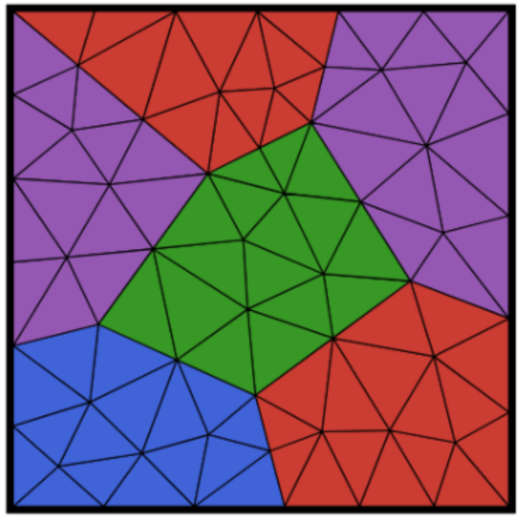
Technische Universität Braunschweig

IAM INSTITUT FÜR ANGEWANDTE MECHANIK

Introduction to the Finite Element Toolbox



Ferrite.jl

Knut Andreas Meyer

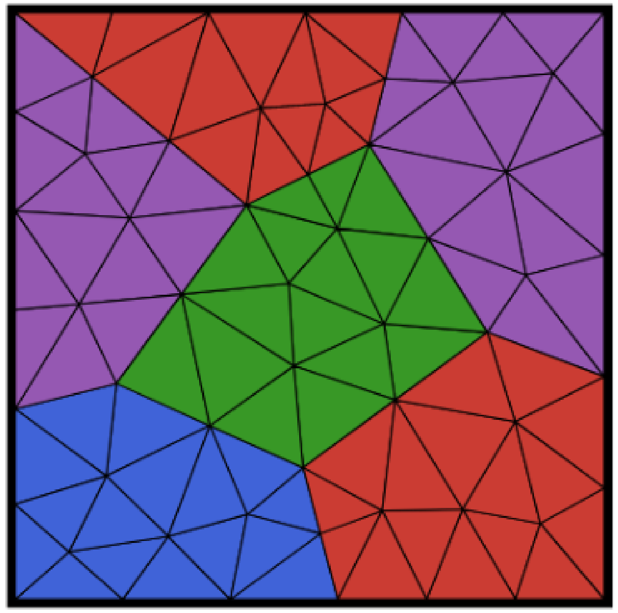
Institute of Applied Mechanics

TU Braunschweig

FerriteCon2024

Chalmers

Outline



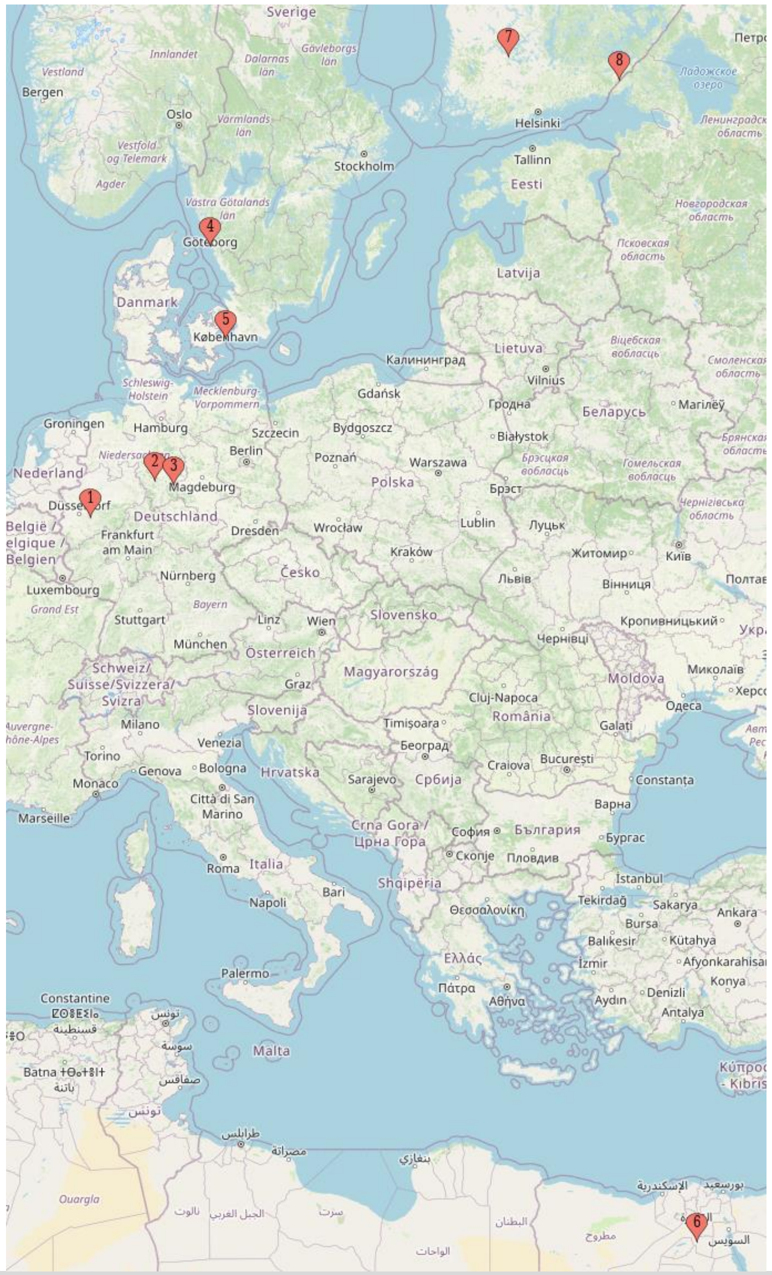
- What is the Julia Programming Language?
- History and users of Ferrite.jl
- What is Ferrite?
  - The FEM Puzzle Pieces
- **How to use Ferrite?**
- Research examples
- Ongoing and future work
- Concluding remarks and questions

What is Julia?

- Open Source (MIT License)
- A young language
  - First public in 2012
  - Release 1.0 in 2018
  - Currently 1.10
- “Easy as python”
  - High-level dynamic programing language
- “Fast as C”
  - Just-in-time compilation

History and users of Ferrite.jl

- **Kristoffer Carlsson** & **Fredrik Ekre**, Chalmers (2016)
- A **toolbox** providing FE building blocks: *Inspired by Deal.ii*
- **Adoption**
    - 31 different contributors
    - 339 github stars / 122 members in Slack
- **FerriteCon**
    - 2022: Braunschweig, Germany
    - 2023: Bochum, Germany
    - 2024: Gothenburg, Sweden
- Version 1.0 will be released soon



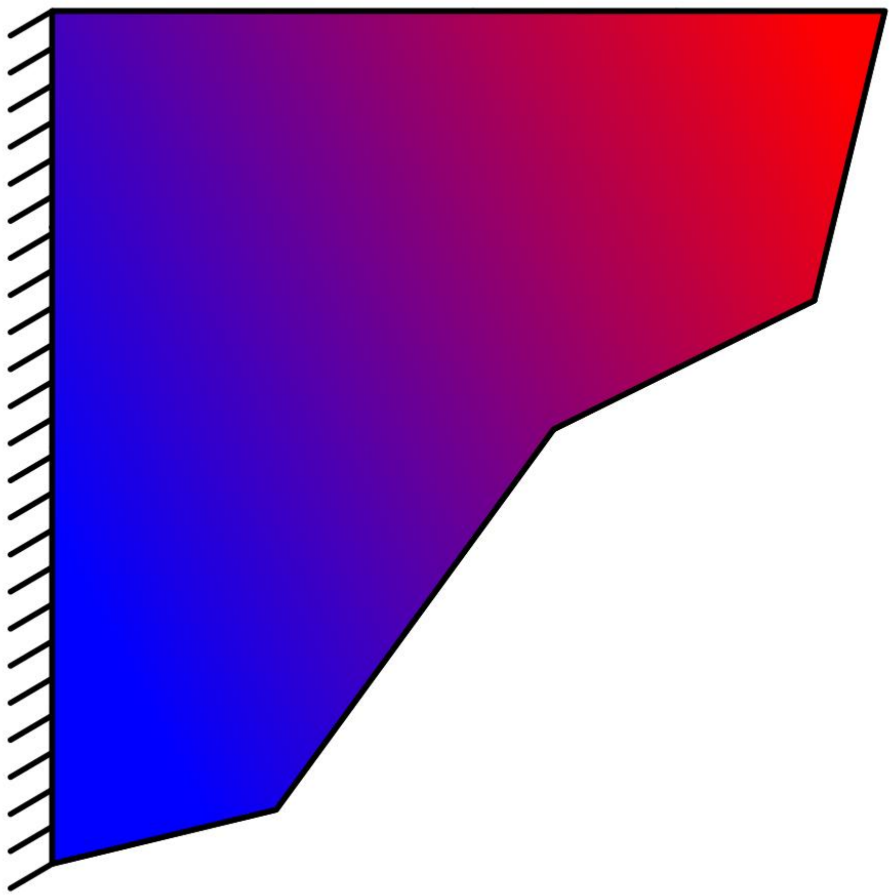
FEM Puzzle Pieces: What does Ferrite provide?

1) Geometry
2) Mesh
3) **Degrees of Fre**
   - Vector field,
   - Scalar field, p (only purple domain)
4) **Tools for assembly**
   - Shape functions evaluation (interpolations)
   - Mapping shape functions to cell coordinates
   - Quadrature rules
   - Assemble local into to global sparse matrix
5) **Constraint handling**
6) Linear solver
7) Post-processing
   - **Quadrature point data (*L2Projector*)**
   - Visualization

- Simple builtin mesh generator
- Gmsh interface package **`FerriteGmsh.jl`**
- Abaqus input file parser **`FerriteMeshParser.jl`**

- ”\”
- **`LinearSolve.jl`**

- Builtin VTK (ParaView) export
- **`FerriteViz.jl`**



FEM Puzzle Pieces: How does Ferrite do that?

1) Tensors.jl
2) Sationary heat equation (Poisson’s equation)
3) Triangle to Quadrilateral & 2D -> 3D
4) Change to linear elasticity
5) Advanced setup
    - *Porous media with solid aggregates*
    - *Mixed element shapes*
6) Advanced showcase
   Navier-Stokes with DifferentialEquations.jl
7) *Research examples*

Tensors.jl

```julia
using Tensors

calculate_stress(ϵ, G, K) = 2G*dev(ϵ) + 3K*vol(ϵ)

function calculate_stiffness(G, K)
    I2 = one(SymmetricTensor{2,3}) # 2nd order identity
    I4 = one(SymmetricTensor{4,3}) # 4th order identity
    return 2G*(I4 - I2 ⊗ I2 / 3) + K * I2 ⊗ I2
end

G = 80.e3; K = 155.e3
ϵ = rand(SymmetricTensor{2,3}) # Random 2nd order tensor

σ = calculate_stress(ϵ, G, K)
C = calculate_stiffness(G, K)

# Automatic Differentiation (using ForwardDiff.Dual)
gradient(e -> calculate_stress(e, G, K), ϵ) ≈ C
```

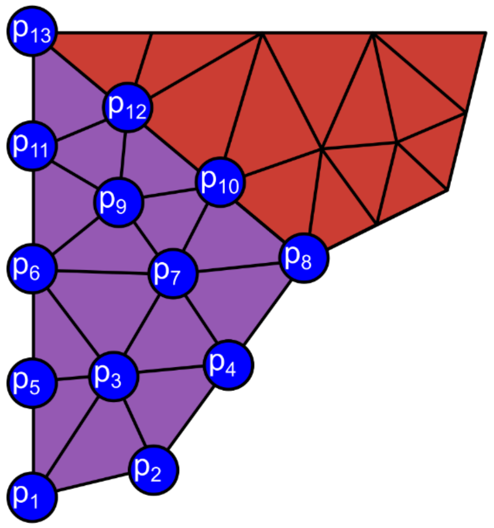
Linear Elasticity

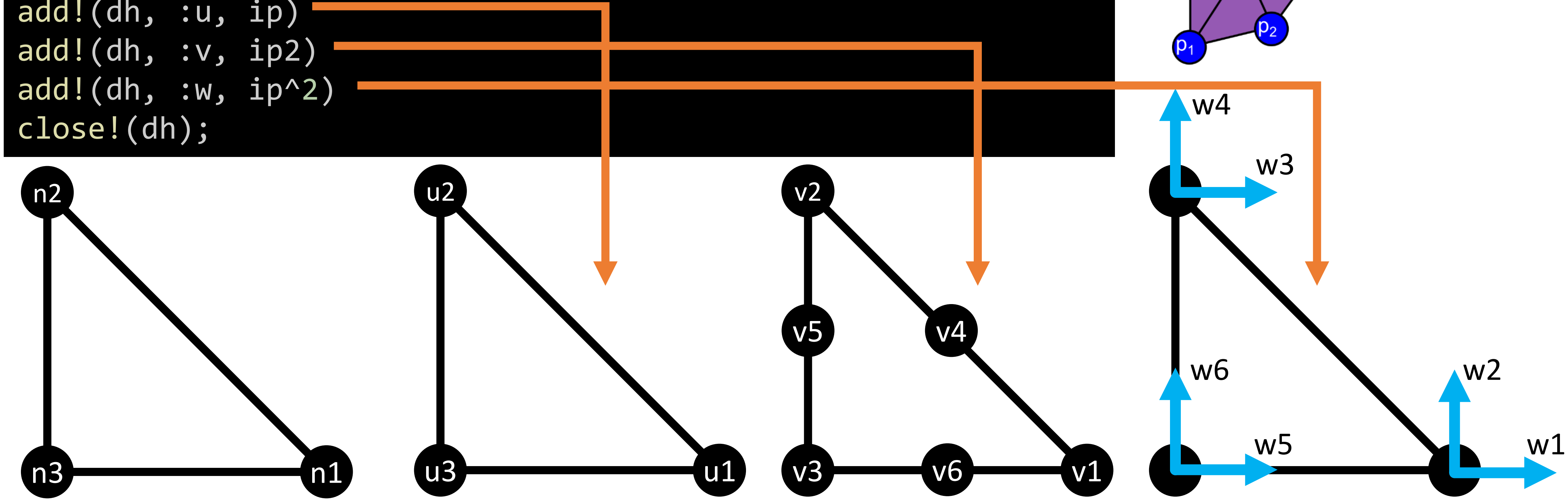
$$\boldsymbol{\sigma} = 2G\boldsymbol{\epsilon}^{\mathrm{dev}} + 3K\boldsymbol{\epsilon}^{\mathrm{vol}}$$

$$\mathbf{C} = \frac{\partial \boldsymbol{\sigma}}{\partial \boldsymbol{\epsilon}}$$
$$= 2G\left[\mathbf{I} - \frac{1}{3}\boldsymbol{I}\otimes\boldsymbol{I}\right] + K\boldsymbol{I}\otimes\boldsymbol{I}$$

DoF Distribution

```
grid = generate_grid(Triangle, (20, 20));
ip   = Lagrange{RefTriangle, 1}()
ip2 = Lagrange{RefTriangle, 2}()

dh = DofHandler(grid)
add!(dh, :u, ip)
add!(dh, :v, ip2)
add!(dh, :w, ip^2)
close!(dh);
```





`Lagrange{RefTriangle,1}()` `Lagrange{RefTriangle,2}()` `Lagrange{RefTriangle,1}()^2`

Assembly: Calculate cell contribution

Setup

```
ip = Lagrange{RefTriangle, 1}()                # Defined before
ip_geo = ip                                    # Geometric interpolation
qr = QuadratureRule{RefTriangle}(2)            # Numerical integration
cellvalues = CellValues(qr, ip, ip_geo)        # “Shape function object”
```

Element routine

```
function element_routine!(Ke::Matrix, fe::Vector, cellvalues::CellValues, x::Vector)
    # Map shape functions etc. to current geometry
    reinit!(cellvalues, x)
    # Loop over quadrature points
    for q_point in 1:getnquadpoints(cellvalues)
        # Get the quadrature weight
        dΩ = getdetJdV(cellvalues, q_point)
        # Loop over test shape functions
        for i in 1:getnbasefunctions(cellvalues)
            δN  = shape_value(cellvalues, q_point, i)
            ∇δN = shape_gradient(cellvalues, q_point, i)
            # Add contribution to fe
            fe[i] += δN * dΩ
            # Loop over trial shape functions
            for j in 1:getnbasefunctions(cellvalues)
                ∇N = shape_gradient(cellvalues, q_point, j)
                # Add contribution to Ke
                Ke[i, j] += (∇δN ⋅ ∇N) * dΩ
            end
        end
    end
    return Ke, fe
end
```

Heat Equation

$$\int_\Omega \nabla\delta u\cdot\nabla u\ \mathrm{d}\Omega = \int_\Omega \delta u\ \mathrm{d}\Omega$$

$$\left[\int_\Omega \nabla\delta N_i\cdot\nabla N_j\ \mathrm{d}\Omega\right] a_j = \int_\Omega \delta N_i\ \mathrm{d}\Omega$$

$$K_{ij}a_j = f_i$$

```julia
function element_routine!(Ke::Matrix, fe::Vector, cellvalues::CellValues, cellcoords::Vector)
    # Map shape function gradients etc. to current geometry
    reinit!(cellvalues, cellcoords)
    # Loop over quadrature points
    for q_point in 1:getnquadpoints(cellvalues)
        # Get the quadrature weight
        dΩ = getdetJdV(cellvalues, q_point)
        # Loop over test shape functions
        for i in 1:getnbasefunctions(cellvalues)
            δNᵢ  = shape_value(cellvalues, q_point, i)
            ∇δNᵢ = shape_gradient(cellvalues, q_point, i)
            # Add heat source to fe
            fe[i] += δNᵢ * dΩ
            # Loop over trial shape functions
            for j in 1:getnbasefunctions(cellvalues)
                ∇Nⱼ = shape_gradient(cellvalues, q_point, j)
                # Add contribution to Ke
                Ke[i, j] += (∇δNᵢ ⋅ ∇Nⱼ) * dΩ
            end
        end
    end
    return Ke, fe
end
```

Heat Equation

$$\int_\Omega \nabla\delta u \cdot \nabla u \; \mathrm{d}\Omega = \int_\Omega \delta u \; \mathrm{d}\Omega$$

$$\left[\int_\Omega \nabla\delta N_i \cdot \nabla N_j \; \mathrm{d}\Omega\right] a_j = \int_\Omega \delta N_i \; \mathrm{d}\Omega$$

$$K_{ij} a_j = f_i$$

```julia
function element_routine!(Ke::Matrix, fe::Vector, cellvalues::CellValues, cellcoords::Vector)
    # Map shape function gradients etc. to current geometry
    reinit!(cellvalues, cellcoords)
    # Loop over quadrature points
    for q_point in 1:getnquadpoints(cellvalues)
        # Get the quadrature weight
        dΩ = getdetJdV(cellvalues, q_point)
        # Loop over test shape functions
        for i in 1:getnbasefunctions(cellvalues)
            δNᵢ  = shape_value(cellvalues, q_point, i)
            ∇δNᵢ = shape_gradient(cellvalues, q_point, i)
            # Add heat source to fe @ test function i
            fe[i] += δNᵢ * dΩ
            # Loop over trial shape functions
            for j in 1:getnbasefunctions(cellvalues)
                ∇Nⱼ = shape_gradient(cellvalues, q_point, j)
                # Add contribution to Ke @ test i, trial j
                Ke[i, j] += (∇δNᵢ ⋅ ∇Nⱼ) * dΩ
            end
        end
    end
    return Ke, fe
end
```

Heat Equation

$$\int_\Omega \nabla \delta u \cdot \nabla u \ \mathrm{d}\Omega = \int_\Omega \delta u \ \mathrm{d}\Omega$$

$$\boxed{\begin{aligned}\left[\int_\Omega \nabla \delta N_i \cdot \nabla N_j \ \mathrm{d}\Omega\right] \\ K_{ij}\end{aligned}}\begin{aligned} a_j &= \\ a_j &= \end{aligned}\boxed{\begin{aligned}&\int_\Omega \delta N_i \ \mathrm{d}\Omega \\ &f_i\end{aligned}}$$

Assembly: Assemble cell contribution

```julia
function assemble_global(cellvalues::CellValues, dh::DofHandler)
    # Allocate global stiffness matrix, K, and force vector, f
    K = allocate_matrix (dh)
    f = zeros(ndofs(dh))
    # Allocate the element stiffness matrix and element force vector
    n_basefuncs = getnbasefunctions(cellvalues)
    Ke = zeros(n_basefuncs, n_basefuncs)
    fe = zeros(n_basefuncs)

    assembler = start_assemble(K, f) # Create an assembler
    for cell in CellIterator(dh)     # Loop over all cells
        fill!(Ke, 0); fill!(fe, 0)   # Reset Ke and fe
        # Compute element contribution
        element_routine!(Ke, fe, cellvalues, getcoordinates(cell))
        # Assemble local, Ke and fe, into global, K and f
        assemble!(assembler, celldofs(cell), Ke, fe)
    end
    return K, f
end
```

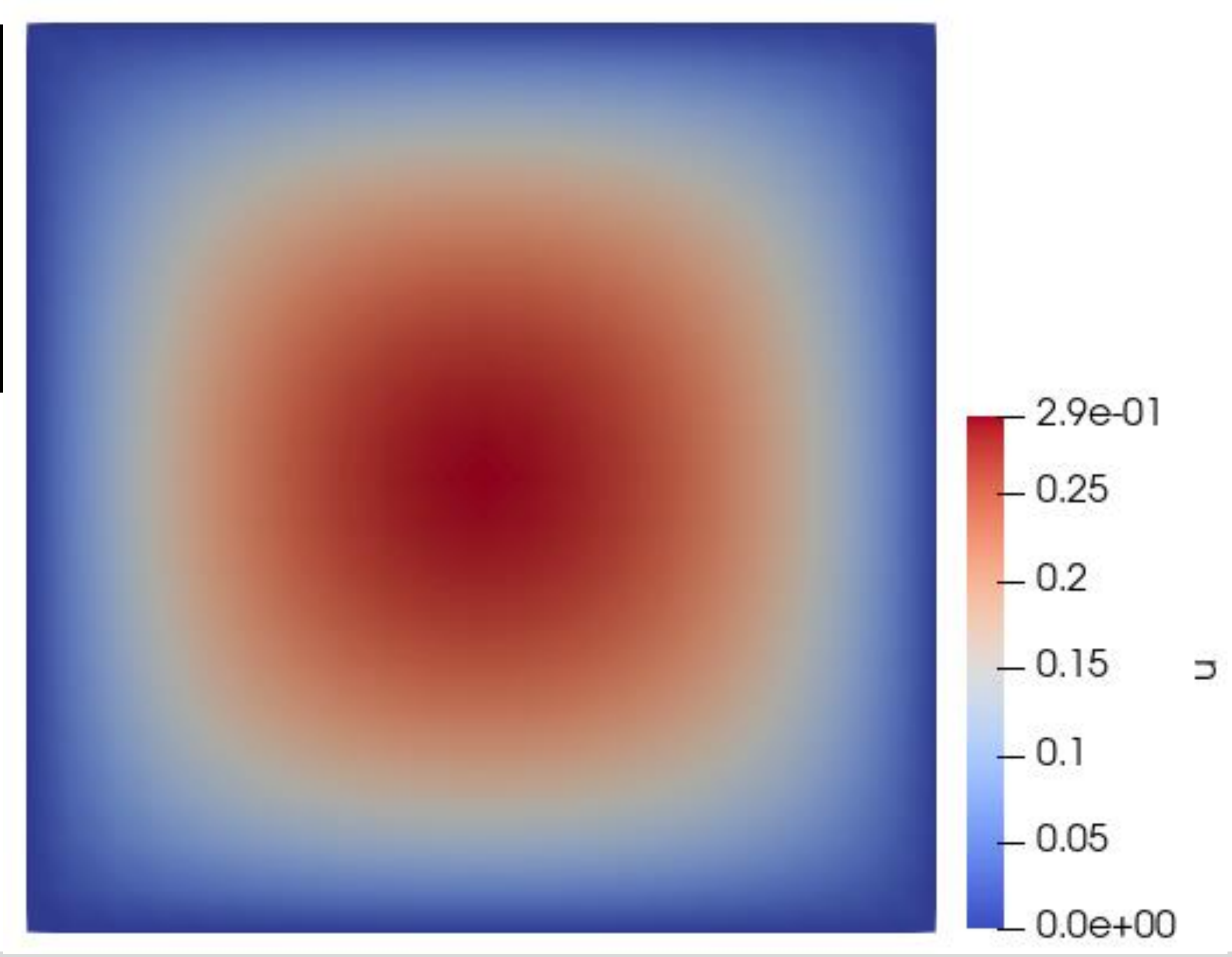
Constraints: Dirichlet Boundary Conditions

```
grid, dh, cellvalues = setup() # Pseudocode, see earlier slides

K, f = assemble_global(cellvalues, dh)

ch = ConstraintHandler(dh);

∂Ω = union(getfacetset(grid, "left"), getfacetset(grid, "right"),
           getfacetset(grid, "top"),  getfacetset(grid, "bottom"));

add!(ch, Dirichlet(:u, ∂Ω, (x, t) -> 0))

close!(ch)

apply!(K, f, ch) # Modify K and f to fulfill constraints
```

Putting it together: Stationary Heat Equation

https://ferrite-fem.github.io/Ferrite.jl/dev/tutorials/heat_equation/

```
u = K \ f # Solve linear system

# Export solution
VTKGridFile("heat_equation", dh) do vtk
    write_solution(vtk, dh, u)
end
```



Change Triangles to Quadrilateral?

```
# Old setup
grid = generate_grid(Triangle, (20, 20));
ip   = Lagrange{RefTriangle, 1}()
qr   = QuadratureRule{RefTriangle}(2)
```

```
# New setup
grid = generate_grid(Quadrilateral, (20, 20));
ip   = Lagrange{RefQuadrilateral, 1}()
qr   = QuadratureRule{RefQuadrilateral}(2)
```

Change 2d to 3d?

```julia
# Old setup
grid = generate_grid(Triangle, (20, 20));
ip   = Lagrange{RefTriangle, 1}()
qr   = QuadratureRule{RefTriangle}(2)
```

```julia
# New setup
grid = generate_grid(Tetrahedron, (20, 20, 20));
ip   = Lagrange{RefTetrahedron, 1}()
qr   = QuadratureRule{RefTetrahedron}(2)
```

Change to linear elasticity

```
# Old setup
ip   = Lagrange{RefTriangle, 1}()
```

```
# New setup (vector problem in 2d)
ip   = Lagrange{RefTriangle, 1}()^2
```

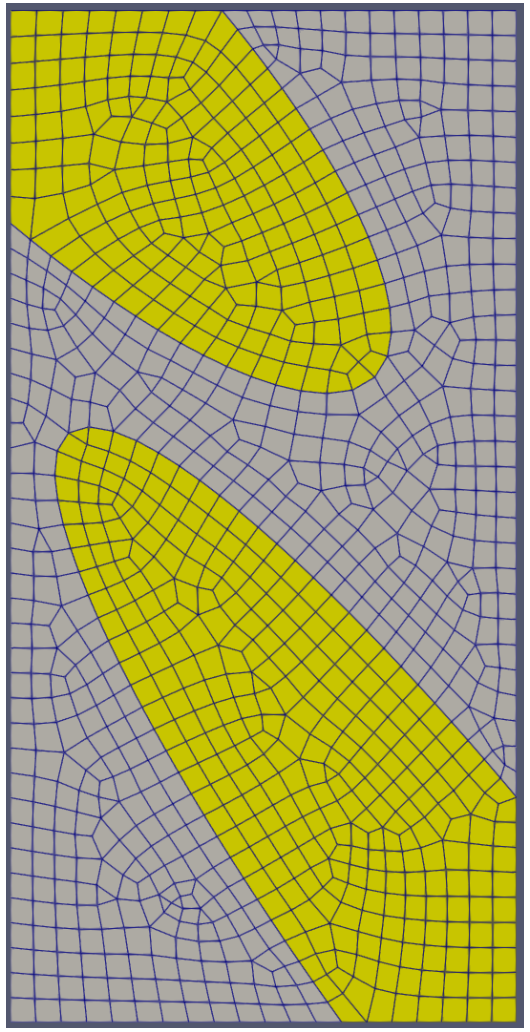
+ new physics

```
function element_routine!(Ke::Matrix, fe::Vector, cv::CellValues, cellcoords::Vector)
    reinit!(cv, cellcoords)                               # Map cellvalues to cell geometry
    for q_point in 1:getnquadpoints(cv)                   # Loop over quadrature points
        dΩ = getdetJdV(cv, q_point)                       # Get the quadrature weight
#=new=# C = calculate_stiffness(1.0, 2.0)                  # Stiffness from before
        for i in 1:getnbasefunctions(cv)                  # Loop over test shape functions
            δNᵢ  = shape_value(cv, q_point, i)
            ∇δNᵢ = shape_gradient(cv, q_point, i)
            fe[i] += δNᵢ * dΩ                             # Add heat source to fe
#=new=#     fe[i] += (δNᵢ · Vec((0.0, -1.0)))* dΩ         # Add body force to fe
            for j in 1:getnbasefunctions(cv)              # Loop over trial shape functions
                ∇Nⱼ = shape_gradient(cv, q_point, j)
                Ke[i, j] += (∇δNᵢ · ∇Nⱼ) * dΩ            # Add contribution to Ke
#=new=#         Ke[i, j] += (∇δNᵢ ⊡ C ⊡ ∇Nⱼ) * dΩ        # Add contribution to Ke
            end
        end
    end
    return Ke, fe
end
```

Change to linear elasticity

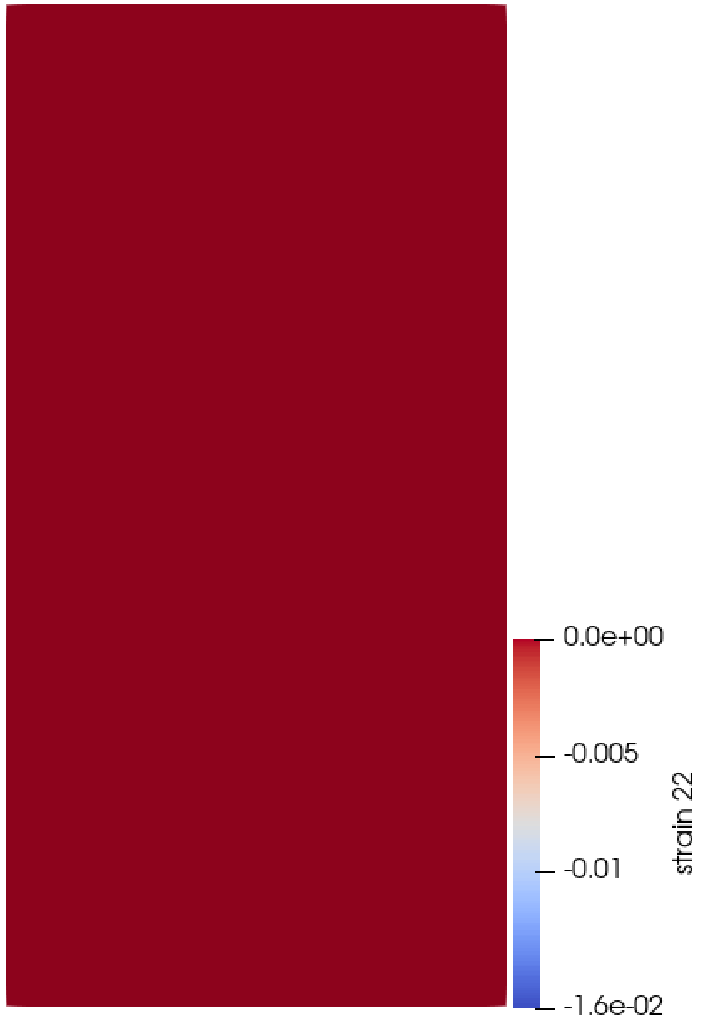
```julia
function element_routine!(Ke::Matrix, fe::Vector, cv::CellValues, cellcoords::Vector)
    reinit!(cv, cellcoords)                          # Map cellvalues to cell geometry
    for q_point in 1:getnquadpoints(cv)              # Loop over quadrature points
        dΩ = getdetJdV(cv, q_point)                  # Get the quadrature weight
#=new=# C = calculate_stiffness(1.0, 2.0)            # Stiffness from before
        for i in 1:getnbasefunctions(cv)             # Loop over test shape functions
            δNᵢ  = shape_value(cv, q_point, i)
            ∇δNᵢ = shape_gradient(cv, q_point, i)
          # fe[i] += δNᵢ * dΩ
#=new=#     fe[i] += (δNᵢ ⋅ Vec((0.0, -1.0)))* dΩ    # Add body force to fe
            for j in 1:getnbasefunctions(cv)         # Loop over trial shape functions
                ∇Nⱼ = shape_gradient(cv, q_point, j)
              # Ke[i, j] += (∇δNᵢ ⋅ ∇Nⱼ) * dΩ
#=new=#         Ke[i, j] += (∇δNᵢ ⊡ C ⊡ ∇Nⱼ) * dΩ  # Add contribution to Ke
            end
        end
    end
    return Ke, fe
end
```

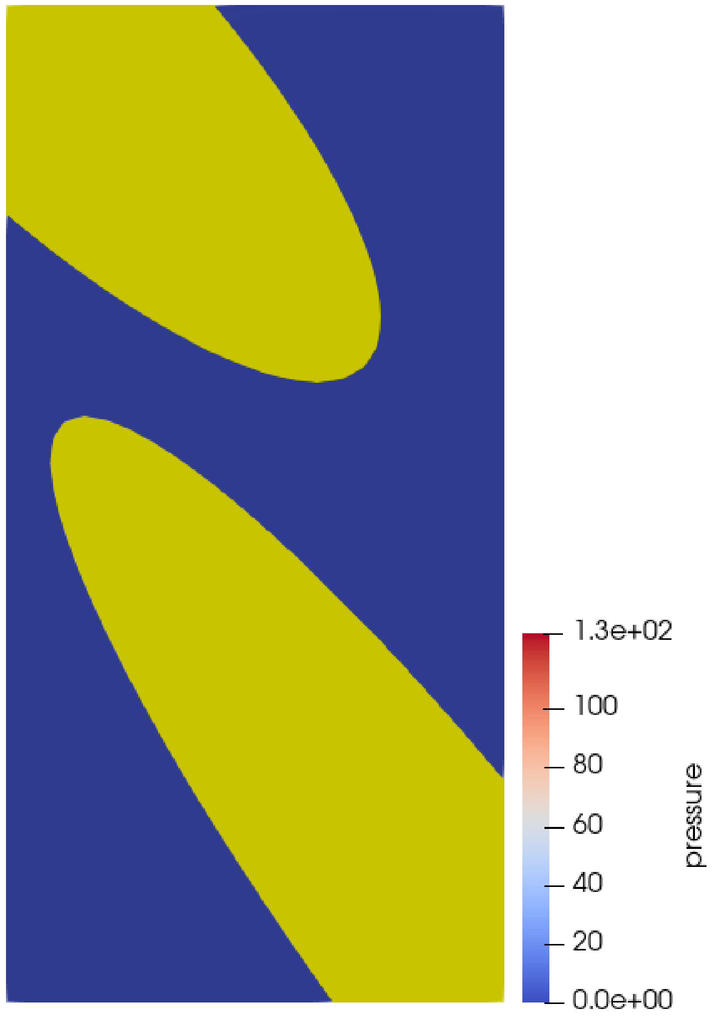
More advanced cases:
Porous media, grid with mixed element shapes



Solid aggregates
(Linear elasticity)

Porous matrix
(Linear poro-elasticity)

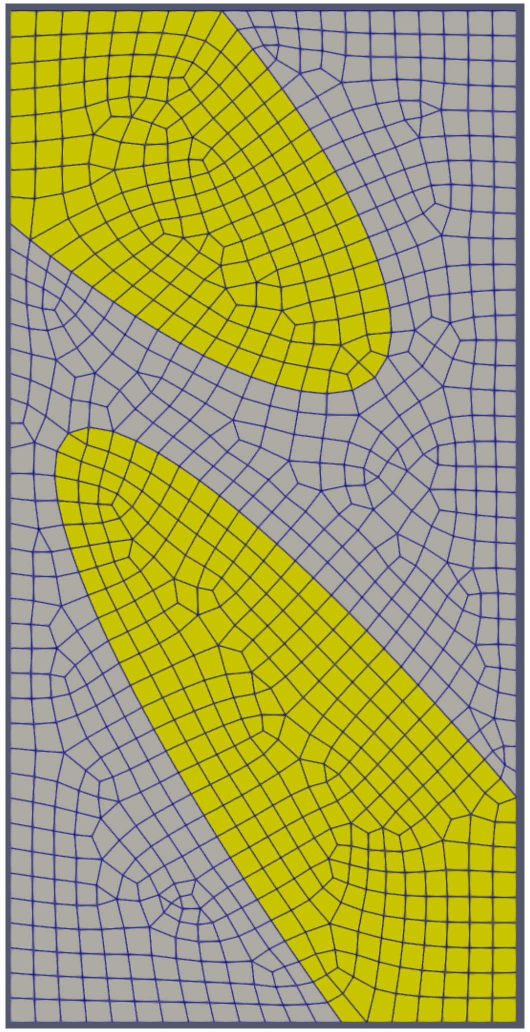




More advanced cases: Porous media, mixed grid

Solid aggregates
Linear elasticity
Displacement, **u**(**x**)
Quads and triangles

Porous matrix
Linear poro-elasticity
Pressure, p(**x**), and displacement, **u**(**x**)
Quads and triangles

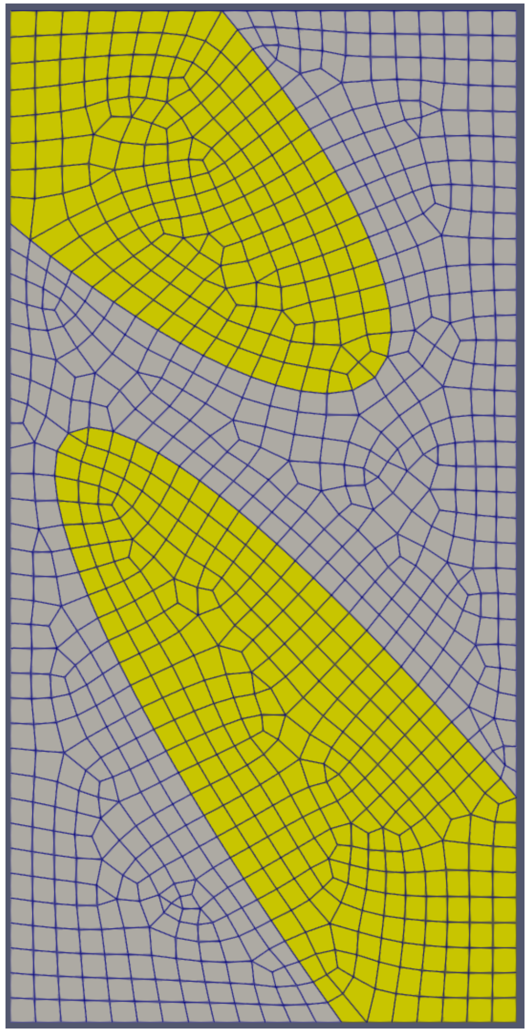


```
# Define interpolations
#  Quadratic displacement, quad elements
ipu_quad = Lagrange{RefQuadrilateral, 2}()^2
#  Quadratic displacement, triangular elements
ipu_tri  = Lagrange{RefTriangle, 2}()^2
#  Linear pressure, quad elements
ipp_quad = Lagrange{RefQuadrilateral, 1}()
#  Linear pressure, triangular elements
ipp_tri  = Lagrange{RefTriangle, 1}()

# Quadrature rules
qr_quad = QuadratureRule{RefQuadrilateral}(2) # 2x2 quadrature
qr_tri  = QuadratureRule{RefTriangle}(2)      # 3 quadrature points
```

More advanced cases: Porous media, mixed grid

Solid aggregates
Linear elasticity
Displacement, **u**(**x**)
Quads and triangles

Porous matrix
Linear poro-elasticity
Pressure, p(**x**), and displacement, **u**(**x**)
Quads and triangles



```
# Setup the DofHandler
dh = DofHandler(grid)
# Solid quads
sdh_solid_quad = SubDofHandler(dh, getcellset(grid,"solid4"))
add!(sdh_solid_quad, :u, ipu_quad)
# Solid triangles
sdh_solid_tri = SubDofHandler(dh, getcellset(grid,"solid3"))
add!(sdh_solid_tri, :u, ipu_tri)
# Porous quads
sdh_porous_quad = SubDofHandler(dh, getcellset(grid, "porous4"))
add!(sdh_porous_quad, :u, ipu_quad)
add!(sdh_porous_quad, :p, ipp_quad)
# Porous triangles
sdh_porous_tri = SubDofHandler(dh, getcellset(grid, "porous3"))
add!(sdh_porous_tri, :u, ipu_tri)
add!(sdh_porous_tri, :p, ipp_tri)

close!(dh)
```

Navier-Stokes

Ferrite.jl + DifferentialEquations.jl

2.2e+00
2
1.5
1
0.5
0.0e+00
v Magnitude

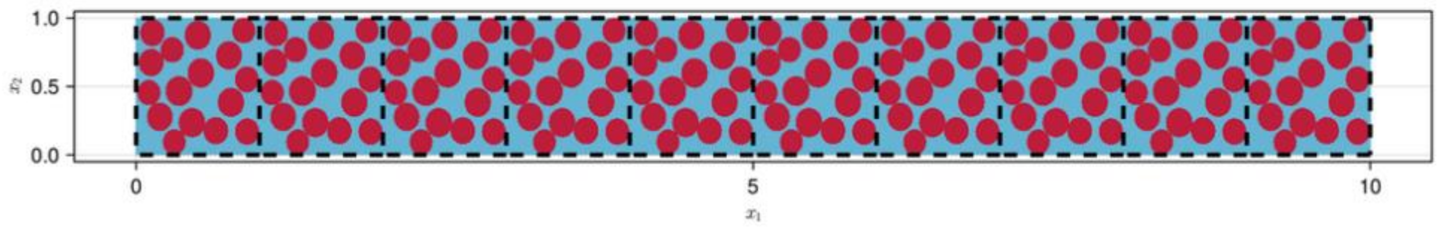
4 research examples

Homogenization of Structural Batteries

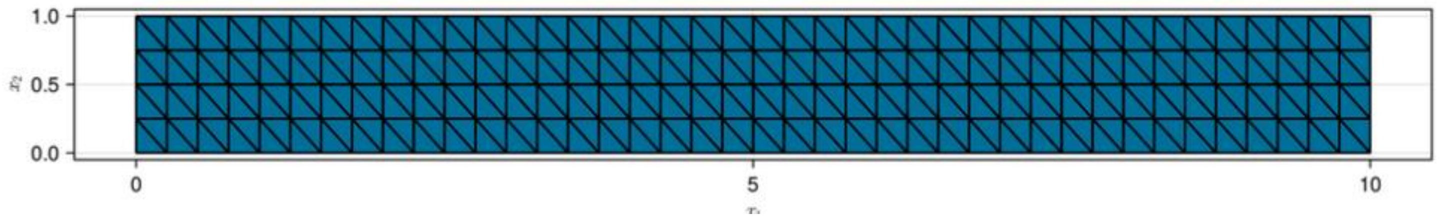
David Rollin: Institute of Applied Mechanics, TU Braunschweig

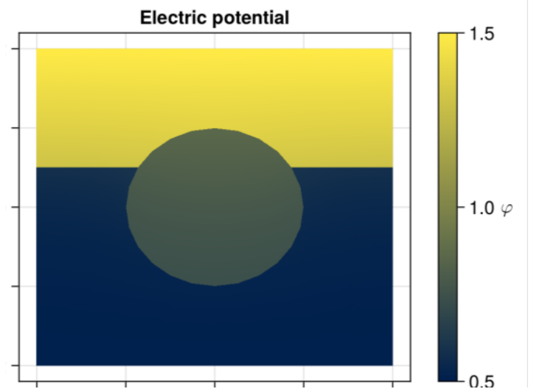
Modeling interfacial behavior in electroactive materials

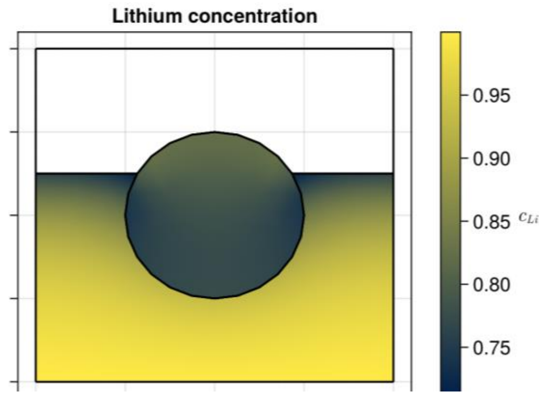
Direct Numerical Simulation (DNS) of an example problem

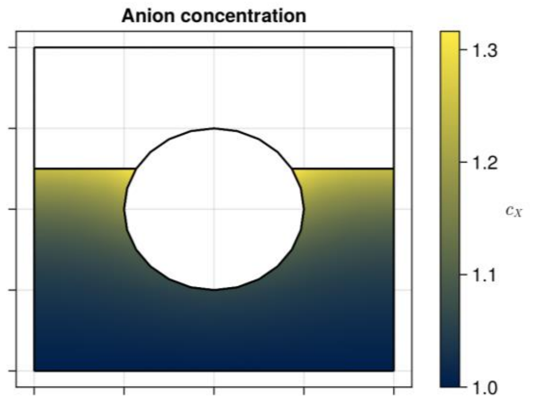


Corresponding macro-scale problem





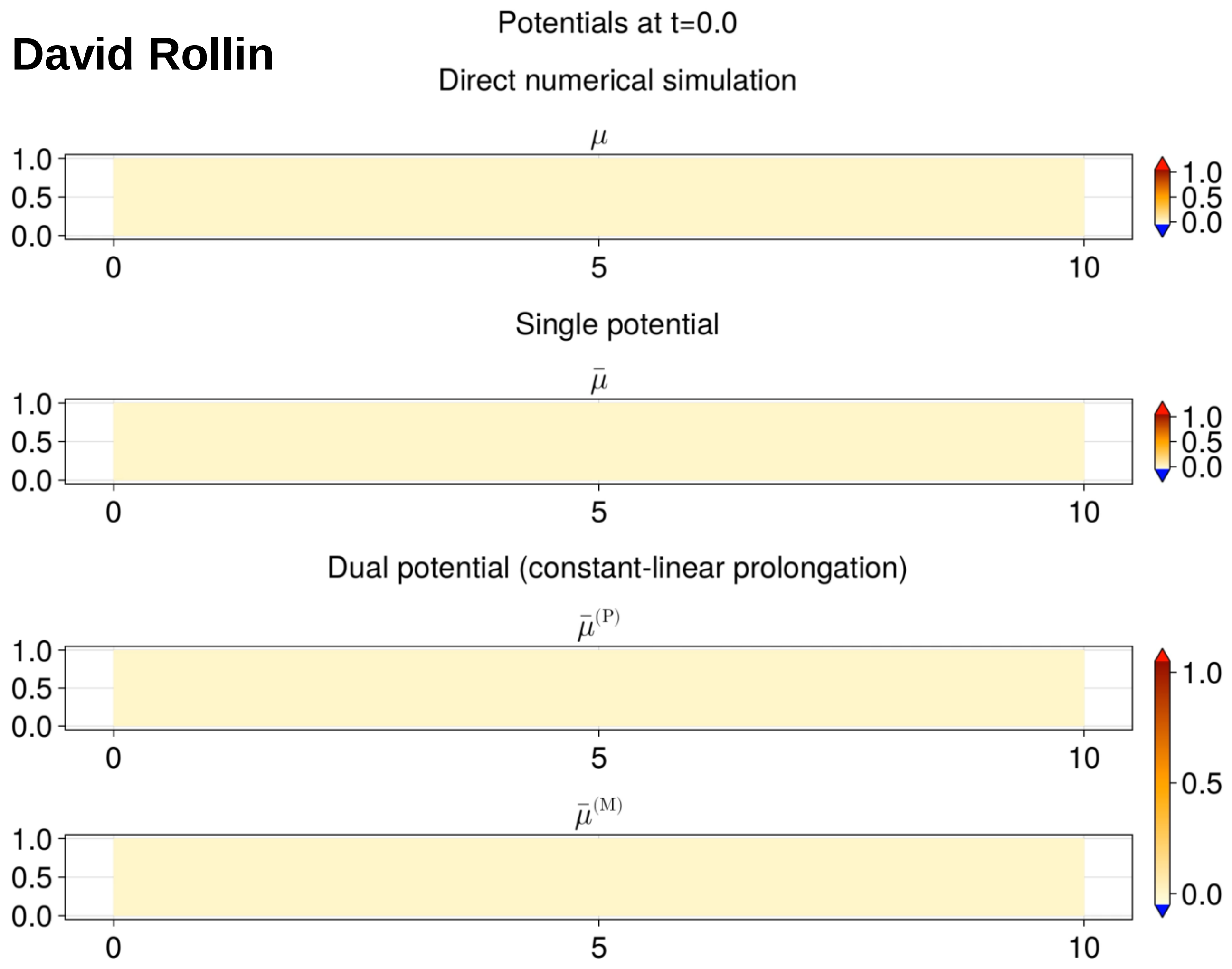


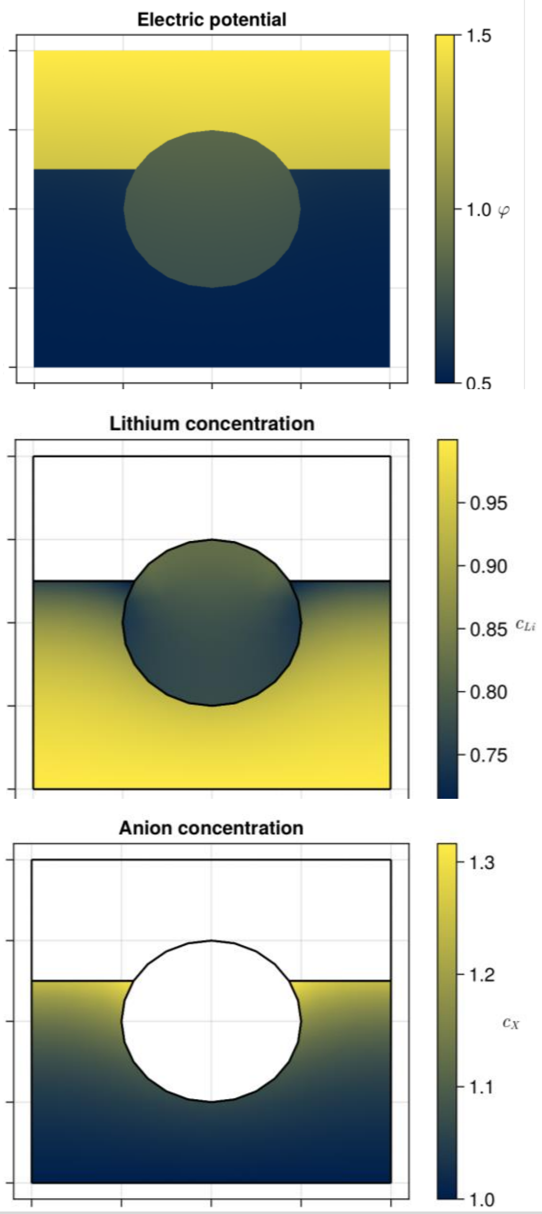


D. R. Rollin, F. Larsson, K. Runesson, and R. Jänicke, "Upscaling of chemo-mechanical properties of battery electrode material," *Int. J. Solids Struct.*, vol. 281, no. February, p. 112405, 2023,

Homogenization of Structural Batteries

David Rollin

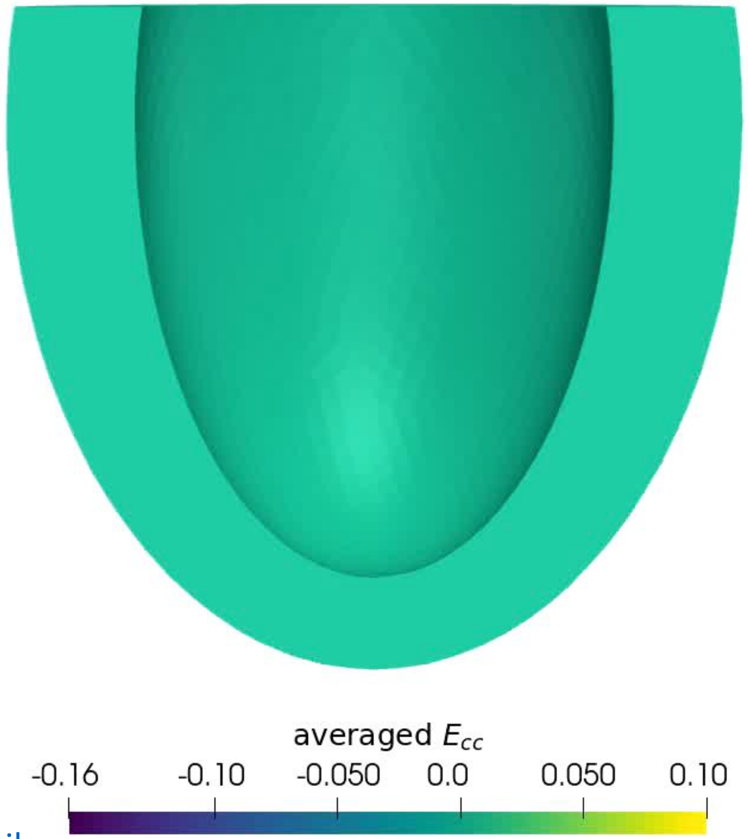


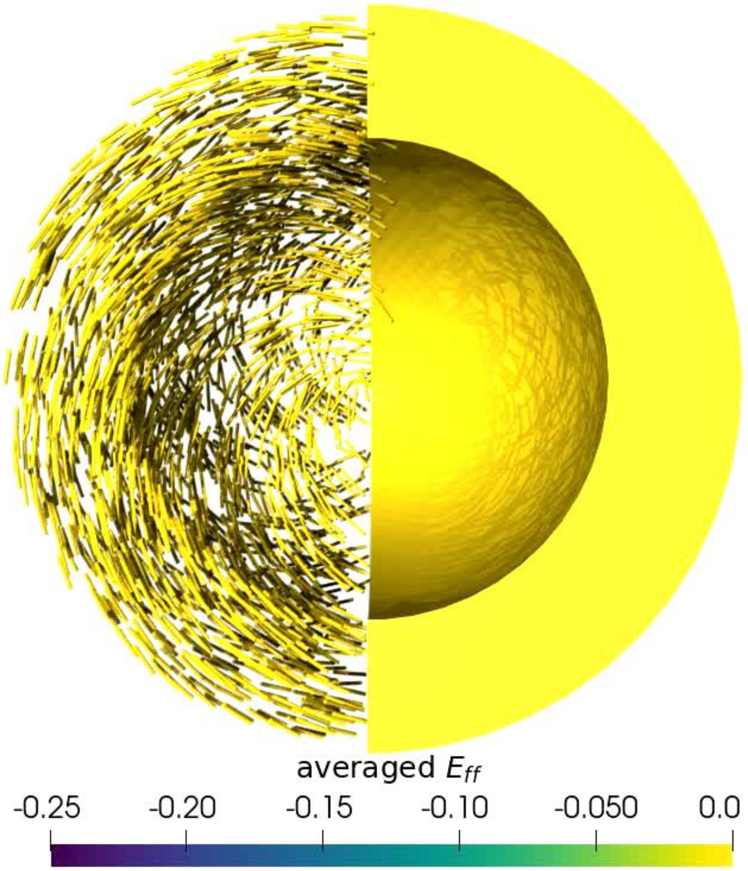


D. R. Rollin, F. Larsson, K. Runesson, and R. Jänicke, "Upscaling of chemo-mechanical properties of battery electrode material," *Int. J. Solids Struct.*, vol. 281, no. February, p. 112405, 2023,

Cardiac Multiphysics

Dennis Ogiermann: Chair of Continuum Mechanics, Ruhr-Universität Bochum





Code: github.com/termi-official/Thunderbolt.jl

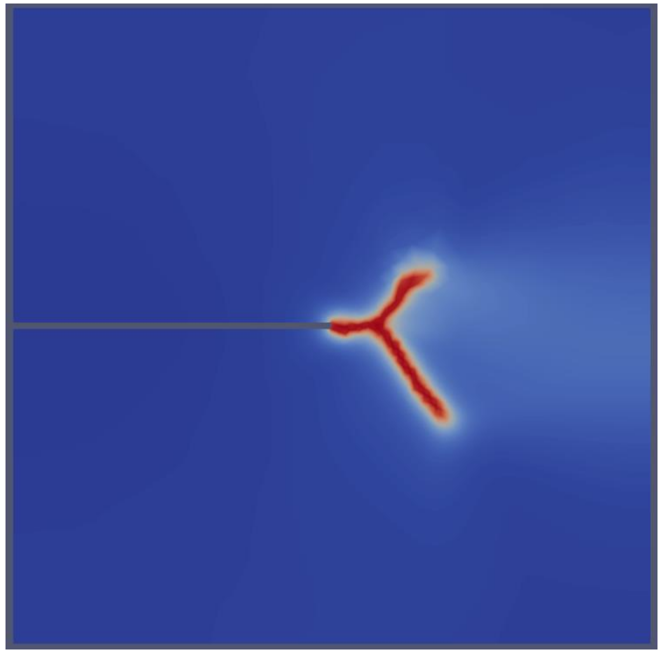
D. Ogiermann, D. Balzani, and L. E. Perotti, "An Extended Generalized Hill Model for Cardiac Tissue: Comparison with Different Approaches Based on Experimental Data," in *Functional Imaging and Modeling of the Heart*, 2023, pp. 555–564.

Stochastic **phase-field fracture**: Ensemble Kalman filtering

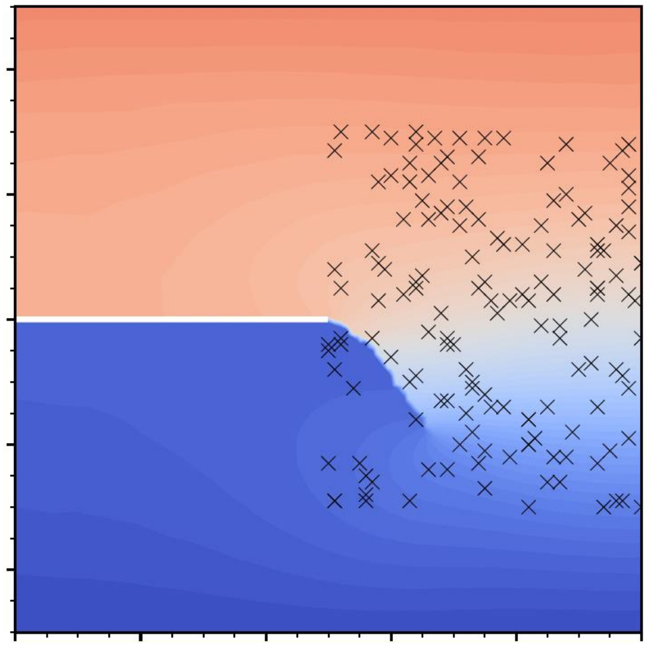
- Phase-field fracture simulations using Ferrite.jl
- Ensemble Kalman filtering to update simulations

Lucas Hermann
TU Braunschweig

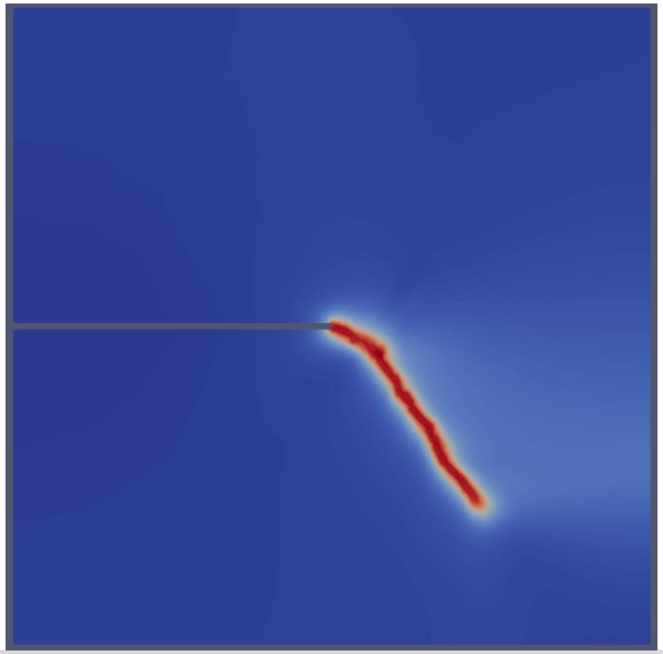
Initial model prediction



Add measurements

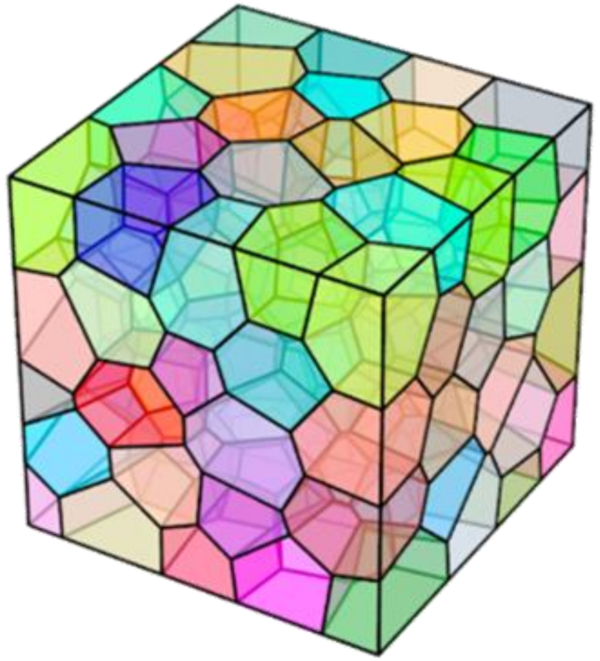


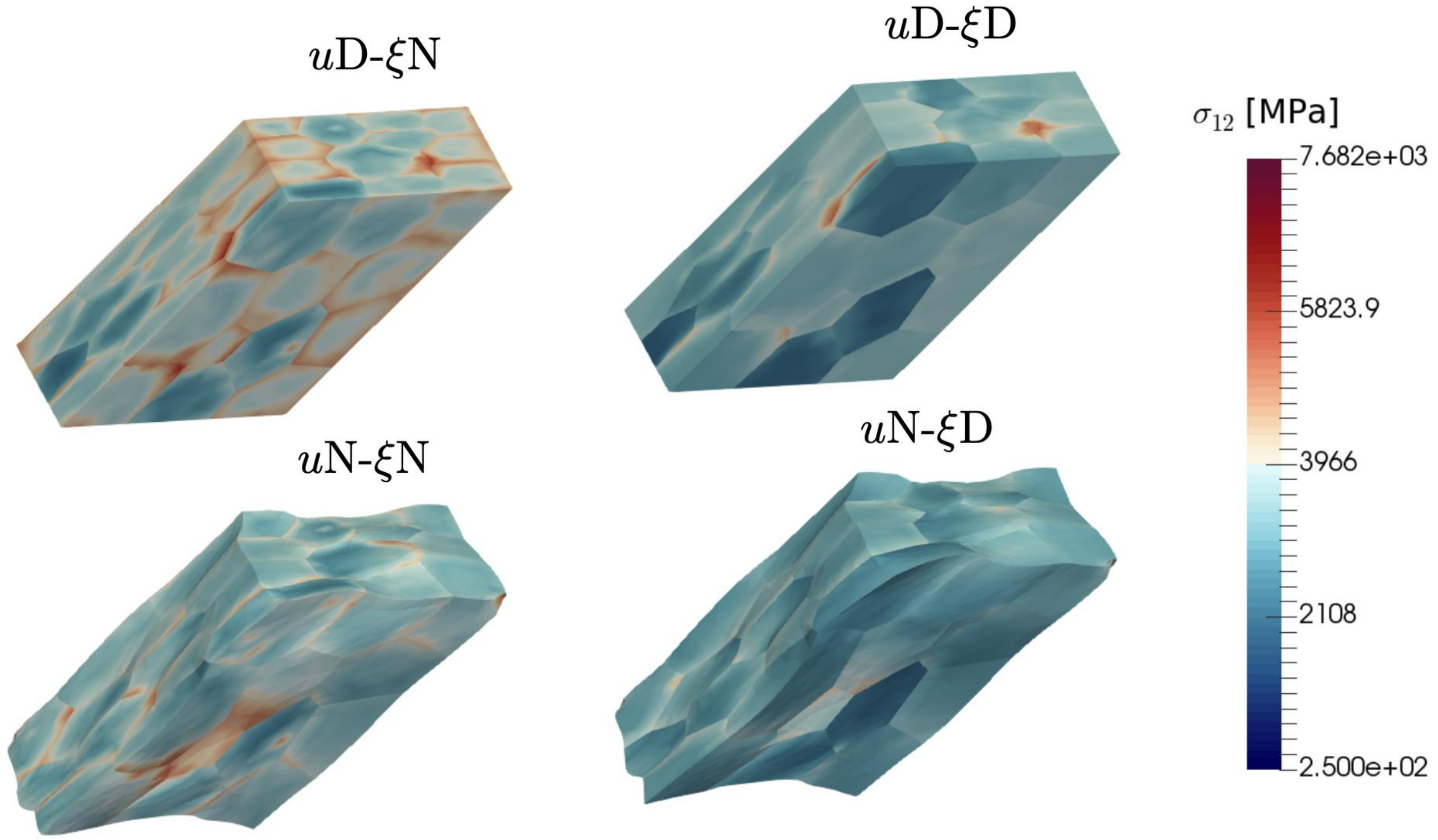
Updated model prediction



Gradient Crystal Plasticity

Kristoffer Carlsson: Division of Material and Computational Mechanics, Chalmers, Sweden





K. Carlsson, F. Larsson, and K. Runesson, “Bounds on the effective response for gradient crystal inelasticity based on homogenization and virtual testing,” *Int. J. Numer. Methods Eng.*, vol. 119, no. 4, pp. 281–304, 2019

Recent and ongoing work

- **Discontinuous Lagrange**: Abdulaziz Hamid, (Google Summer of Code / RUB)
- **IGA and Immersed Methods**: Elias Börjesson (Chalmers)
- **Interface Elements**: David Rollin (TU BS) and Kim Louisa Auth (Chalmers)
- **Mesh Adaptivity**: Maximilian Köhler (RUB)
- **Distributed Assembly**: Dennis Ogiermann (RUB)
- **Hdiv and Hcurl Interpolations**: Knut Andreas Meyer (TU BS)
- **Flexible sparsity pattern:** Fredrik Ekre (JuliaHub)

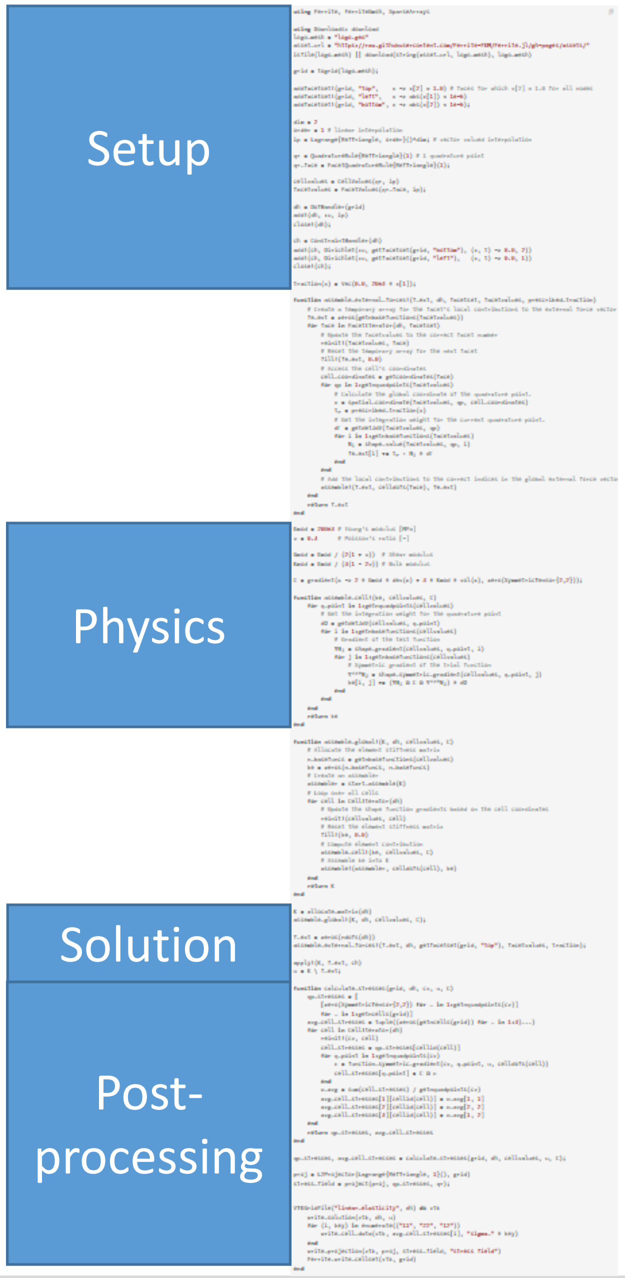
FerriteCon2023: https://ferrite-fem.github.io/FerriteCon/
Recorded talks: https://www.youtube.com/playlist?list=PLP8iPy9hna6RTilqYSvKzfTVh291rwo2l

Linear Elasticity Tutorial

Problem-specific parts

What is the rest?
Assembly

Why not in Ferrite?
Common element interface



```julia
using Ferrite, FerriteGmsh, SparseArrays

using Downloads: download
logo_mesh = "logo.geo"
asset_url = "https://raw.githubusercontent.com/Ferrite-FEM/Ferrite.jl/gh-pages/assets/"
isfile(logo_mesh) || download(string(asset_url, logo_mesh), logo_mesh)

grid = togrid(logo_mesh);

addfacetset!(grid, "top",    x -> x[2] ≈ 1.0) # faces for which x[2] ≈ 1.0 for all nodes
addfacetset!(grid, "left",   x -> abs(x[1]) < 1e-6)
addfacetset!(grid, "bottom", x -> abs(x[2]) < 1e-6);

dim = 2
order = 1 # linear interpolation
ip = Lagrange{RefTriangle, order}()^dim; # vector valued interpolation

qr = QuadratureRule{RefTriangle}(1) # 1 quadrature point
qr_face = FacetQuadratureRule{RefTriangle}(1);

cellvalues = CellValues(qr, ip)
facetvalues = FacetValues(qr_face, ip);

dh = DofHandler(grid)
add!(dh, :u, ip)
close!(dh);

ch = ConstraintHandler(dh)
add!(ch, Dirichlet(:u, getfacetset(grid, "bottom"), (x, t) -> 0.0, 2))
add!(ch, Dirichlet(:u, getfacetset(grid, "left"),   (x, t) -> 0.0, 1))
close!(ch);

traction(x) = Vec(0.0, 20e3 * x[1]);

function assemble_external_forces!(f_ext, dh, facetset, facetvalues, prescribed_traction)
    # Create a temporary array for the facet's local contributions to the external force vector
    fe_ext = zeros(getnbasefunctions(facetvalues))
    for face in FacetIterator(dh, facetset)
        # Update the facetvalues to the correct facet number
        reinit!(facetvalues, face)
        # Reset the temporary array for the next facet
        fill!(fe_ext, 0.0)
        # Access the cell's coordinates
        cell_coordinates = getcoordinates(face)
        for qp in 1:getnquadpoints(facetvalues)
            # Calculate the global coordinate of the quadrature point.
            x = spatial_coordinate(facetvalues, qp, cell_coordinates)
            tₚ = prescribed_traction(x)
            # Get the integration weight for the current quadrature point.
            dΓ = getdetJdV(facetvalues, qp)
            for i in 1:getnbasefunctions(facetvalues)
                Nᵢ = shape_value(facetvalues, qp, i)
                fe_ext[i] += tₚ ⋅ Nᵢ * dΓ
            end
        end
        # Add the local contributions to the correct indices in the global external force vector
        assemble!(f_ext, celldofs(face), fe_ext)
    end
    return f_ext
end

Emod = 200e3 # Young's modulus [MPa]
ν = 0.3      # Poisson's ratio [-]

Gmod = Emod / (2(1 + ν))  # Shear modulus
Kmod = Emod / (3(1 - 2ν)) # Bulk modulus

C = gradient(ϵ -> 2 * Gmod * dev(ϵ) + 3 * Kmod * vol(ϵ), zero(SymmetricTensor{2,2}));

function assemble_cell!(ke, cellvalues, C)
    for q_point in 1:getnquadpoints(cellvalues)
        # Get the integration weight for the quadrature point
        dΩ = getdetJdV(cellvalues, q_point)
        for i in 1:getnbasefunctions(cellvalues)
            # Gradient of the test function
            ∇Nᵢ = shape_gradient(cellvalues, q_point, i)
            for j in 1:getnbasefunctions(cellvalues)
                # Symmetric gradient of the trial function
                ∇ˢʸᵐNⱼ = shape_symmetric_gradient(cellvalues, q_point, j)
                ke[i, j] += (∇Nᵢ ⊡ C ⊡ ∇ˢʸᵐNⱼ) * dΩ
            end
        end
    end
    return ke
end

function assemble_global!(K, dh, cellvalues, C)
    # Allocate the element stiffness matrix
    n_basefuncs = getnbasefunctions(cellvalues)
    ke = zeros(n_basefuncs, n_basefuncs)
    # Create an assembler
    assembler = start_assemble(K)
    # Loop over all cells
    for cell in CellIterator(dh)
        # Update the shape function gradients based on the cell coordinates
        reinit!(cellvalues, cell)
        # Reset the element stiffness matrix
        fill!(ke, 0.0)
        # Compute element contribution
        assemble_cell!(ke, cellvalues, C)
        # Assemble ke into K
        assemble!(assembler, celldofs(cell), ke)
    end
    return K
end

K = allocate_matrix(dh)
assemble_global!(K, dh, cellvalues, C);

f_ext = zeros(ndofs(dh))
assemble_external_forces!(f_ext, dh, getfacetset(grid, "top"), facetvalues, traction);

apply!(K, f_ext, ch)
u = K \ f_ext;

function calculate_stresses(grid, dh, cv, u, C)
    qp_stresses = [
        [zero(SymmetricTensor{2,2}) for _ in 1:getnquadpoints(cv)]
        for _ in 1:getncells(grid)]
    avg_cell_stresses = tuple((zeros(getncells(grid)) for _ in 1:3)...)
    for cell in CellIterator(dh)
        reinit!(cv, cell)
        cell_stresses = qp_stresses[cellid(cell)]
        for q_point in 1:getnquadpoints(cv)
            ε = function_symmetric_gradient(cv, q_point, u, celldofs(cell))
            cell_stresses[q_point] = C ⊡ ε
        end
        σ_avg = sum(cell_stresses) / getnquadpoints(cv)
        avg_cell_stresses[1][cellid(cell)] = σ_avg[1, 1]
        avg_cell_stresses[2][cellid(cell)] = σ_avg[2, 2]
        avg_cell_stresses[3][cellid(cell)] = σ_avg[1, 2]
    end
    return qp_stresses, avg_cell_stresses
end

qp_stresses, avg_cell_stresses = calculate_stresses(grid, dh, cellvalues, u, C);

proj = L2Projector(Lagrange{RefTriangle, 1}(), grid)
stress_field = project(proj, qp_stresses, qr);


VTKGridFile("linear_elasticity", dh) do vtk
    write_solution(vtk, dh, u)
    for (i, key) in enumerate(("11", "22", "12"))
        write_cell_data(vtk, avg_cell_stresses[i], "sigma_" * key)
    end
    write_projection(vtk, proj, stress_field, "stress field")
    Ferrite.write_cellset(vtk, grid)
end
```

FerriteAssembly.jl

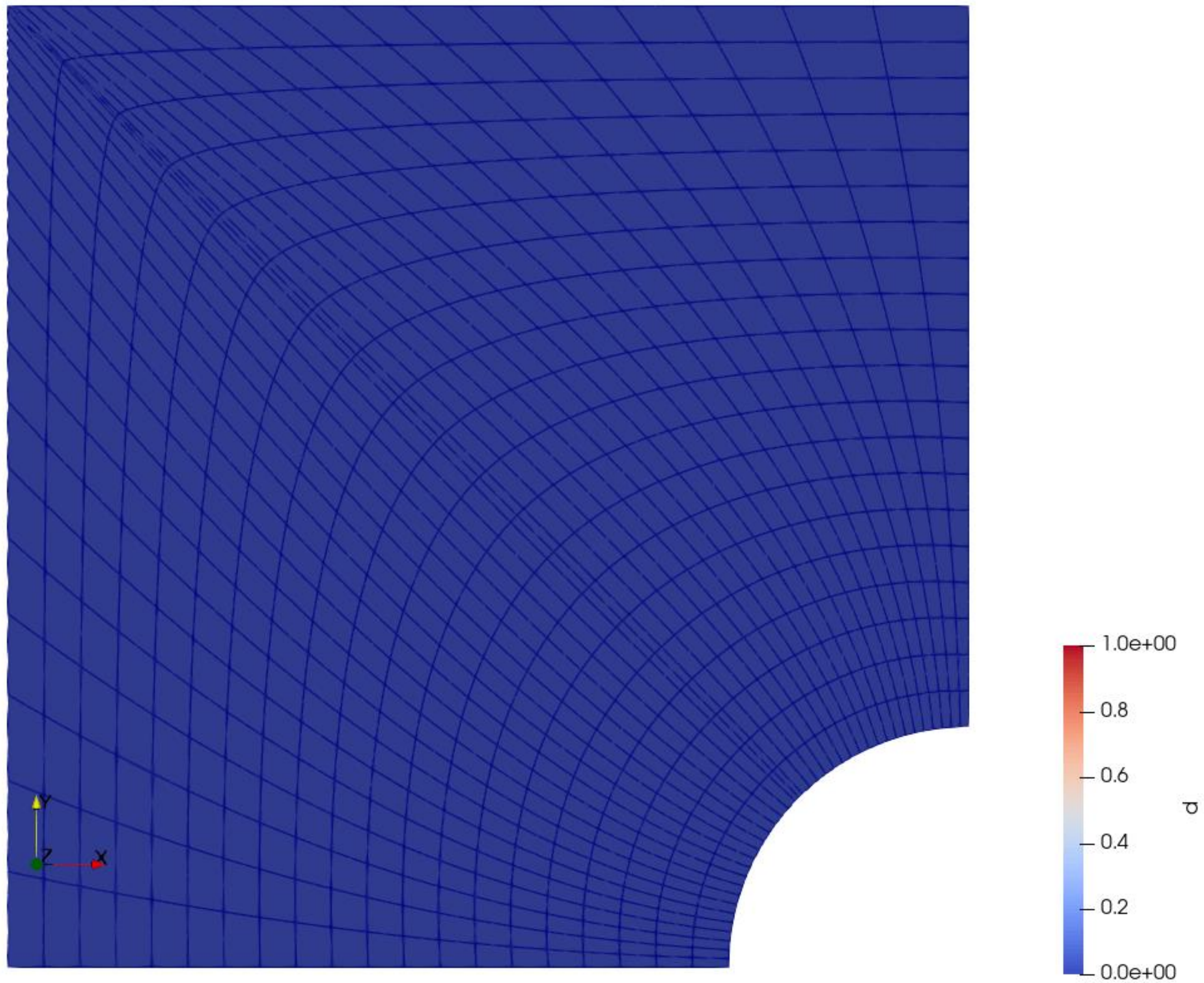
```julia
using Ferrite, FerriteAssembly
import FerriteAssembly.ExampleElements as EE
grid = generate_grid(Triangle, (20, 20))
ip = Lagrange{RefTriangle, 1}()^2

dh = DofHandler(grid)
add!(dh, :u, ip)
close!(dh)

qr = QuadratureRule{RefTriangle}(2)
cellvalues = CellValues(qr, ip)

# Material behavior
material = EE.ElasticPlaneStrain(E=1.0,ν=0.3)
ds = DomainSpec(dh, material, cellvalues)
db = setup_domainbuffer(ds; threading=true)
K = create_sparsity_pattern(dh)
f = zeros(ndofs(dh))
a = zeros(ndofs(dh))
work!(start_assemble(K, f), db; a)
```

```julia
# Body load
lh = LoadHandler(dh)
add!(lh, BodyLoad(:u, 2, Returns(Vec((0., 1.)))))
apply!(f, lh, 0.0)

# Dirichlet boundary conditions
ch = ConstraintHandler(dh)
∂Ω = union((getfacetset(grid, k) for k in
    ("left", "right", "top", "bottom"))...)
add!(ch, Dirichlet(:u, ∂Ω, Returns(Vec((0., 0.)))))
close!(ch)

apply!(K, f, ch) # Modify K and f to fulfill BC
u = K \ f;       # Solve linear system

# Export solution
VTKGridFile("solution", dh) do vtk
    write_solution(vtk, dh, u)
end
```

Phase-field fracture with IGA.jl

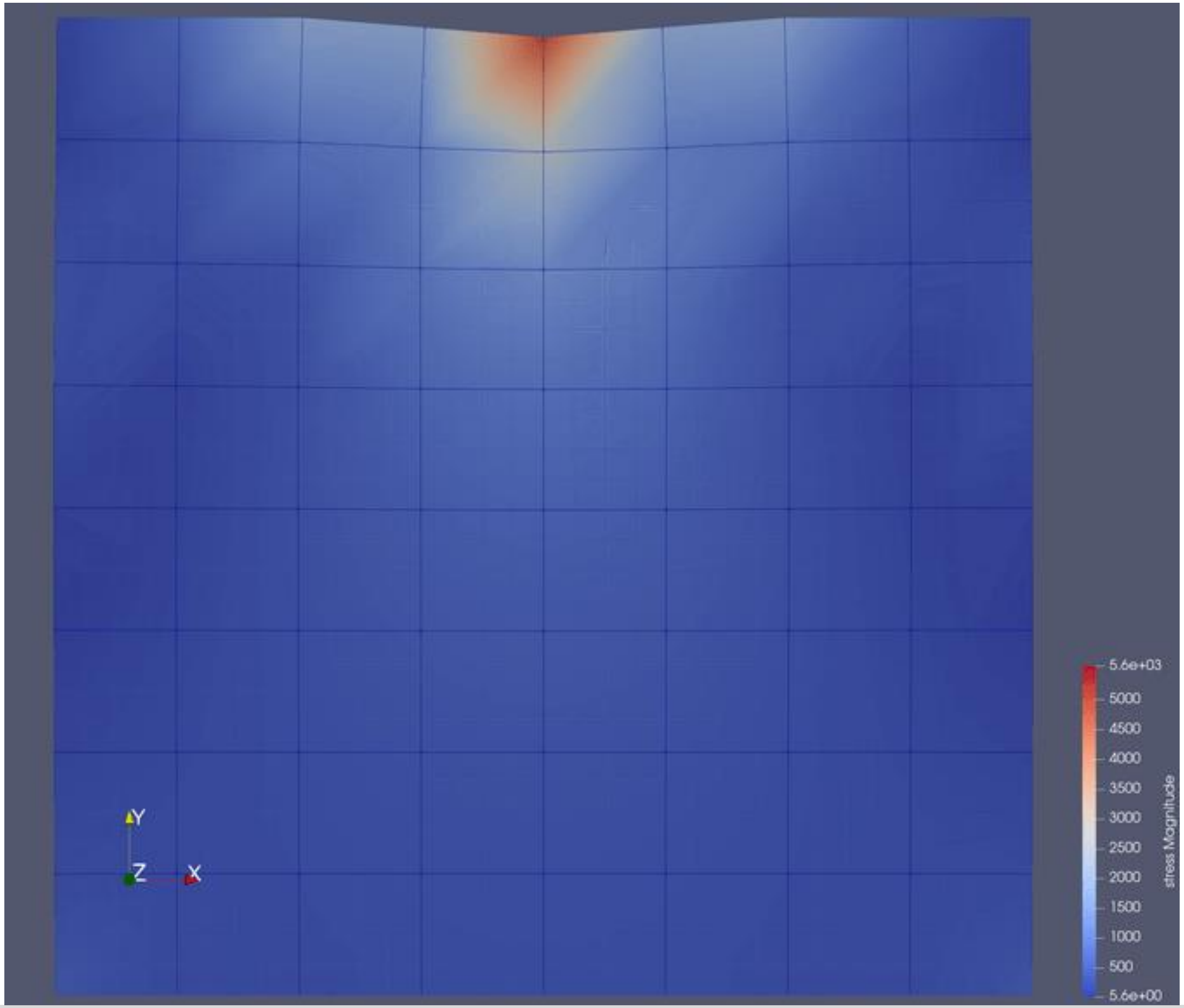
- Just for fun: *Phase-field brittle fracture*
    - Model from Bharali et al. (2023)
    - FerriteAssembly.jl for assembly
    - Using IGA.jl for mesh and interpolations

R. Bharali, F. Larsson, and R. Jänicke, "A micromorphic phase-field model for brittle and quasi-brittle fracture," *Comput. Mech.*, 2023



P4est Adaptive mesh refinement

Maximilian Köhler,
Ruhr Universität Bochum



Community and documentation

- **Documentation** and examples: https://ferrite-fem.github.io/Ferrite.jl/
- **Slack**: https://julialang.org/slack/, and join `#ferrite-fem`
  - Getting help
  - Sharing code snippets
  - Discussion about solving problems, theory, etc.
- **Github**
  - Issues: Requesting features / reporting bugs
  - PRs: Making fixes / enhancements
  - Discussions: Asking questions

Final thoughts

When should you **not** choose Ferrite?

- All you (ever) want to do can be solved in a commercial solver
- You don’t mind black-box / unknown parts

When should you choose Ferrite?

If you want

- an easy start, but scalability and flexibility
- to solve non-standard problems
- to learn a lot about finite elements

Ferrite.jl