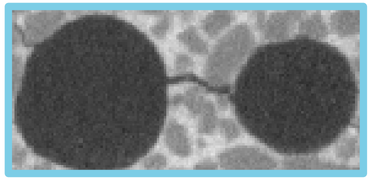
A Ferrite.jl problem for FESolvers.jl

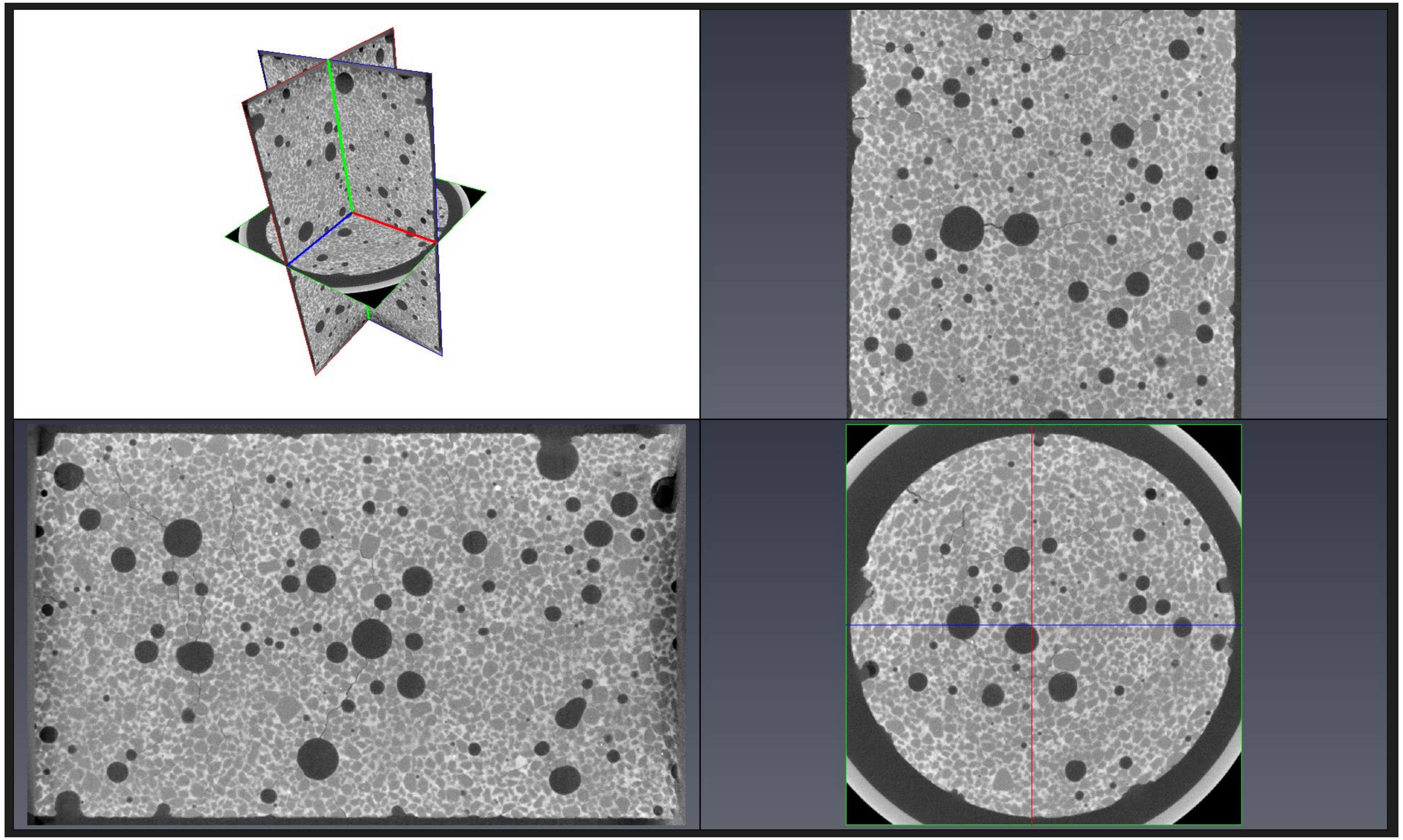



```
@kwdef struct NLHeatPostProc{PVD}
    pvd::PVD=paraview_collection("solution")
end
function FESolvers.postprocess!(post::NLHeatPostProc, problem, st
    dh = FerriteProblems.get_dofhandler(problem)
    vtk_grid(string("solution", step), dh) do vtk
        vtk_point_data(vtk, dh, FESolvers.getunknowns(problem))
        post.pvd[FerriteProblems.get_time(problem)] = vtk
    end
end
function FerriteProblems.close_postprocessing(post::NLHeatPostProc, problem)
    vtk_save(post.pvd)
end

problem = FerriteProblem(FEDefinition(domainspec; ch), NLHeatPostProc())

solver = QuasiStaticSolver(;nlsolver=NewtonSolver(),
                            timestepper=FixedTimeStepper(;num_steps=10, Δt=0.1))
solve_problem!(problem, solver)
```

Examples
From research and fun

Frost damage in concrete



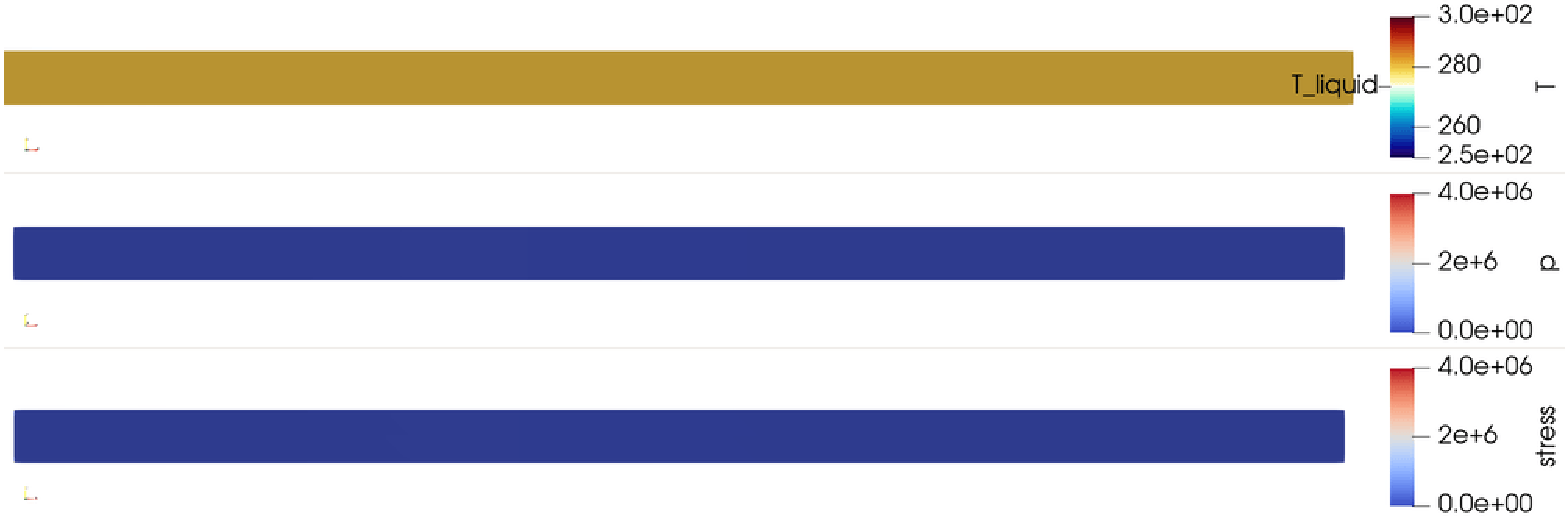


Credit
Aykut Levent
Roland Kruse (IAM)

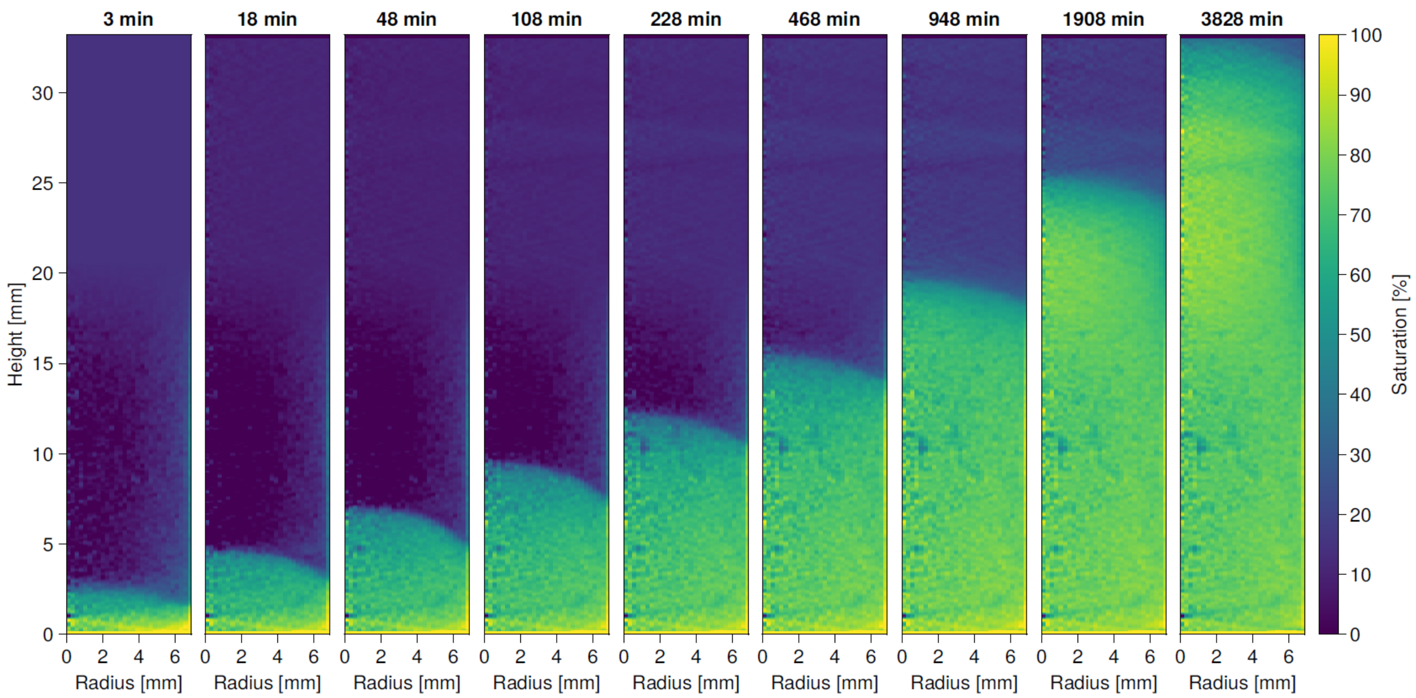
Frost damage in concrete

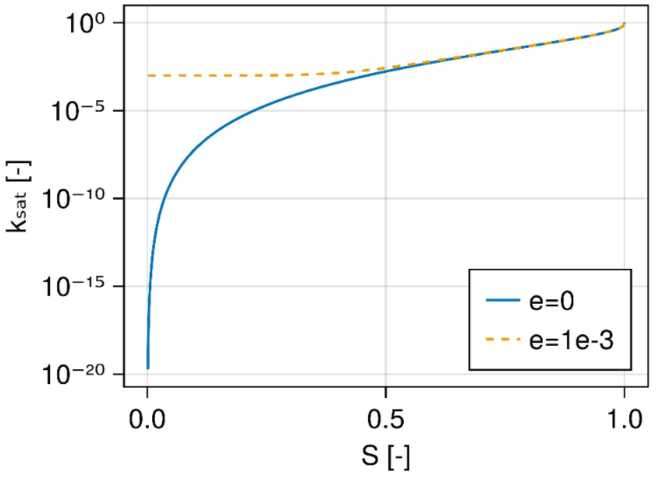
- Mechanical equilibrium (linear elasticity)
- Mass conseration (darcy flow)
- Energy balance (Fick’s law)

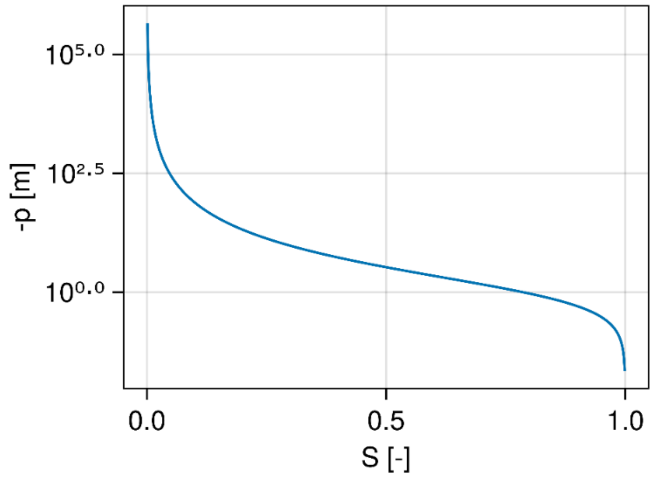
+ Phase transformation: Freezing/thawing
-> Volume expansion



Highly nonlinear problems
Partially saturated porous media

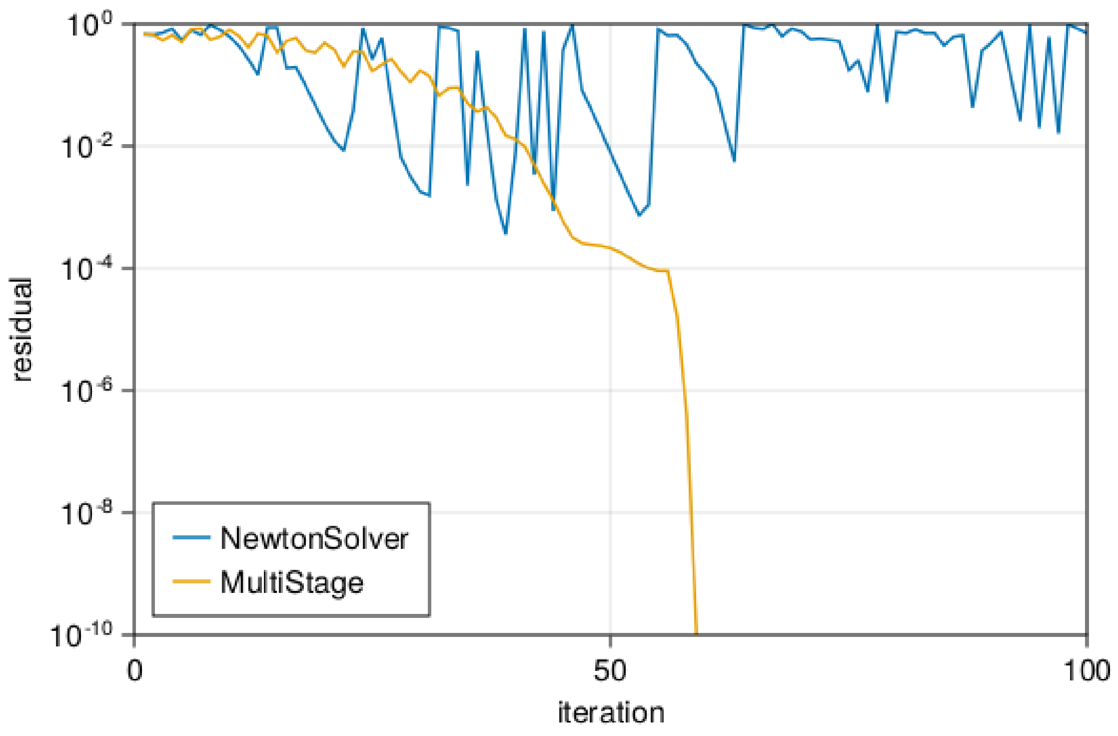






Highly nonlinear problems

```
import PorousMedia: JacobianSpec

# Fast, but requires good guess!
newton_solver = NewtonSolver(;
    linsolver=ITU.TridiagonalSolver(),
    tolerance=1.e-10, maxiter=100,
    update_type=JacobianSpec(:TrueJacobian))

# Slow, but less sensitive to guess!
picard_solver = NewtonSolver(;
    linsolver=ITU.TridiagonalSolver(),
    tolerance=1.e-04, maxiter=100,
    update_type=JacobianSpec(:ModifiedPicard))

# Best of both worlds?
FESolvers.MultiStageSolver([
    picard_solver, newton_solver]
        false)
```



Phase-field fracture with IGA.jl

- For fun
- Simulate brittle fracture
- Phase-field model from Bharali et al. (2023)
- Combine FerriteAssembly/FerriteProblems etc. with @ljias' IGA.jl
- Simulate fracture of "plate with a hole"

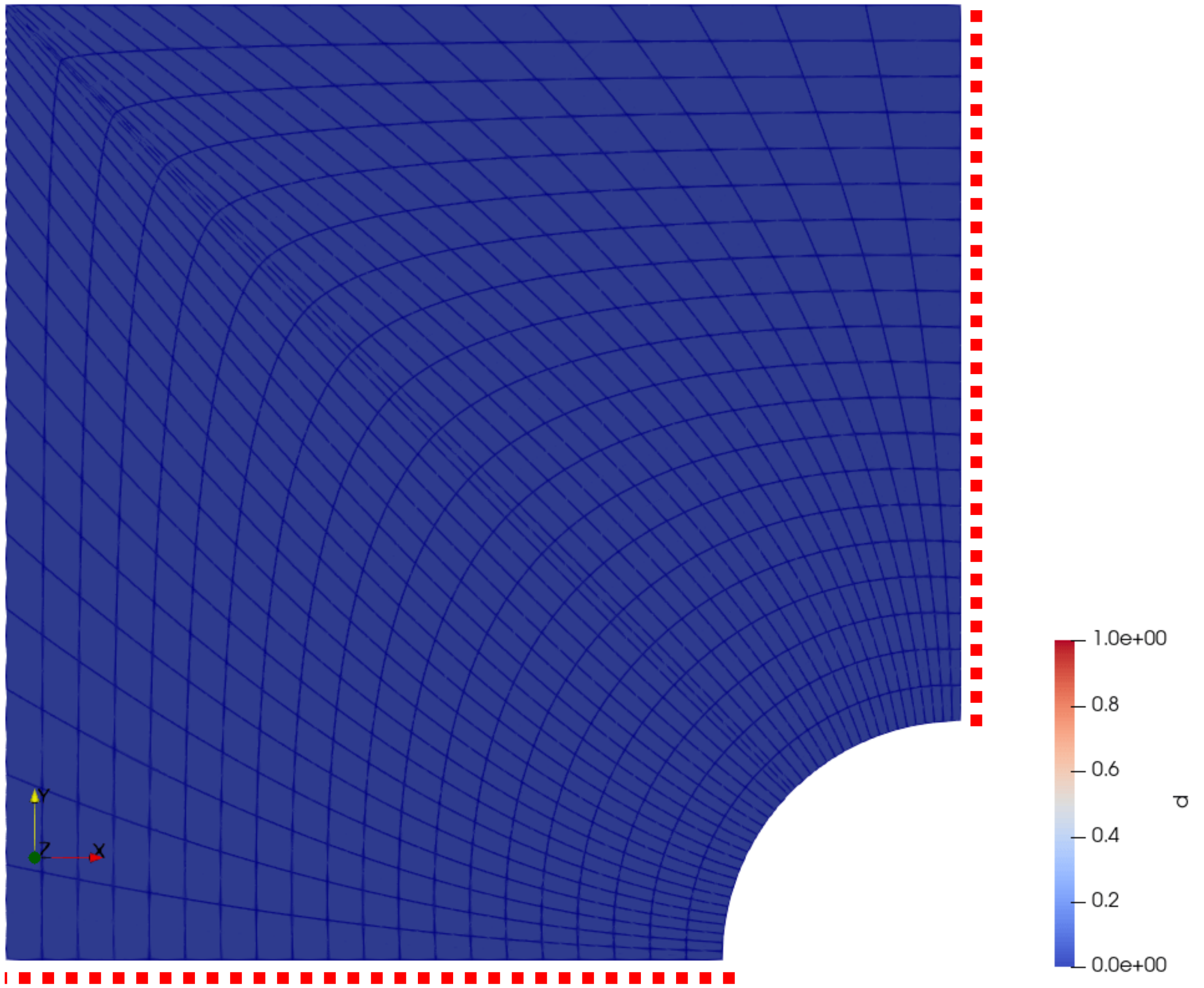
```
# Create the grid using routines in IGA.jl
grid = create_mesh();

# Define the special IGA-interpolation
ip = BernsteinBasis{2,(2,2)}()
qr = QuadratureRule{2,RefCube}(4) # As usual

# Define the special IGA cell values
cv = (
u = BezierCellValues(CellVectorValues(qr, ip)),
d = BezierCellValues(CellScalarValues(qr, ip)))

dh = ... # Create [Mixed]DofHandler as usual
ch = ... # Create ConstraintHandler as usual

# DomainSpec and FEDefinition as usual
domain_spec = DomainSpec(sdh, material, cv)
def = FEDefinition(domain_spec; ch)

# Define problem and solve it as usual
problem = FerriteProblem(def, post)
solve_problem!(problem, solver)
```

R. Bharali, F. Larsson, and R. Jänicke, "A micromorphic phase-field model for brittle and quasi-brittle fracture," *Comput. Mech.*, 2023, doi: 10.1007/s00466-023-02380-1.

Phase-field fracture with IGA.jl

- Simulate brittle fracture
- Phase-field model from Bharali et al. (2023)
- Combine FerriteAssembly/FerriteProblems etc. with @ljias' IGA.jl
- Simulate fracture of "plate with a hole"

Ramp displacement



Symmetry conditions

R. Bharali, F. Larsson, and R. Jänicke, "A micromorphic phase-field model for brittle and quasi-brittle fracture," *Comput. Mech.*, 2023, doi: 10.1007/s00466-023-02380-1.

Challenges and outlook

Challenges

When there is a bug in a user's element routine

```
struct MyMat end

FerriteAssembly.create_cell_state(::MyMat, cv::CellValues, args...) = zeros(getnquadpoints(cv))

function FerriteAssembly.element_residual!(re, state, ae, ::MyMat, cv, buffer)
    old_state = FerriteAssembly.get_old_state(buffer)
    for q_point in 1:getnquadpoints(cv)
        dΩ = getdetJdV(cv, q_point)
        ∇u = function_gradient(cv, q_point, ae)
        e = old_state[q_point] + ∇u⋅∇u # Calculate accumulated "energy"
        for i in 1:getnbasefunctions(cv)
            ∇δu = shape_gradient(cv, q_point, i)
            re[i] += (∇δu ⋅ ∇u) *(1 + e)*dΩ
        end
        state[q_point] = e
    end
end
```

```
state::Vector{Float64}
e::ForwardDiff.Dual

state[q_point] = ForwardDiff.value(e)
```

Challenges

When there is a bug in a user's element routine

1/6 of the error message....

```
   @ C:\Users\meyer\.julia\packages\FerriteAssembly\d4gWT\src\work.jl:41 [inlined]
 [15] #work!#47
   @ C:\Users\meyer\.julia\packages\FerriteAssembly\d4gWT\src\work.jl:16 [inlined]
 [16] work!
   @ C:\Users\meyer\.julia\packages\FerriteAssembly\d4gWT\src\work.jl:15 [inlined]
 [17] update_problem!(p::FerriteProblem{FEDefinition{DofHandler{2, Grid{2, Quadrilateral, Float64}}, ConstraintHandler{DofHandler{2, Grid
LoadHandler{DofHandler{2, Grid{2, Quadrilateral, Float64}}}, FerriteProblems.AbsoluteResidual}, Nothing, FerriteProblems.FEBuffer{Float6
Int64}, Vector{Float64}, FerriteAssembly.DomainBuffer{Int64, FerriteAssembly.CellBuffer{Float64, Vector{Vec{2, Float64}}, CellValues{La
Float64, Vec{2, Float64}, Vec{2, Float64}, Float64, Vec{2, Float64}, QuadratureRule{RefQuadrilateral, Float64, 2}, Lagrange{RefQuadrila
Tuple{UnitRange{Int64}}}, MyMat, Vector{Float64}, Nothing, Nothing}, Vector{Float64}, SubDofHandler{DofHandler{2, Grid{2, Quadrilatera
FerriteProblems.TolScaling{FerriteProblems.AbsoluteResidual, FerriteAssembly.NoScaling, Nothing}}, Nothing}, Δa::Nothing, update_spec
```

```
If the following error is related to converting objects with `ForwardDiff.Dual`s
entries into objects with regular numbers, please consult the docs of `element_residual!`
ERROR: MethodError: no method matching Float64(::ForwardDiff.Dual{ForwardDiff.Tag{FerriteAs
Nothing}, Float64, Vec{2, Float64}, Vec{2, Float64}, Float64, Vec{2, Float64}, QuadratureRu
FerriteAssembly.CellBuffer{Float64, Vector{Vec{2,
Float64}}, CellValues{Lagrange{RefQuadrilateral, 1, Nothing}, Float64, Vec{2, Float64}, Vec
Lagrange{RefQuadrilateral, 1, Nothing}}, NamedTuple{(:u,), Tuple{UnitRange{Int64}}}, MyMat,

Closest candidates are:
  (::Type{T})(::Real, ::RoundingMode) where T<:AbstractFloat
```

```
SparseArrays.SparseMatrixCSC{Float64, Int64}, Vector{Float64}, FerriteAssembly.DomainBuffer{Int64, FerriteAssembly.CellBuffer{Float64, Vec
CellValues{Lagrange{RefQuadrilateral, 1, Nothing}, Float64, Vec{2, Float64}, Vec{2, Float64}, Float64, Vec{2, Float64}, QuadratureRule{Ref
Lagrange{RefQuadrilateral, 1, Nothing}}, NamedTuple{(:u,), Tuple{UnitRange{Int64}}}, MyMat, Vector{Float64}, Nothing, Nothing}, Vector{Flo
Grid{2, Quadrilateral, Float64}}}}, FerriteProblems.TolScaling{FerriteProblems.AbsoluteResidual, FerriteAssembly.NoScaling, Nothing}}, Not
solver::QuasiStaticSolver{NewtonSolver{BackslashSolver, NoLineSearch, Float64}, FixedTimeStepper{Float64}})
   @ FESolvers C:\Users\meyer\.julia\packages\FESolvers\7W19R\src\FESolvers.jl:43
 [21] top-level scope
   @ C:\Users\meyer\Presentations\2023\FerriteCon\Code\Example4\error.jl:35
```

Challenges

When the problem is not converging

Using FerriteAssembly you can

1) Use relative residual tolerance for each field, based on the Lp-norm of nodal “force” contributions

Using FESolvers you can

1) Use the adaptive time stepper
   (Based on @lijas’ algorithm)
2) Use linesearches (by @koehlerson)
3) Use improved initial guess strategies (under development)

---

4) Implement quasi-Newton iterations and even adaptively switching between different methods (under development)
5) Implement your custom nonlinear solver

Before you cross this line, you probably want to step through the code in FESolvers and FerriteProblems

To use or not to use

Experience level ↑

Ferrite.jl developer	Need hackable solutions Deep julia understanding	✓ ✗	*Like my own code best, but...* - *Reduce risk of bugs* - *PRs are welcome ☺*
Have written multiple Ferrite.jl programs	Want custom solutions. Need good performance. Scalable datastructures Not focus on everything	✓	*Main user group* *Reduce risk of bugs* *Fewer design iterations* *Can opt-out by branching*
New PhD students Advanced master students	Learning Ferrite and FE-implementation	✗	*Advanced tasks require good Ferrite.jl and Julia experience*
B.Sc. students Intro for M.Sc. students	Focus on physics Easy convergence	✓	*But why not use a commercial tool?*

Final remarks

FerriteAssembly.jl:	Perform work on domains efficiently	github.com/knutam/FerriteAssembly.jl
FESolvers.jl:	Solve nonlinear [quasi]time-dependent problems	github.com/knutam/FESolvers.jl
FerriteProblems.jl:	Defines a problem to be solved with FESolvers.jl	github.com/knutam/FerriteProblems.jl

- **FerriteAssembly.jl**
  - A package like this can benefit the Ferrite.jl community by
    - Defining a common interface for defining physics – easy to share code
    - Remove a lot of boilerplate for “boring” coding, e.g. postprocessing
  - Call for feedback: Check it out and let me know of any dealbreakers!

- **FESolvers.jl** and **FerriteProblems.jl**
  - Difficult to make it general, hackable, and easy to use
  - Currently: Good for benchmarking and checking your own code
  - Not possible, nor the aim, to compete with commercial codes