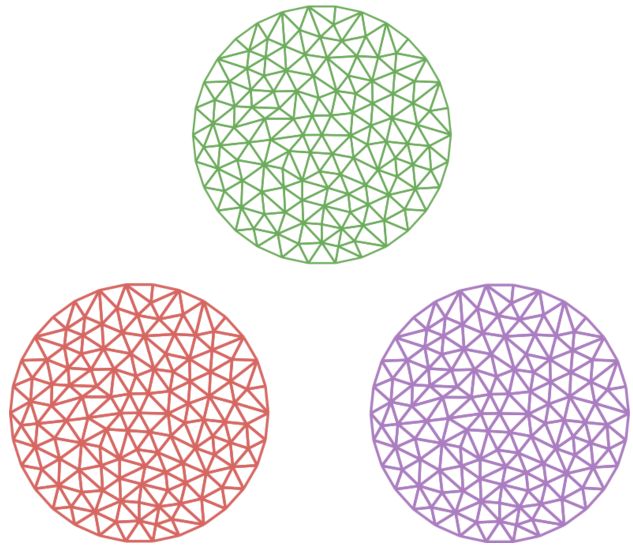
FEM in Julia

An overview of the package landscape for FEM in Julia



Kristoffer Carlsson

Division of Material and Computational Mechanics

Chalmers University of Technology

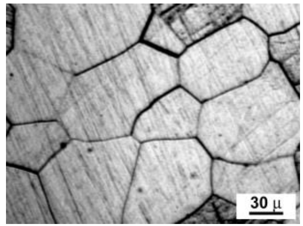
@KristofferC

kristoffer.carlsson@chalmers.se

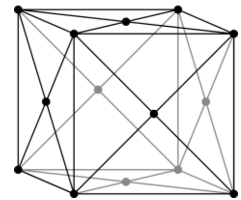
CHALMERS

Motivation

macro $l \approx 10^{-2}\mathrm{m}$



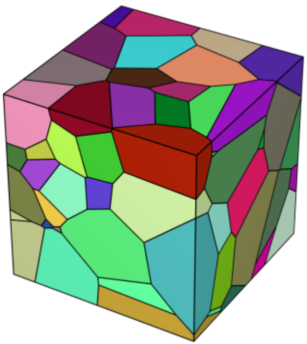
meso $l \approx 10^{-6}\mathrm{m}$

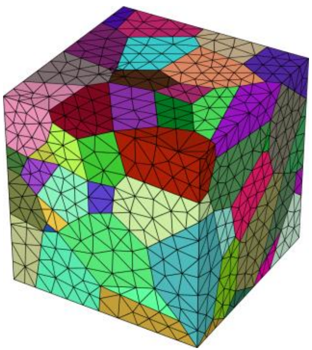


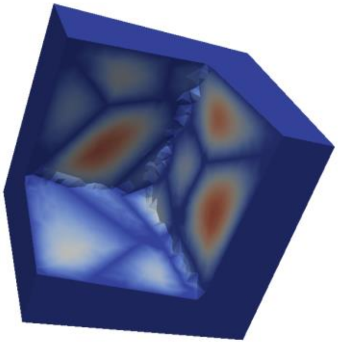
micro $l \approx 10^{-9}\mathrm{m}$

- Macroscopic homogeneous response is a volume average of heterogenities on a smaller scale
- Material models on macroscopic scale cannot predict new response to changes on the smaller scale

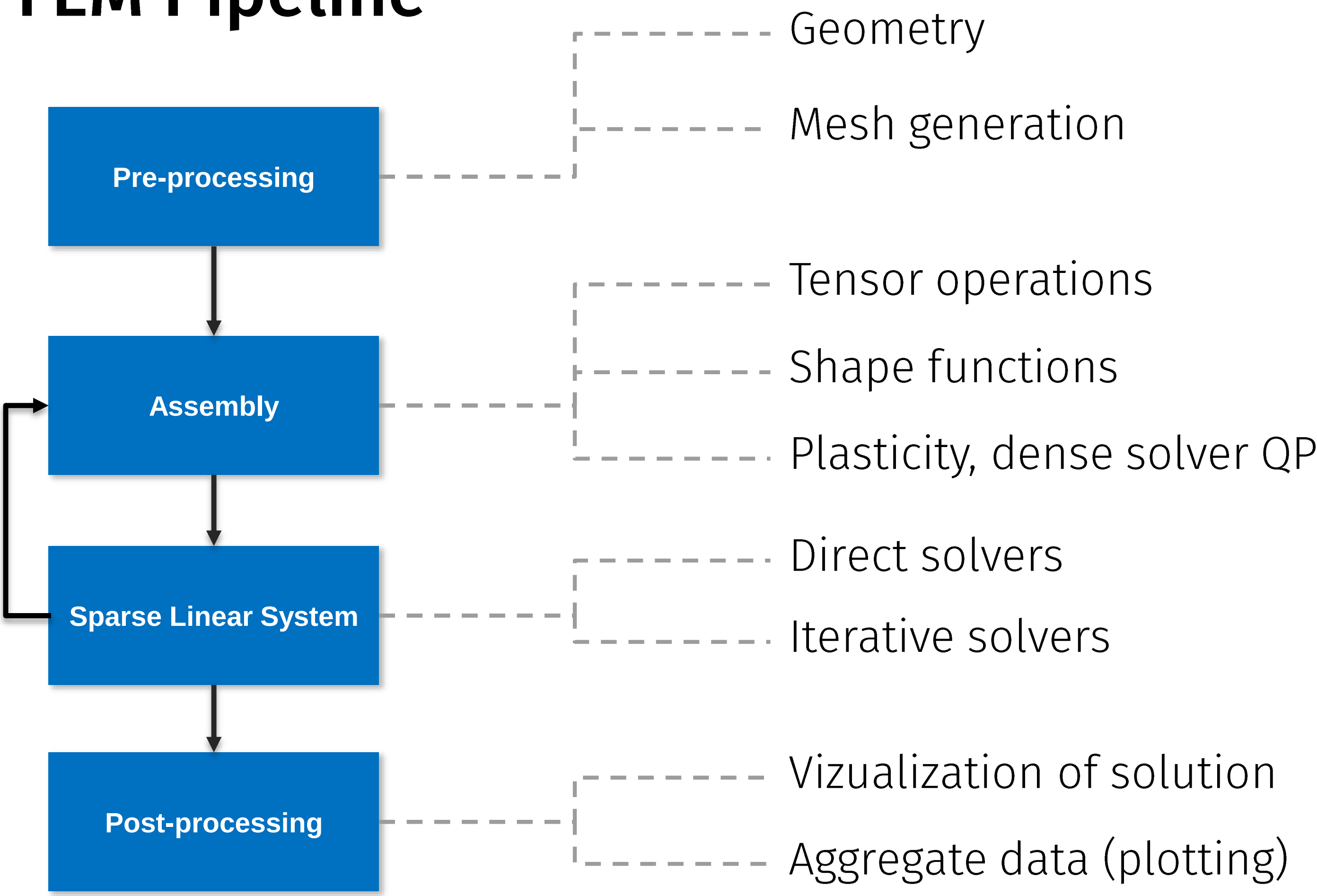
Motivation







FEM Pipeline



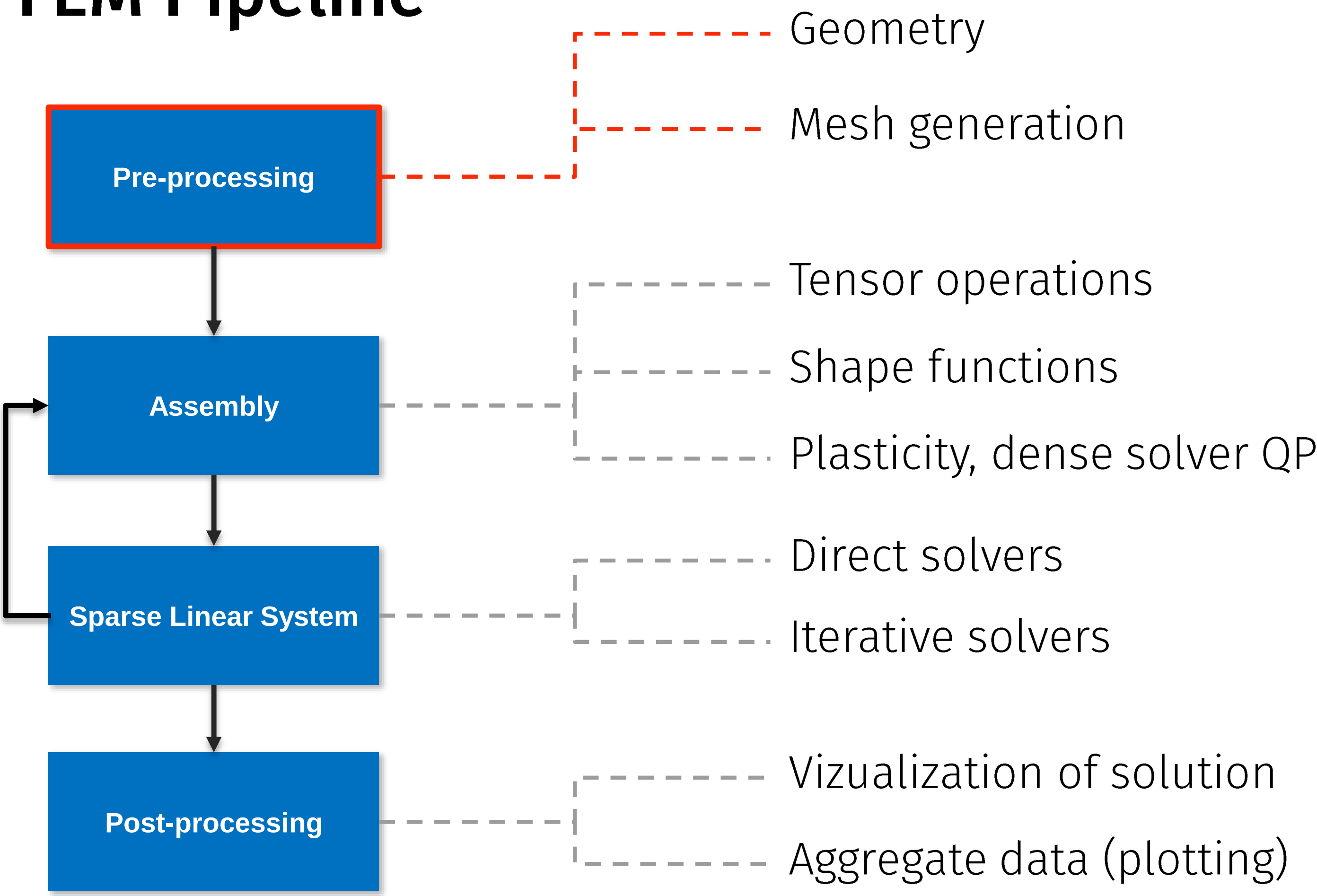
Full fledged Julia FEM package?

- A full FEM stack is a huge project
  - Deal.II – C++ - 30 000 commits – 20 years
  - DOLFIN – C++ - 21 000 commits – 10 years

- Julia public for ≈ 4 years

`JuliaFEM.jl - @ahojukka5` et al.
`EllipticFEM.jl - @gerhardtulzer`
`DifferentialEquations.jl -@ChrisRackauckas`

FEM Pipeline



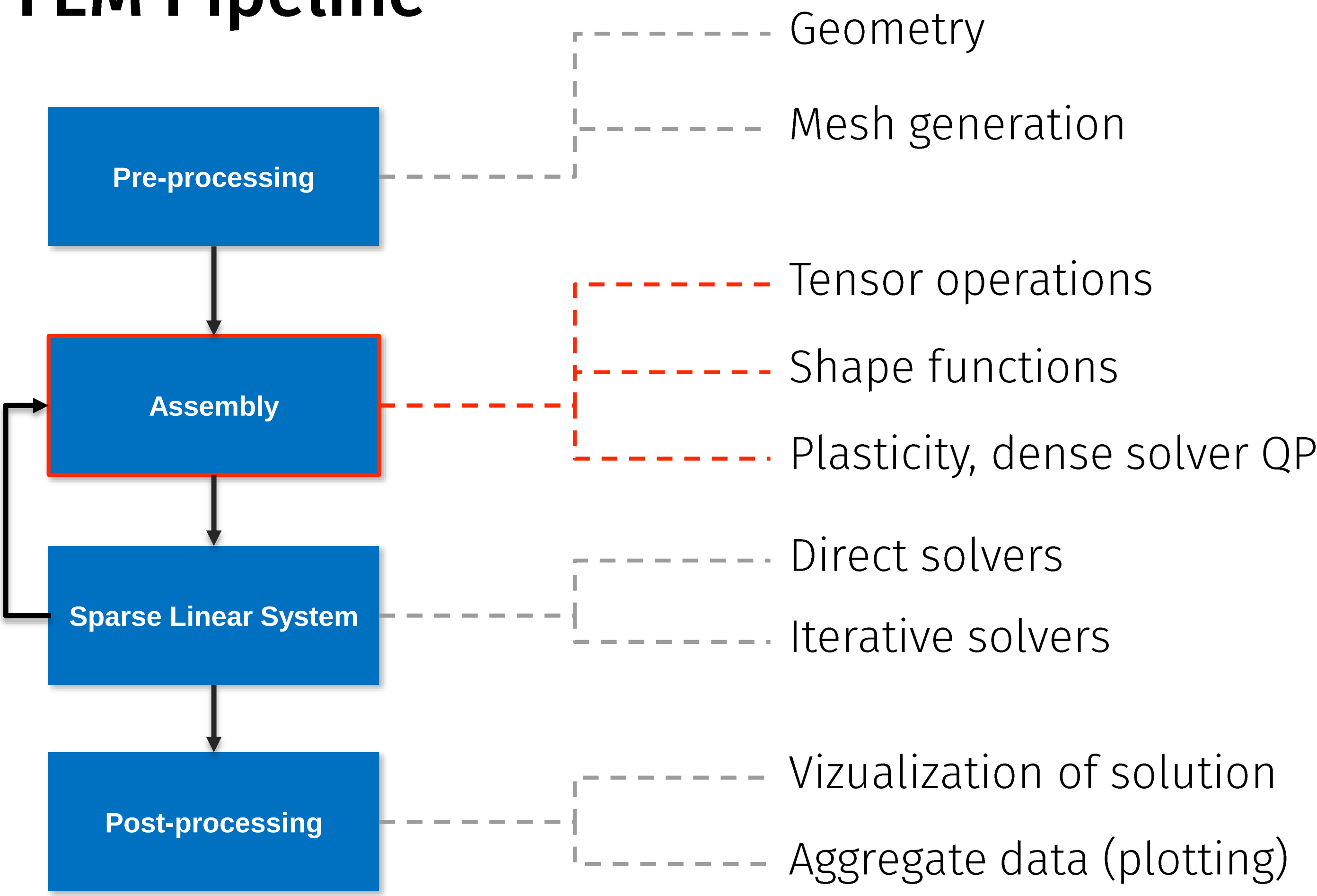
Geometry + Meshing

- Many existing free non-Julia tools for geometry + meshing. GMSH, netgen, tetgen etc.

- Works well, however, need to read the generated mesh files into Julia

- `Meshio.jl - @SimonDanisch`
  - Currently, mostly computer graphics mesh formats
  - I recently added ABAQUS .inp format (in a PR)
  - Call to action!

FEM Pipeline



Tensor operations

- Typically PDE’s are given as a collection of tensorial relations
- Working with ”naked” Julia arrays as a representation of tensors possible
- This is Julia! No fear of creating new types that maps more directly to our mathematical objects

Tensor operations

- Einsum.jl - @ahwillia

```
using Einsum
A = zeros(5,6,7)
X = randn(5,2)
Y = randn(6,2)
Z = randn(7,2)
@einsum A[i,j,k] = X[i,r]*Y[j,r]*Z[k,r]
```

- TensorOperations.jl – @Jutho

```
using TensorOperations
α=randn()
A=randn(5,5,5,5,5,5)
B=randn(5,5,5)
C=randn(5,5,5)
D=zeros(5,5,5)
@tensor begin
    D[a,b,c] = A[a,e,f,c,f,g]*B[g,b,e] + α*C[c,a,b]
    E[a,b,c] := A[a,e,f,c,f,g]*B[g,b,e] + α*C[c,a,b]
end
```

Tensor operations

- `ContMechTensors.jl - @KristofferC`
  - Size of tensors known at compile time (1D, 2D, 3D), loop unrolling benefitial
  - Symmetric tensors are common
  - Stack allocation + infix notation → beautiful code

```julia
using ContMechTensors
A = rand(SymmetricTensor{2, 2})
B = rand(Tensor{4, 2})
x = rand(Vec{2})

A ⋅ x # dot product
A ⊗ A # open product
B ⊡ A # double contraction

det(A), inv(A), trace(A)
```

  - Maps directly to assembly instructions, no SIMD yet.

Shape functions

- After variational formulation $f = \int_{\Omega_k} v(\boldsymbol{x}) f(\boldsymbol{x}) d\Omega_k$

- Discretization $v \approx v_h = \sum_{i=1}^{n} V_i \phi_i(\boldsymbol{x})$

- Quadrature $f_i = \sum_{q=1}^{n_{\mathrm{qp}}} \phi_i(\boldsymbol{x}_q) f(\boldsymbol{x}_q) w_q$

- Need to evaluate shape functions $\boldsymbol{\phi}$ at quadrature points defined by $\{\boldsymbol{x}_q, w_q\}_{q=1}^{n_{\mathrm{qp}}}$

Shape functions

- `JuAFEM.jl` — @KristofferC, @fredrikekre
  - Quadrature

```
julia> quadrule = QuadratureRule(:legendre, Dim{2}, RefTetrahedron(), 1);

julia> points(quadrule)
1-element Array{ContMechTensors.Tensor{1,2,Float64,2},1}:
 [0.333333,0.333333]

julia> weights(quadrule)
1-element Array{Float64,1}:
 0.5
```

  - `FastGaussQuadrature.jl` for cubes, tables for tetrahedrons.

Shape functions

- JuAFEM.jl — @KristofferC, @fredrikekre
    - Basis

```
julia> const dim = 2;

julia> basis_order =  1;

julia> basis = Lagrange{dim, RefTetrahedron, basis_order}();
```

Shape functions

- JuAFEM.jl — @KristofferC, @fredrikekre
  - FEValues = quadrature + basis

```
julia> fe_values = FEValues(quadrule, basis);

julia> ele_coords = [Vec{2}((0.0, 0.0)),
                     Vec{2}((1.0, 0.0)),
                     Vec{2}((1.0, 1.0))]

julia> reinit!(fe_values, ele_coords)

julia> q_point = 1;

julia> node = 2;

julia> shape_value(fe_values, q_point, node)
 0.3333333333333333

julia> shape_gradient(fe_values, q_point, node)
2-element ContMechTensors.Tensor{1,2,Float64,2}:
  1.0
 -1.0
```

BlockArrays

- Coupled/mixed problem leads to block like structures

$$R_p(\boldsymbol{u}, p; \delta p) = 0 \quad \forall \delta \boldsymbol{p}$$

$$R_u(\boldsymbol{u}, p; \delta u) = 0 \quad \forall \delta \boldsymbol{u}$$

$$\boldsymbol{f} = \left[\begin{array}{c} \boldsymbol{f_u} \\ \hline \boldsymbol{f_p} \end{array}\right] \qquad \boldsymbol{K} = \left[\begin{array}{c|c} \boldsymbol{K_{uu}} & \boldsymbol{K_{up}} \\ \hline \boldsymbol{K_{pu}} & \boldsymbol{K_{pp}} \end{array}\right]$$

- Desireable to have a Julia array type that represents this

BlockArrays

- `BlockArrays.jl` — `@KristofferC`, v0.5 only
- Proposes an `AbstractBlockArray` interface, extension to `AbstractArray`
- Two implementations of block arrays
  - `PseudoBlockArray` — whole matrix stored contiguously
  - `BlockArray` — each block stored contiguously

BlockArrays

```
julia> A = PseudoBlockArray(rand(3,3), [2,1], [2,1])
2×2-blocked 3×3 BlockArrays.PseudoBlockArray{Float64,2,Array{Float64,2}}:
 0.587037  0.443899  │  0.801079
 0.132292  0.196876  │  0.972342
 ────────────────────┼──────────
 0.800054  0.251887  │  0.78099

julia> A[Block(1, 2)]
2×1 Array{Float64,2}:
 0.801079
 0.972342

julia> A[Block(1, 2)] = [0, 0];

julia> A
2×2-blocked 3×3 BlockArrays.PseudoBlockArray{Float64,2,Array{Float64,2}}:
 0.587037  0.443899  │  0.0
 0.132292  0.196876  │  0.0
 ────────────────────┼──────────
 0.800054  0.251887  │  0.78099

julia> nblocks(A)
(2,2)

julia> blocksize(A, 1, 2)
(2,1)

julia> full(A); # returns the ”normal” matrix
```

Dense solvers

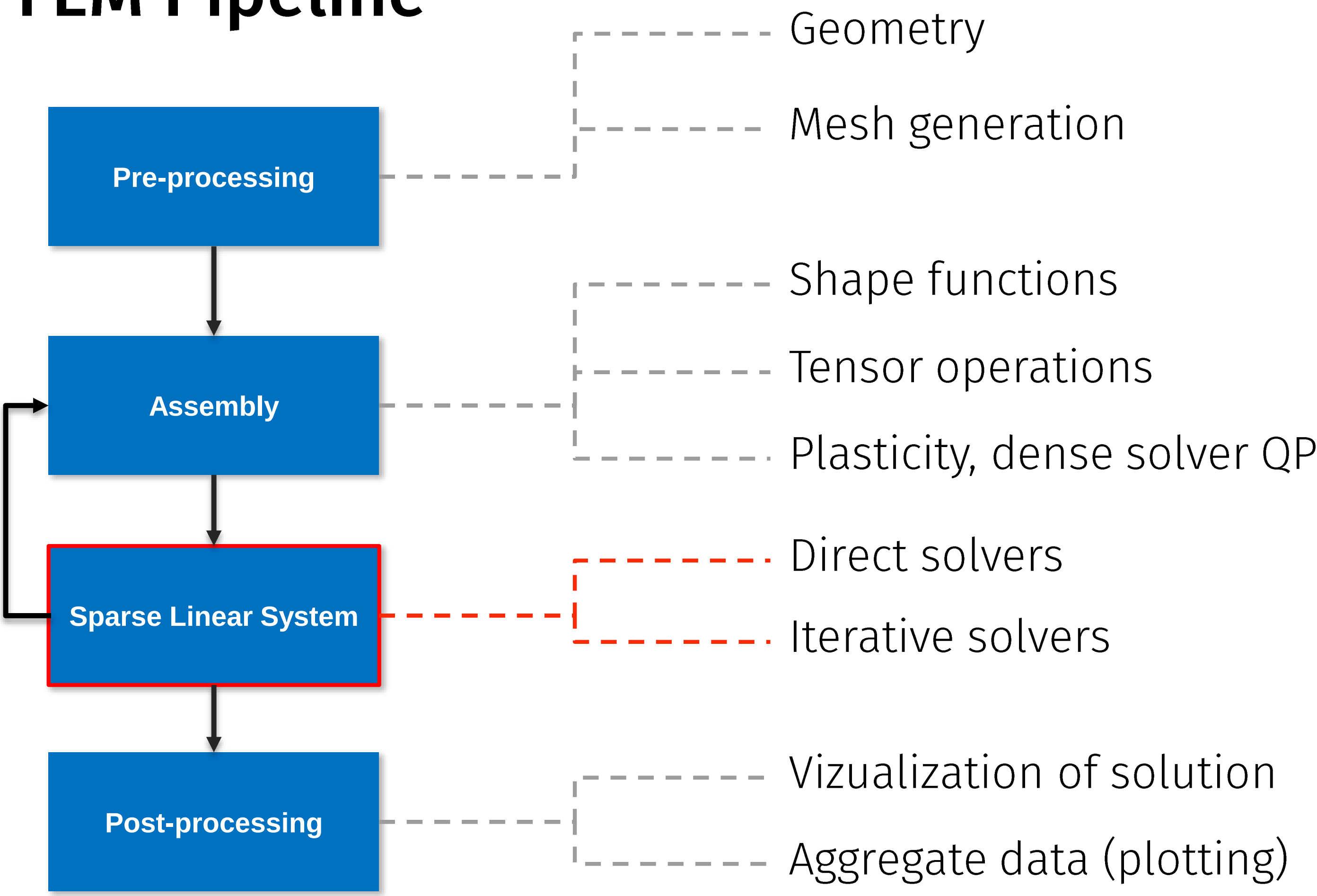
- LAPACK wrapped in Base Julia, 'nuff said?
- Plasticity → maximum dissipation principle → fulfill KKT condition in each quadrature point
- Need to solve small dense nonlinear system, $\approx 10^1$ number of unknowns
- 100 000 elements, 4 quadrature points → 400 000 systems to solve per Newton iteration
- Overhead in solver packages can dominate

```julia
function solve(f, x0)
    # allocate temporary arrays
    for i in iter
        do_step(f, x)
    end
    return solution
end
```

```julia
function solve(f, x0, cache=SolveCache(x0))
    # unpack cache (Parameters.jl, @mauro3)
    for i in iter
        do_step(f, x)
    end
    return solution
end
```

- `NLsolve.jl, Optim.jl + ForwardDiff.jl =` ❤️

FEM Pipeline

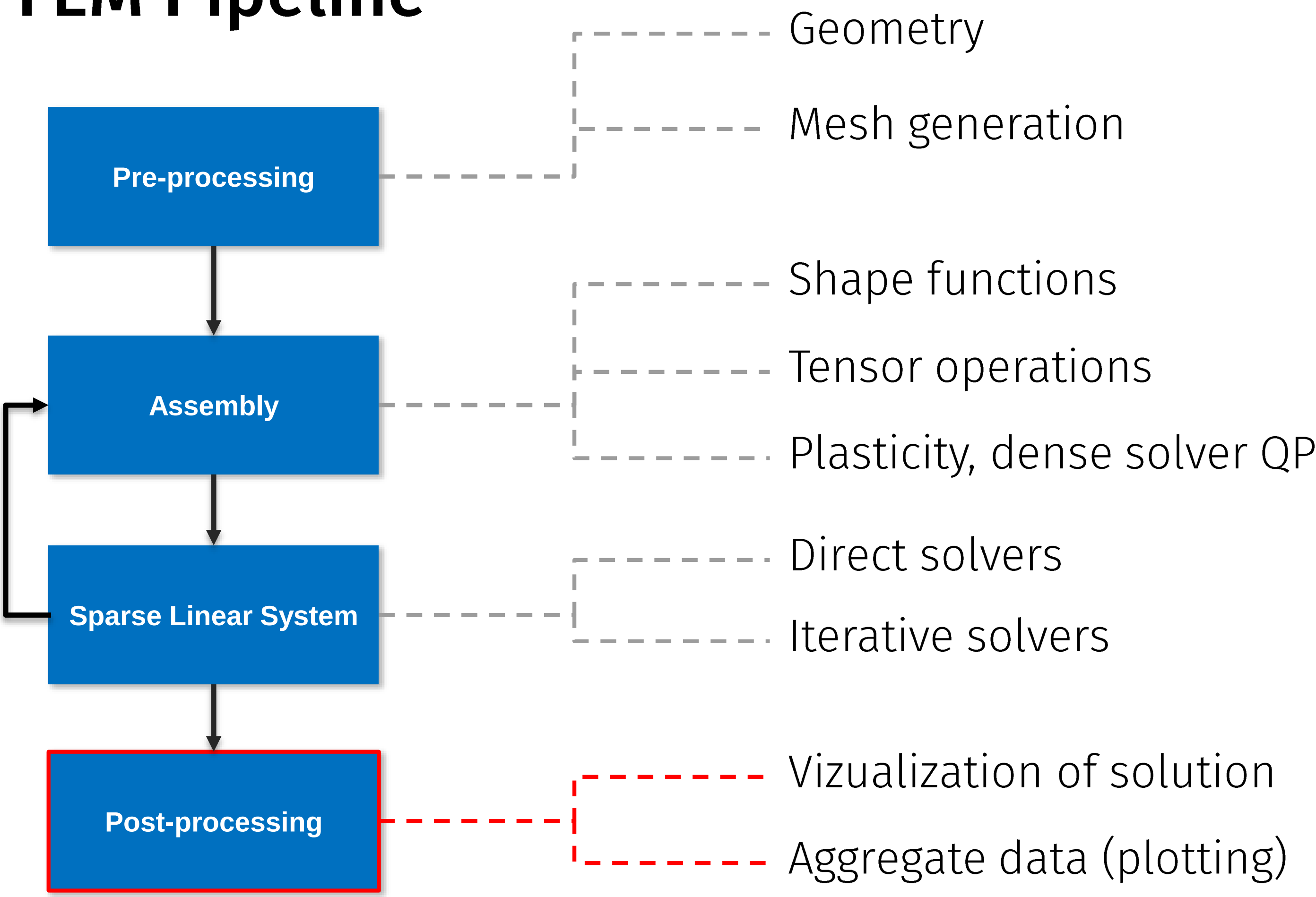


Sparse Linear Solvers

- Direct solvers
  - Cholmod + UMFPACK — Comes with Julia, "\"
  - `Pardiso.jl`, `Mumps.jl`, Wrappers to C libraries

- Iterative solvers
  - `IterativeSolvers.jl - @jiahao` et al.
  - `KrylovMethods.jl - @lruthotto`
  - `Krylov.jl - @dpo`
  - `PyAMG.jl - @cortner` — PyAMG wrapper with `PyCall`
  - See Lars Ruthotto: Iterative Methods for Sparse Linear Sy Julia: A Quick Overview
- PETSc.jl — Jared Crean — Talk on Friday

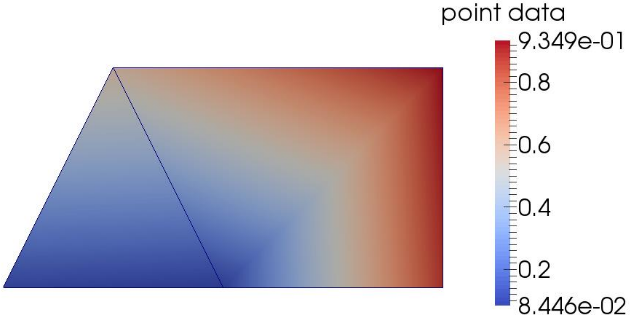
FEM Pipeline



Vizualization of solutions

- Dominating format: .VTK + Paraview/Visit
- `WriteVTK.jl - @jipolanco`

```julia
using WriteVTK
points = [0.0 0.5 2.0 1.0 2.0;
          0.0 1.0 0.0 0.0 1.0;
          0.0 0.0 0.0 0.0 0.0]
cells = [MeshCell(VTKCellType.VTK_TRIANGLE, [1, 4, 2]),
         MeshCell(VTKCellType.VTK_QUAD, [2, 4, 3, 5])]
vtkfile = vtk_grid("my_vtk_file", points, cells)
vtk_point_data(vtkfile, rand(5), "point data")
vtk_cell_data(vtkfile, rand(2), "cell data")
vtk_save(vtkfile)
```



Aggregate data (plotting)

- Plotting landcape in Julia used to be quite fragmented
- Thanks to `@tbreloff` and `Plots.jl` this is no longer the case
- `Dataframes.jl` + `JLD.jl` + `Plots.jl` (`PGFPlots.jl` backend for publications)

```
using JLD
using Dataframes
using Plots
pgfplots()

df = load("data.jld")[”analysis_data"]
data = extract_data(df)
p = plot(data)
PGFPlots.save("pgfplot.tex", p.o, include_preamble=false)
```

Kristoffer's wishlist

- Make working with stack allocated arrays easier
  - `Array` – powerful but big and heap allocated
  - Goto solution is tuples
  - Generated code explosions, type inference problems, immutability not always desired
  - See issue #11902

- Facilitate not double paying lookup cost when mutating `LinearSlow` elements
  - `K[i,j] += 1.0` → `K[i,j] = K[i,j] + 1.0`
  - `K` sparse → `[i,j]` expensive, double paying
  - See issue #15630 and `https://github.com/KristofferC/UpdateIndex.jl`

Conclusions

- Package landscape for FEM is developed well enough to do actual work
- State of the art Automatic Differentiation makes local stiffness tangents a breeze to evaluate
- Possible to use Julia for only a part of the FEM stack by calling into Julia from C++ FEM libraries

Thank you!

Slides

`https://github.com/KristofferC/JuliaCon_FEM`

Acknowledgements

Swedish Research Council

Vetenskapsrådet

Julia Computing

Julia computing