Tensors.jl

Efficient tensor computations with support for automatic differentiation

FerriteCon 2025

Kristoffer Carlsson
JuliaHub

Tensors in physics and engineering

Electromagnetism

$$\mathbf{D} = \boldsymbol{\varepsilon} \cdot \mathbf{E}$$

- $\boldsymbol{\varepsilon}$: Permittivity tensor (rank 2).

Inertia and rotation

$$\mathbf{L} = \mathbf{I} \cdot \boldsymbol{\omega}$$

- $\mathbf{I}$: Moment of inertia tensor (rank 2).

Stress–strain relations

$$\boldsymbol{\sigma} = \mathbb{C} : \boldsymbol{\varepsilon}$$

- $\mathbb{C}$: Stiffness tensor (rank 4).
- $\boldsymbol{\varepsilon}$: Strain tensor (rank 2).
- : Double contraction ($\mathbb{A} : \mathbf{b} = A_{ijkl}b_{kl}$).

From continuum to finite elements

Weak form of balance of momentum

$$\int_{\Omega} \nabla \mathbf{u} : \boldsymbol{\sigma} \, d\Omega = \int_{\Omega} \mathbf{u} \cdot \mathbf{b} \, d\Omega + \int_{\partial\Omega} \mathbf{u} \cdot \mathbf{t} \, d\Gamma$$

Finite element discretization – linear elasticity

Shape function approximation:

$$\mathbf{u} = \sum_{i} \mathbf{N}_i u_i$$

“Stiffness matrix”:

$$K_{ij} = \int_{\Omega} \nabla \mathbf{N}_i : \mathbb{C} : \nabla \mathbf{N}_j \, d\Omega$$

Where:

- $\nabla \mathbf{N}_i$: Shape function gradient (tensor)
- $\mathbb{C}$: Fourth-order elasticity tensor

FE assembly implementation

- Stiffness assembly for one element:

```
function assemble_stiffness!(K, C)
    for (w, ξ) in quadrature_rule
        ∇N = shape_gradients(ξ)
        dΩ = det(jacobian(ξ)) * w
        for i in 1:n_basefuncs, j in 1:n_basefuncs
            K[i,j] += (∇N[i] : C : ∇N[j]) * dΩ
        end
    end
end
```

Questions

- How should we store `∇N[i]` and `C` (possibly symmetric)?
- How should we compute `C : ∇N[j]` and other tensor operatoions?

Voigt format – storage

A technique to embed higher-order (possibly symmetric) tensors into standard linear algebra:

- **Rank 2 tensors → Vectors**
  - General: 9 components (3D), 4 components (2D)

$$\overline{\mathbf{F}} = [F_{11}, F_{22}, F_{12}, F_{21}]$$

  - Symmetric: 6 components (3D), 3 components (2D)

$$\overline{\boldsymbol{\sigma}} = [\sigma_{11}, \sigma_{22}, \sigma_{12}]$$

- **Rank 4 tensors → Matrices**
  - General: 9×9 (3D), 4×4 (2D) : $\mathbb{D}_{ijkl} \to \begin{bmatrix} D_{1111} & D_{1122} & D_{1112} & D_{1121} \\ D_{2211} & D_{2222} & D_{2212} & D_{2221} \\ D_{1211} & D_{1222} & D_{1212} & D_{1221} \\ D_{2111} & D_{2122} & D_{2112} & D_{2121} \end{bmatrix}$
  - Symmetric: 6×6 (3D), 3×3 (2D): $\mathbb{C}_{ijkl} \to \begin{bmatrix} C_{1111} & C_{1122} & C_{1112} \\ C_{2211} & C_{2222} & C_{2212} \\ C_{1211} & C_{1222} & C_{1212} \end{bmatrix}$

Voigt format – operations

- **Double contraction (rank 4 and rank 2) → Matrix–vector product**

$$\mathbb{C} : \nabla \mathbf{N}_j \quad \rightarrow \quad \mathbf{D}\mathbf{b}_j$$

- **Double contraction (rank 2 and rank 2) → dot product**

$$\mathbf{S} : \mathbf{E} = \overline{S}^\top \overline{E}$$

Voigt format – scaling off-diagonals

"Engineering strain" (different representation for stress and strain in symmetric tensors):

- $\overline{\boldsymbol{\sigma}} = [\sigma_{11}, \sigma_{22}, \sigma_{12}]$
- $\overline{\boldsymbol{\varepsilon}} = [\varepsilon_{11}, \varepsilon_{22}, 2\varepsilon_{12}]$
- $\boldsymbol{\sigma} : \boldsymbol{\varepsilon} = \overline{\boldsymbol{\sigma}}^T \overline{\boldsymbol{\varepsilon}}$ still gives the correct energy.
- Mandel notation uses $\sqrt{2}$ factor on both stress and strain.

Voigt format in FEM

From tensor loops to matrix operations

```
for i in 1:n_basefuncs, j in 1:n_basefuncs
    K[i,j] += (∇N[i] : C : ∇N[j]) * dΩ
end
```

becomes

```
for i in 1:n_basefuncs, j in 1:n_basefuncs
    K[i,j] += (b_i' * D * b_j) * dΩ
end
```

where `b_i = voigt(∇N[i])`.

Voigt format in FEM

The "B-matrix"

Each `b_i` becomes a **column** in the $\mathbf{B}$ matrix:

$$\mathbf{B} = [\mathbf{b}_1 \quad \mathbf{b}_2 \quad \cdots \quad \mathbf{b}_n]$$

This yields the compact form: $K = \mathbf{B}^\top \mathbf{D} \mathbf{B}$

$$\mathbf{B}^\top = \begin{bmatrix} \mathbf{b}_1^\top \\ \mathbf{b}_2^\top \\ \vdots \\ \mathbf{b}_n^\top \end{bmatrix}, \qquad \mathbf{D}\mathbf{B} = [\mathbf{D}\mathbf{b}_1 \quad \mathbf{D}\mathbf{b}_2 \quad \cdots \quad \mathbf{D}\mathbf{b}_n]$$

$$\mathbf{B}^\top \mathbf{D} \mathbf{B} = \begin{bmatrix} \mathbf{b}_1^\top \mathbf{D}\mathbf{b}_1 & \mathbf{b}_1^\top \mathbf{D}\mathbf{b}_2 & \cdots & \mathbf{b}_1^\top \mathbf{D}\mathbf{b}_n \\ \mathbf{b}_2^\top \mathbf{D}\mathbf{b}_1 & \mathbf{b}_2^\top \mathbf{D}\mathbf{b}_2 & \cdots & \mathbf{b}_2^\top \mathbf{D}\mathbf{b}_n \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{b}_n^\top \mathbf{D}\mathbf{b}_1 & \mathbf{b}_n^\top \mathbf{D}\mathbf{b}_2 & \cdots & \mathbf{b}_n^\top \mathbf{D}\mathbf{b}_n \end{bmatrix}$$

Each element: $K_{ij} = (\mathbf{B}^\top \mathbf{D} \mathbf{B})_{ij} = \mathbf{b}_i^\top \mathbf{D} \mathbf{b}_j$

Drawbacks of Voigt

Based on personal experience teaching (and using) FEM with Voigt:

- Easy to forget scaling factors for shear terms for strains.
- The “B-matrix” becomes a somewhat magical object that loses correspondence to the FEM formulation.
- Generic linear algebra operations are slow for such small sizes.

Tensors.jl

Code:: https://github.com/Ferrite-FEM/Tensors.jl

Paper: Carlsson, K. & Ekre, F. (2019). *Tensors.jl — Tensor Computations in Julia*. Journal of Open Research Software, 7:7

Some history

- Feb 2016: Created as *ContMechTensors.jl* (commit).
- Intended as an alternative to "raw" Voigt form for FEM codes.
- Mar 2016: Integrated into Ferrite (then JuAFEM) (PR #51).
  - Made Ferrite code more similar to the mathematical description.
  - Made the code less mutating, better performance:

```diff
- @inline function function_scalar_gradient!{dim, T}(grad::Vector{T}, ...
+ @inline function function_scalar_gradient{dim, T}(...)
```

  - Okay, maybe that's another issue then, that only we are using it? ;) We need to spread the word!

Basic usage – Creating tensors

```
julia> v = rand(Vec{2})
2-element Vec{2, Float64}:
 0.4518004270728473
 0.9514979486051207

# [Symmetric]Tensor{order, dim, T}
julia> S = SymmetricTensor{2,2,Float64}((i,j) -> i + j)
2×2 SymmetricTensor{2, 2, Float64, 3}:
 2.0  3.0
 3.0  4.0

julia> sizeof(S)  # only symmetric part stored
24

julia> one(Tensor{4, 2})
2×2×2×2 Tensor{4, 2, Float64, 16}:
[:, :, 1, 1] =
 1.0  0.0
 0.0  0.0
...
```

Basic usage – Basic operations

Dot product (single contraction)

$$\mathbf{a} = \mathbf{B} \cdot \mathbf{c} \quad \Leftrightarrow \quad a_i = B_{ij} c_j$$

```
julia> B = rand(Tensor{2,2}); c = rand(Vec{2});

julia> a = B ⋅ c  # or dot(B, c)
2-element Vec{2, Float64}:
 0.2973081283150573
 0.5776654547151459
```

Double contraction

$$\mathbf{A} = \mathsf{C} : \mathbf{B} \quad \Leftrightarrow \quad A_{ij} = C_{ijkl} B_{kl}$$

```
julia> C = rand(SymmetricTensor{4,2}); B = rand(SymmetricTensor{2,2});

julia> A = C ⊡ B  # or dcontract(C, B)
2×2 SymmetricTensor{2, 2, Float64, 3}:
 1.30202  1.11747
 1.11747  0.50486
```

- Symmetry is preserved in the type.

Basic usage – More operations

Tensor product (outer product)

$$\mathbf{A} = \mathbf{b} \otimes \mathbf{c} \quad \Leftrightarrow \quad A_{ij} = b_i c_j, \quad \mathbb{D} = \mathbf{B} \otimes \mathbf{C} \quad \Leftrightarrow \quad D_{ijkl} = B_{ij} C_{kl}$$

```
julia> b = rand(Vec{2}); c = rand(Vec{2});

julia> A = b ⊗ c  # or otimes(b, c)
2×2 Tensor{2, 2, Float64, 4}:
 0.620584  0.331023
 0.270906  0.144503
```

- `otimesu(A, B)` : "Upper" product $A_{ik} B_{jl}$
- `otimesl(A, B)` : "Lower" product $A_{il} B_{jk}$

Norm, trace, determinant

$$\|\mathbf{A}\| = \sqrt{\mathbf{A} : \mathbf{A}}, \quad \mathrm{tr}(\mathbf{A}) = A_{ii}, \quad \det(\mathbf{A})$$

```
julia> A = rand(SymmetricTensor{2,2});

julia> norm(A), tr(A), det(A)
(0.5889762248690359, 0.5098257705880324, -0.043485338552650056)
```

Basic usage – Additional operations

Transpose and symmetry operations

- `transpose(A)` : $A_{ij}^T = A_{ji}$
- `symmetric(A)` : $A^{\text{sym}} = \frac{1}{2}(A + A^T)$
- `skew(A)` : $A^{\text{skw}} = \frac{1}{2}(A - A^T)$

Tensor decompositions

- `dev(A)` : Deviatoric part $A^{\text{dev}} = A - \frac{1}{3}\text{tr}(A)I$
- `vol(A)` : Volumetric part $A^{\text{vol}} = \frac{1}{3}\text{tr}(A)I$

Other operations

- `inv(A)` : Matrix inverse
- `sqrt(A)` : Tensor square root (for symmetric positive definite)
- `eigen(A)` : Eigenvalues/eigenvectors
- `tdot(F)` : $F^\top \cdot F$ (transpose dot, returns symmetric tensor)

Basic usage – performance

- Operations are specialized on size and element type
- Uses tuples internally for non-allocating operations

```
julia> t = rand(Tensor{4,3});

julia> length(t)
81

julia> @btime $t + $t
  11.721 ns (0 allocations: 0 bytes)
3×3×3×3 Tensor{4, 3, Float64, 81}:
[:, :, 1, 1] =
 1.73626   1.92998  1.34596
 1.12944   1.83721  0.1809
 0.278972  1.39517  0.0216357
```

- Uses SIMD instructions (more on that later)

```
julia> @code_llvm debuginfo=:none t+t
...
  %21 = load <4 x double>, ptr ...
  %22 = load <4 x double>, ptr ...
  %23 = fadd <4 x double> %21, %22
...
```

Storage format

- Started with just wrapping arrays (basically Voigt under the hood):

```
immutable Tensor{order, dim, T <: Number, M} <: AbstractTensor{order, dim, T, M}
   data::Array{T, M}
end
```

→ All operations allocated memory.

- Quickly moved to tuples (julia just got good support for tuples). Made operations specialize on tuple size and avoid allocations:

```
immutable Tensor{order, dim, T <: Real, M} <: AbstractTensor{order, dim, T}
  data::NTuple{M, T}
end
```

- Tried wrapping StaticArrays. Removed later since it provided little benefit:

```
immutable Tensor{order, dim, T <: Real, M} <: AbstractTensor{order, dim, T}
  data::SVector{M, T}
end
```

Automatic differentiation

- `Tensor` could be used inside functions that were automatically differentiated.
- But one could not directly use AD on tensor functions to return tensors.

Function Types and Gradients

Input Type	Output Type	Gradient Type	Hessian Type	Mathematical Form
Vec	Scalar	Vec	Tensor{2}	$\nabla f = \frac{\partial f}{\partial \mathbf{x}}, \quad \mathbf{H} = \frac{\partial^2 f}{\partial \mathbf{x} \partial \mathbf{x}}$
Tensor{2}	Scalar	Tensor{2}	Tensor{4}	$\frac{\partial f}{\partial \mathbf{A}}, \quad \frac{\partial^2 f}{\partial \mathbf{A} \partial \mathbf{A}}$
Vec	Vec	Tensor{2}	–	$\mathbf{J} = \frac{\partial \mathbf{f}}{\partial \mathbf{x}}$
Tensor{2}	Tensor{2}	Tensor{4}	–	$\frac{\partial \mathbf{F}}{\partial \mathbf{A}}$

- Support for `gradient` and `hessian`: Dec 10, 2016
- Support for `curl`, `divergence`, and `laplace`: Sep 27, 2017 (Fredrik Ekre)

Automatic differentiation – examples

Norm of a vector

$$f(\mathbf{x}) = \|\mathbf{x}\| \quad \Rightarrow \quad \frac{\partial f}{\partial \mathbf{x}} = \frac{\mathbf{x}}{\|\mathbf{x}\|}$$

```
julia> x = rand(Vec{2});

julia> gradient(norm, x)
2-element Vec{2, Float64}:
 0.5105128363207563
 0.859870132026771

julia> x / norm(x)  # analytical solution
2-element Vec{2, Float64}:
 0.5105128363207563
 0.8598701320267711
```

Automatic differentiation – examples

Determinant of a symmetric tensor

$$f(\mathbf{A}) = \det \mathbf{A} \quad \Rightarrow \quad \frac{\partial f}{\partial \mathbf{A}} = \det(\mathbf{A})\, \mathbf{A}^{-T}$$

```
julia> A = rand(SymmetricTensor{2,2});

julia> gradient(det, A)
2×2 SymmetricTensor{2, 2, Float64, 3}:
  0.218587  -0.549051
 -0.549051   0.325977

julia> inv(A)' * det(A)  # analytical: A^(-T) * det(A)
2×2 SymmetricTensor{2, 2, Float64, 3}:
  0.218587  -0.549051
 -0.549051   0.325977
```

Automatic differentiation – examples

Hessian of a quadratic potential

$$\psi(\mathbf{e}) = \frac{1}{2}\,\mathbf{e} : \mathsf{E} : \mathbf{e} \quad \Rightarrow \quad \frac{\partial^2\psi}{\partial\mathbf{e}\otimes\partial\mathbf{e}} = \mathsf{E}^{\mathrm{sym}}$$

where $\mathsf{E}^{\mathrm{sym}}$ is the major-symmetric part of E.

```
julia> const E = rand(SymmetricTensor{4,3});

julia> ψ(ϵ) = 1/2 * ϵ ⊡ E ⊡ ϵ;

julia> ϵ = rand(SymmetricTensor{2,3});

julia> E = @btime hessian(ψ, $ϵ)
  317.441 ns (0 allocations: 0 bytes)
3×3×3×3 SymmetricTensor{4, 3, Float64, 36}:
[:, :, 1, 1] =
 0.26313   0.57441   0.337005

julia> norm(majorsymmetric(E) - E)
0.0
```

Implementing custom gradients

If the function is a "black box" (maybe calls into C) or have an analytical form that is much more efficient than the automatic AD it can be added with:

```
@implement_gradient(f, f_dfdx)
```

- `f_dfdx` must return both the value and the gradient: `fval, dfdx_val = f_dfdx(x)`
- Called automatically when `f` is used in AD contexts
- Added by Knut Andreas Meyer in Jan 26, 2022

Implementing custom gradients

```
# Define functions
h(x) = norm(x)
f(x) = x ⋅ x

# Composed functions
cfun(x) = h(f(dev(x)))

# Define analytical derivative for f: ∂(A⋅A)/∂A = A⊗I + I⊗A
function dfdx(x::Tensor{2,dim}) where {dim}
    println("Calling analytical gradient")
    I2 = one(Tensor{2,dim})
    dfdx_val = otimesu(I2, transpose(x)) + otimesu(x, I2)
    return f(x), dfdx_val
end

# Register the custom gradient
@implement_gradient f dfdx

x = rand(Tensor{2, 2})

julia> gradient(cfun, x)
Calling analytical gradient
2×2 Tensor{2, 2, Float64, 4}:
 0.432701   0.450401
 0.546796  -0.0714608
```

Automatic Differentiation – Implementation

- Uses `Dual` numbers from `ForwardDiff.jl`.
- Insert seeded partials into the tensor, call the function, and extract the result into the corresponding tensor type:

$$A_{dual} = A + \sum_i e_i \epsilon_i, \quad f(A_{dual}) = f(A) + \sum_i \frac{\partial f}{\partial A_i} \epsilon_i$$

```
julia> A = rand(Tensor{2,2});

julia> Tensors.gradient(det, A)
2×2 Tensor{2, 2, Float64, 4}:
  0.791411  -0.234868
 -0.524795   0.447615

julia> A_dual = Tensors._load(A, nothing)
2×2 Tensor{2, 2, ForwardDiff.Dual{Nothing, Float64, 4}, 4}:
 Dual{Nothing}(0.447615,1.0,0.0,0.0,0.0)  Dual{Nothing}(0.524795,0.0,0.0,1.0,0.0)
 Dual{Nothing}(0.234868,0.0,1.0,0.0,0.0)  Dual{Nothing}(0.791411,0.0,0.0,0.0,1.0)

julia> det_dual = det(A_dual)
Dual{Nothing}(0.2309897180432929,0.7914111588502826,-0.52479490941829,-0.23486806142451777,0.4476147159441781)

julia> Tensors._extract_gradient(det_dual, A_dual)
2×2 Tensor{2, 2, Float64, 4}:
  0.791411  -0.234868
 -0.524795   0.447615
```

Automatic Differentiation - Implementation

- Actual code...:

```
@inline function _extract_gradient(v::Tensor{2, 3, <: Dual}, ::Tensor{2, 3})
    @inbounds begin
        p1, p2, p3 = partials(v[1,1]), partials(v[2,1]), partials(v[3,1])
        p4, p5, p6 = partials(v[1,2]), partials(v[2,2]), partials(v[3,2])
        p7, p8, p9 = partials(v[1,3]), partials(v[2,3]), partials(v[3,3])
        ∇f = Tensor{4, 3}((p1[1], p2[1], p3[1], p4[1], p5[1], p6[1], p7[1], p8[1], p9[1],
                           p1[2], p2[2], p3[2], p4[2], p5[2], p6[2], p7[2], p8[2], p9[2],  #    ###  #
                           p1[3], p2[3], p3[3], p4[3], p5[3], p6[3], p7[3], p8[3], p9[3],  #    # #  #
                           p1[4], p2[4], p3[4], p4[4], p5[4], p6[4], p7[4], p8[4], p9[4],  ###  ###  ###
                           p1[5], p2[5], p3[5], p4[5], p5[5], p6[5], p7[5], p8[5], p9[5],
                           p1[6], p2[6], p3[6], p4[6], p5[6], p6[6], p7[6], p8[6], p9[6],
                           p1[7], p2[7], p3[7], p4[7], p5[7], p6[7], p7[7], p8[7], p9[7],
                           p1[8], p2[8], p3[8], p4[8], p5[8], p6[8], p7[8], p8[8], p9[8],
                           p1[9], p2[9], p3[9], p4[9], p5[9], p6[9], p7[9], p8[9], p9[9]))
```

Automatic Differentiation -- Implementation

Implementation of Hessian is quite aesthetically pleasing.

```
function hessian(f::F, v::Union{SecondOrderTensor, Vec, Number}) where {F}
    gradf = y -> gradient(f, y)
    return gradient(gradf, v)
end
```

Automatic Differentiation -- Special handling for symmetric tensors

- For symmetric tensors, without special consideration for off-diagonals, a perturbation of off-diagonals give double contribution compared to diagonal entries.
- $\boldsymbol{\sigma} : \boldsymbol{\sigma} = \sigma_{11}^2 + \sigma_{22}^2 + 2\sigma_{12}^2.$
- Need to compensate by giving half weight to duals for off diagonals

```
SymmetricTensor{2,2}((Dual(data[1], 1, 0, 0),      # (1,1) component
                      Dual(data[2], 0, 1/2, 0),    # (1,2) component
                      Dual(data[3], 0, 0, 1)))     # (2,2) component
```

Explicit SIMD

- Added by Fredrik Ekre in Mar 16, 2017.
- Uses SIMD.jl
- SLP vectorizer optimizer pass (to turn tuple operations into SIMD) used to be enabled in Julia only with `-O3`.
- The SLP vectorizer in LLVM was not that great at that time, explicit SIMD often gave decent speedups.

Double contraction $\mathbb{C} : \boldsymbol{\varepsilon} = C_{ijkl}\varepsilon_{kl} = \boldsymbol{\sigma}_{ij}$ (4th-order with 2nd-order tensor):

```
function dcontract(S1::Tensor{4, 2, T}, S2::Tensor{2, 2, T}) where {T <: SIMDTypes}
    D1 = get_data(S1); D2 = get_data(S2)
    # Load 4th-order tensor slices: C[ij,:,:] for each (i,j)
    SV11 = tosimd(D1, Val{1},  Val{4})    # C[1111, 1112, 1121, 1122]
    SV12 = tosimd(D1, Val{5},  Val{8})    # C[1211, 1212, 1221, 1222]
    SV13 = tosimd(D1, Val{9},  Val{12})   # C[2111, 2112, 2121, 2122]
    SV14 = tosimd(D1, Val{13}, Val{16})   # C[2211, 2212, 2221, 2222]

    # Computes all 4 σ components simultaneously:
    # σ[11], σ[12], σ[21], σ[22] = each SV_ij dotted with [ε11, ε12, ε21, ε22]
    r = muladd(SV14, D2[4], muladd(SV13, D2[3], muladd(SV12, D2[2], SV11 * D2[1])))
    return Tensor{2, 2}(r)  # r = [σ11, σ12, σ21, σ22] as SIMD vector
end
```

- `simd.jl` file

Explicit SIMD needed in modern Julia?

- May no longer be needed (SLP is good enough?).
- PR to remove: https://github.com/Ferrite-FEM/Tensors.jl/pull/93
  - "Probably not needed with llvm6 and SLP vectorization enabled by default"
- Julia now uses LLVM ≥ 15...

Tensors2.jl or a refresh of Tensors.jl?

- Most of Tensors.jl was written when Julia's compiler (and LLVM) was much less capable (almost a decade ago!).
- (Authors of the code were also much less capable Julia programmers)
- Lots of code repetition, code generation, `@generated`, `@inline`, `@pure`, etc.
- Today it should be possible to write a significantly smaller Tensors.jl with the same performance and capability.
- With better abstractions and design, things like third-order tensors and mixed-order tensors might "just work".

On the other hand:

- Tensors.jl already works quite well.
- Good performance, few bugs, decent latency.
- A rewrite solely for the sake of rewriting is rarely an efficient use of time.

Thank You!

Questions?