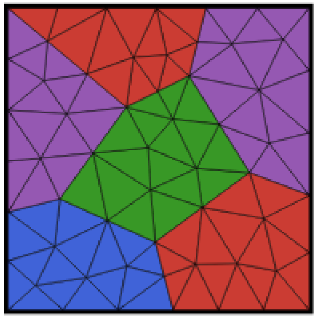
Technische Universität Braunschweig

IAM
INSTITUT FÜR ANGEWANDTE MECHANIK

Introduction and overview of `Ferrite.jl`

Fredrik Ekre

Ferrite.jl User & Developer meetup, 2022-09-26

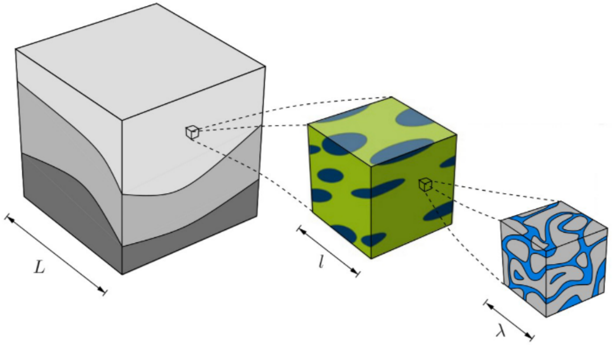


Outline

- What is Julia and `Ferrite.jl`
- Structure of a FEM program
- What `Ferrite.jl` provides
- What `Ferrite.jl` *not* provides
- Other packages and composability
- What needs more work

Who am I?

- Gothenburg, Sweden
- Education at Chalmers University of Technology
  - BSc Civil engineering (2014)
  - MSc Applied Mechanics (2016)
  - PhD Solid and Structural Mechanics (2021). Project: *Numerical model reduction and error estimation for multiscale modeling*
- Postdoc at IAM since Sept. 2021
- Started with Julia 2015 (v0.4.1)
- Co-developed `Ferrite.jl` since the start of my PhD

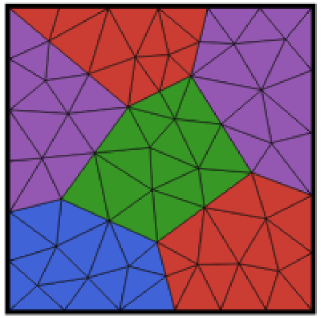


What is Julia?

- High-level dynamic programming language
- General purpose (initially targeting numerical computing)
- Free (as in beer and freedom) and open source
- Ambition to solve the “two-language problem”
- First public in 2012
- Release 1.0 in 2018

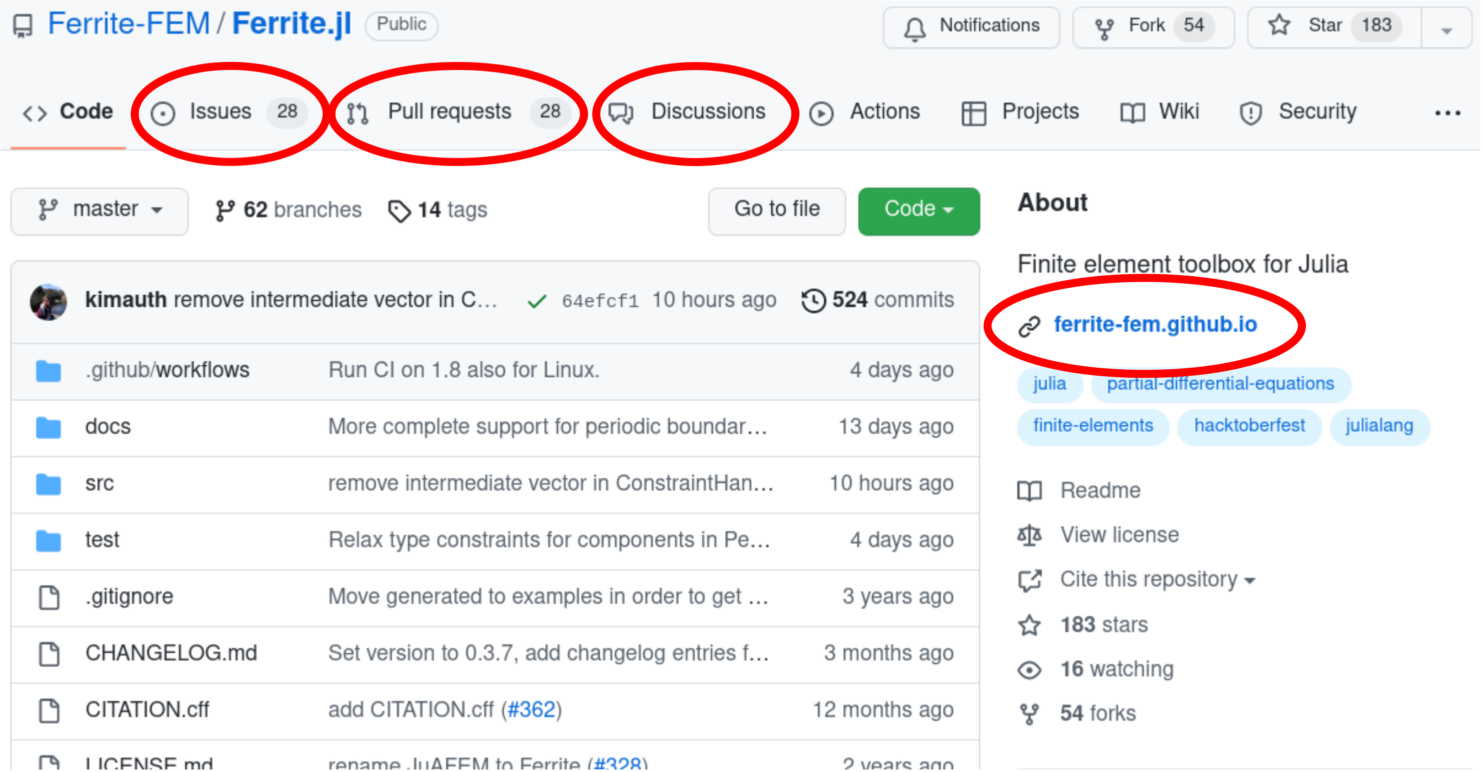
What is `Ferrite.jl`?

- Finite element toolbox written in Julia
- Not a complete FEA software, but provide many important puzzle pieces
- Initiated by Kristoffer Carlsson in 2016
- No particular road map: we have implemented what we need for our particular research problems



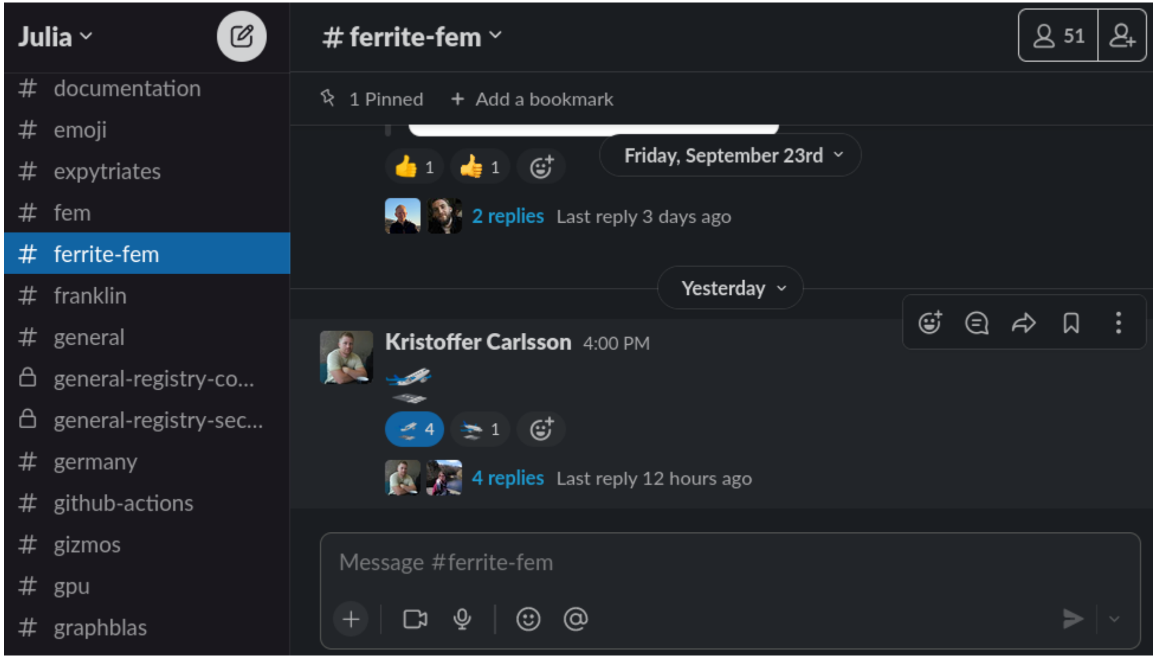
Where to find `Ferrite.jl`?

GitHub: issues, pull requests, discussions, documentation



Where to find `Ferrite.jl`?

`#ferrite-fem` channel on Julia slack workspace



FEM puzzle pieces

Pre-processing

- Geometry
- Meshing

Assembly

- Tensor operations
- Shape functions
- Material modeling
- Numerical integration
- Boundary conditions

Linear solver for $Ax = b$ (sparse system of equations)

- Direct solver
- Iterative solver

Pre-processing

- Evaluation of secondary quantities
- Visualization

FEM puzzle pieces: Pre-processing

Meshing and geometry modeling

- Not directly provided by `Ferrite.jl`
- Leverage existing software by parsing output from other software into `Ferrite.jl` grid format (Gmsh, Abaqus, …)
  - Works great for “static” meshes
  - For mesh adaptivity a library interface would be better
- Preliminary work for mesh adaptivity by Dennis/Maximilian and co-workers

FEM puzzle pieces: Assembly

- Assembly functionality is the core of `Ferrite.jl`
- Distribution and book keeping of degrees-of-freedom
- Finite element “kernel” functionality
  - Evaluation of shape functions
  - Evaluation of functions in FE-space (linear, quadratic, ...)
  - Numerical integration, quadrature rules
- (Material modeling: `Tensors.jl`)
- Routines for assemble element contributions to global system
- Utilities for Dirichlet, Neumann, periodic (Dirichlet) boundary conditions

Dof management: the `DofHandler`

Example: two-field problems with linear approximation for a pressure field and quadratic approximation for a displacement field

```
grid = setup_grid(...)                     # Generate grid

dh = DofHandler(grid)                      # Create DofHandler

push!(dh, :p, Lagrange{3,RefCube,1}()) # Linear pressure field
push!(dh, :u, Lagrange{3,RefCube,2}()) # Quadratic displacement field

close!(dh)                                 # Finalize

# Query information
ndofs_per_cell(dh)                         # Number of dofs per element
celldofs(dh, i)                            # Dofs for element i
```

FE kernel: `FEValues`

Numerical integration and shape function evaluation handled by `CellScalarValues` (or `CellVectorValues`)

```
interpolation = Lagrange{3, RefCube, 1}()   # Linear interpolation
quad_rule = QuadratureRule{dim, RefCube}(2) # Second order quadrature
cellvalues = CellScalarValues(quad_rule, interpolation)

# Value of shape function i in quadrature point qp
shape_value(cellvalues, qp, i)

# Gradient of shape function i in quadrature point qp
shape_gradient(cellvalues, qp, i)

# Value of FE approximated function with element vector ue
function_value(cellvalues, qp, ue)
```

FE kernel: `FEValues`

Example: Integration of element matrix and RHS for heat equation

```
for q_point in 1:getnquadpoints(cellvalues)
    dΩ = getdetJdV(cellvalues, q_point)
    for i in 1:n_basefuncs
        ϕi  = shape_value(cellvalues, q_point, i)
        ∇ϕi = shape_gradient(cellvalues, q_point, i)
        fe[i] += ϕi * dΩ
        for j in 1:n_basefuncs
            ∇ϕj = shape_gradient(cellvalues, q_point, j)
            Ke[i, j] += (∇ϕi ⋅ ∇ϕj) * dΩ
        end
    end
end
```

Global assembly

Efficient global sparse matrix assembly:

```
# Global tangent matrix and RHS
K = create_sparsity_pattern(dh)
f = zeros(ndofs(dh))

# Assembler for efficient sparse matrix assembly
assembler = start_assemble(K, f)

# Assemble all the elements
for i in 1:nelements
    Ke, fe = element_routine(...)
    assemble!(assembler, celldofs(dh, i), Ke, fe)
end
```

Boundary conditions: the `ConstraintHandler`

Dirichlet boundary conditions:

```
ch = ConstraintHandler(dh)                      # Constructor

dbc = Dirichlet(
         :u,                                    # Field name
         getfaceset(grid, "DBC"),               # Boundary domain
         (x, t) -> [sin(t), cos(t)],            # Prescribed value
         [1, 3]                                 # Prescribed components
)

add!(ch, dbc)                                   # Add to ConstraintHandler
close!(ch)                                      # Finalize
```

Boundary conditions: the `ConstraintHandler`

Periodic boundary conditions:

```
ch = ConstraintHandler(dh)                                  # Constructor

# Compute mapping between mirror and image faces
face_pairs = collect_periodic_faces("mirror", "image")

dbc = PeriodicDirichlet(
         :u,                                                # Field name
         face_pairs,                                        # Face mapping
         [2, 3]                                             # Constrained components
)

add!(ch, dbc)                                               # Add to ConstraintHandler
close!(ch)                                                  # Finalize
```

Boundary conditions: Neumann

`FaceScalarValues` (and `FaceVectorValues`) analoguous to `CellScalarValues` for boundary integration

(Currently somewhat “boiler-platey” and would be nice to improve!)

FEM puzzle pieces: Linear system

The default global tangent matrix is a standard Julia `SparseMatrixCSC`: use your favorite solver!
Single-core:

- Direct solvers for sparse system shipped with Julia (CHOLMOD/UMFPACK)
- Iterative solvers from other Julia packages such as `LinearSolve.jl`

Multi-core (MPI):

- `HYPRE.jl` for distributed solvers using compatible with `PartitionedArrays.jl`
- (Preliminary work by Dennis for assembly with `PartitionedArrays.jl`)
- … which he promised me yesterday to finish during the Hackathon today!

FEM puzzle pieces: Post-processing

`Ferrite.jl` provide utilities for

- Evaluation of secondary quantities
- Evaluation of primary/secondary fields in arbitrary points of the domain
- Export to VTK file format for “color mechanics”

`FerriteViz.jl`: `Makie.jl` based plotting of “`Ferrite.jl`” data.

HPC with `Ferrite.jl`?

- Julia (the language) have well developed functionality for both multi-threading (single computer) and multi-core (many computers)
  - Multi-threaded assembly: works nicely, see examples
  - Multi-core assembly: work in progress
- Easier to interface with existing HPC libraries

Composability and interaction with other packages

- `Ferrite.jl` only provide some of the puzzle pieces
- Mostly standard data structures: easy to compose with other packages. Some examples:
  - `Tensors.jl`: Fast tensor operations, automatic differentiation
  - `ForwardDiff.jl`: Automatic differentiation for element routine
  - `BlockArrays.jl`: Block matrix functionality for e.g. multifield problems
  - `MaterialModels.jl`: Library for standard material models
  - `LinearSolve.jl`: collection of linear solvers for (sparse) matrices
  - `DifferentialEquations.jl`: State of the art library for time-integration
  - `NLsolve.jl`: Algorithms for non-linear systems
  - ...

What needs more work?

- Utilities for HPC and multi-core assembly
- Mesh adaptivity
- Less boiler plate code
- Documentation!

Thanks for listening!