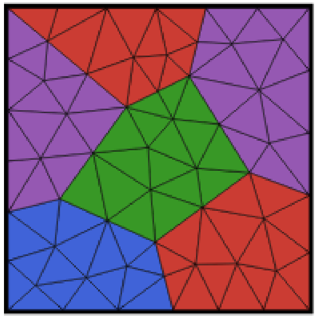
Introduction and overview of `Ferrite.jl`

Fredrik Ekre
Tampere University, 2023-08-10



Outline

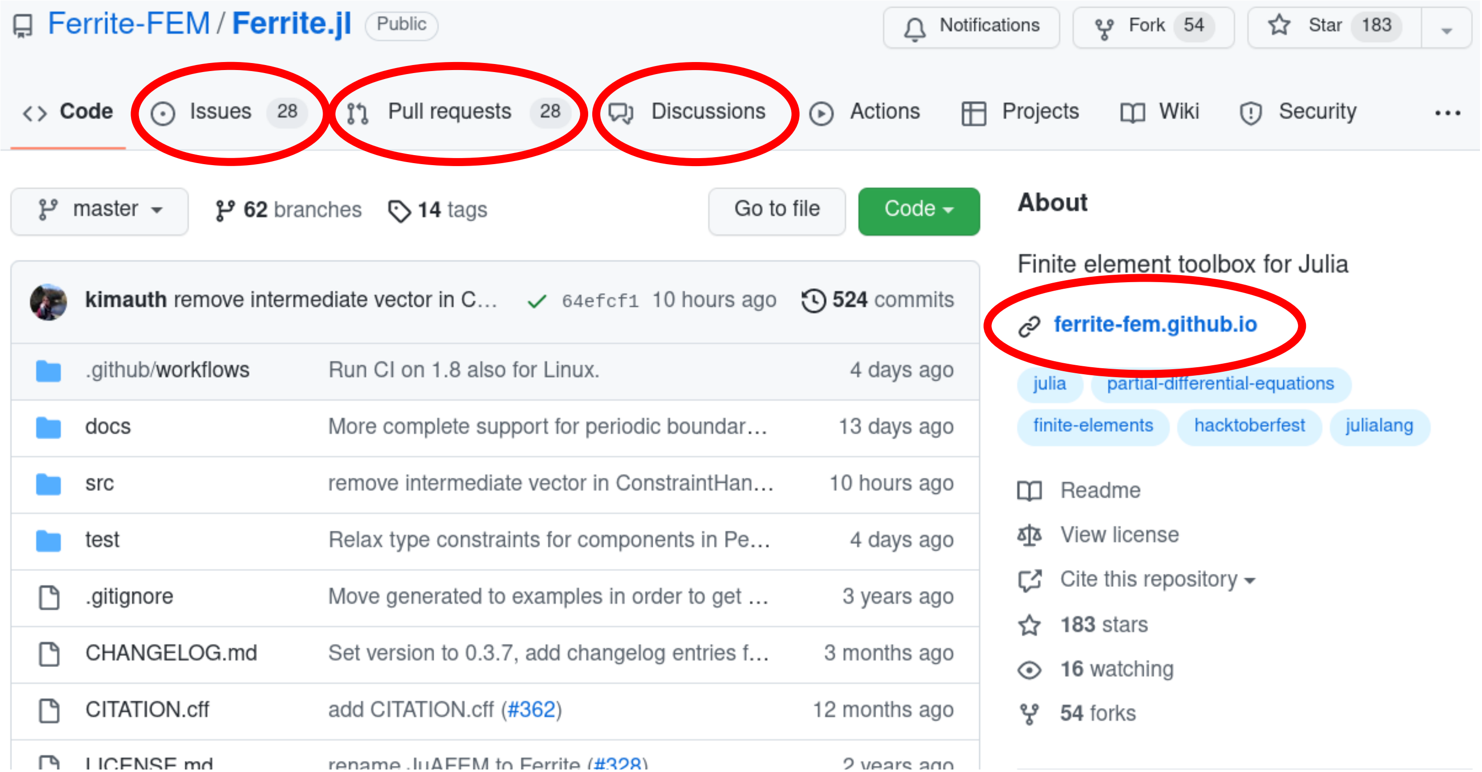
- What is Ferrite?
- Solving the heat equation
- Structure of a FEM program and the pieces provided by `Ferrite.jl`
- What `Ferrite.jl` provides
- What `Ferrite.jl` *not* provides
- Other packages and composability
- ~~What needs more work~~ What is being worked on

What is `Ferrite.jl`?

- Finite element toolbox written in Julia initiated in 2016
- Not a complete FEA software, but provide many important puzzle pieces
- Composable and hackable: easy to couple with other Julia packages and do custom things
- No particular road map: we have implemented what we need for our particular research problems

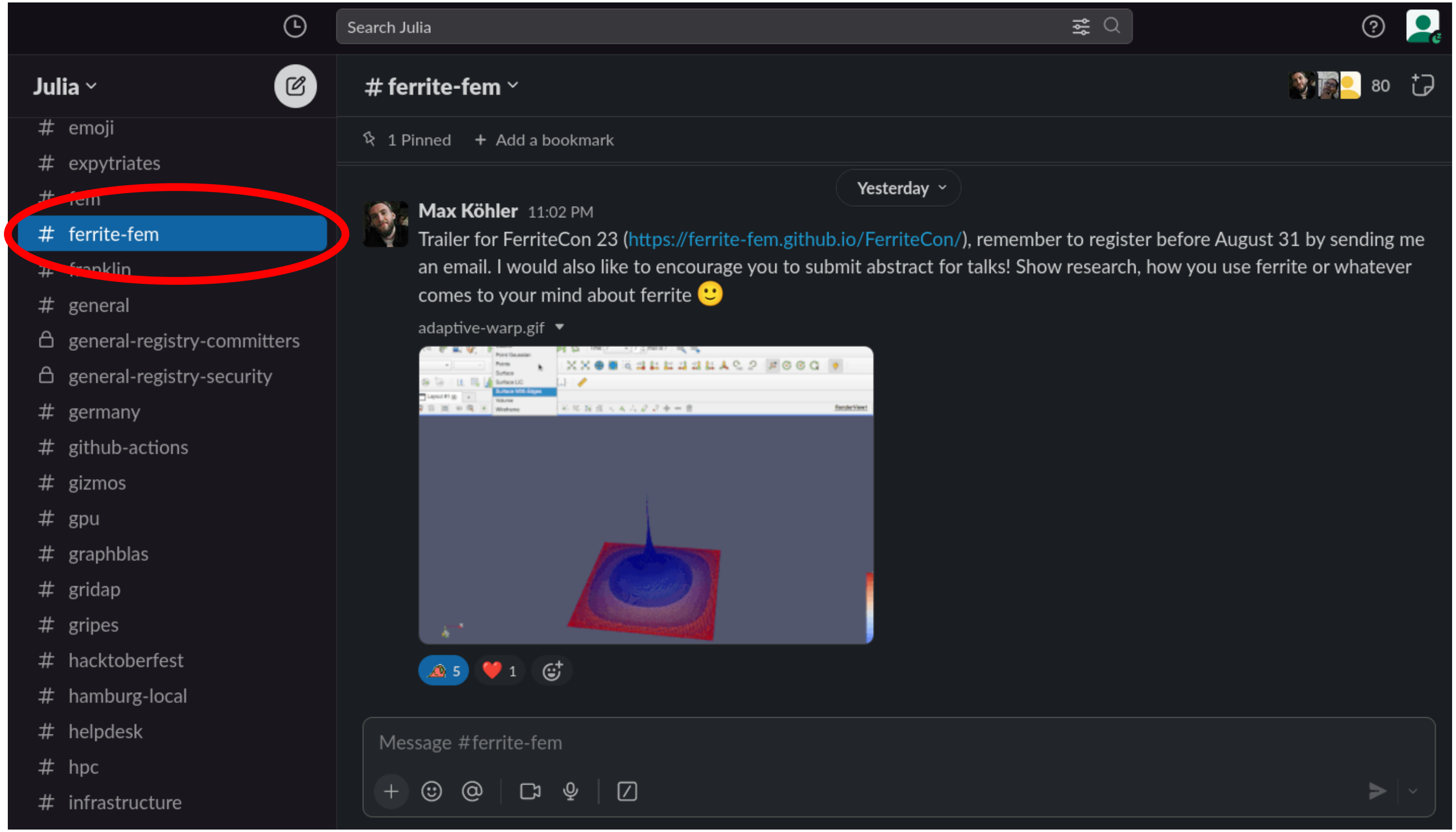
Where to find `Ferrite.jl`?

GitHub: issues, pull requests, discussions, documentation



Where to find `Ferrite.jl`?

`#ferrite-fem` channel on Julia slack workspace



Ferrite.jl User & Developer Conference

On **October 6**, we are hosting the 2^{nd} edition of Ferrite.jl User & Developer Conference at Ruhr University Bochum, Germany.

See https://ferrite-fem.github.io/FerriteCon/ for details.

RUHR UNIVERSITÄT BOCHUM RUB

LKM

GAMM

Solving the heat equation

<Demo>

Ferrite.jl - Heat equation

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

- Only the basics (hypercubes) 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
- Ongoing work for mesh adaptivity by Maximilian Köhler, Dennis Ogiermann,

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 cell in CellIterator(dh)
    Ke, fe = element_routine(...)
    assemble!(assembler, celldofs(cell), 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
update!(ch, 0.0)                                # Recompute prescribed
                                                  values for new time
```

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

```
# Face routine for boundary integration
function boundary_routine(fv::FaceScalarValues)
  fe = zeros(getnbasefunctions(fv))
  # Loop over quadrature points on the face
  for qp in 1:getnquadpoints(fv)
    # Loop over shape functions on the face
    for i in 1:getnbasefunctions(fv)
      fe[i] += (shape_value(fv, qp, i) ⋅ b) * getdetJdV(fv, qp)
    end
  end
end
```

Boundary conditions: Neumann

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

```
# Loop over all faces in the face set with label “Neumann”
for face in FaceIterator(dh, “Neumann”)
    reinit!(fv, face)                       # Update for new face
    fe = boundary_routine(…)                # Compute element contribution
    assemble!(f, fe, celldofs(face))        # Assemble into global RHS
end
```

(Available in the next Ferrite.jl release.)

FEM puzzle pieces: Linear system

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

```
# LU factorization
using LinearAlgebra
u = lu(K) \ f

# Cholesky factorization
u = cholesky(K) \ f

# Conjugate gradients (CG)
using IterativeSolvers
u = cg(K, f)
```

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.

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 is being worked on?

- Google Summer of Code 2023 project for Discontinuous Galerkin (DG) infrastructure *(Abdulaziz Hamid)*
- Interface elements *(David Rollin)*
- Distributed algebraic multigrid solver/preconditioner *(Tirtho Saha)*
- Mesh adaptivity *(Maximilian Köhler)*
- Distributed computing (MPI) *(Dennis Ogiermann)*
- Mesh free methods (GPU friendly) *(Dennis Ogiermann)*

Questions?