Tensors.jl — Tensor Computations in Julia

Journal of
open research software

Carlsson, K and Ekre, F 2019 Tensors.jl — Tensor Computations in Julia. *Journal of Open Research Software,* 7: 7. DOI: https://doi.org/10.5334/jors.182

SOFTWARE METAPAPER

Tensors.jl — Tensor Computations in Julia

Kristoffer Carlsson and Fredrik Ekre

Department of Industrial and Materials Science, Chalmers University of Technology, Gothenburg, SE

Corresponding author: Kristoffer Carlsson (kristoffer.carlsson@chalmers.se)

Tensors.jl is a Julia package that provides efficient computations with symmetric and non-symmetric tensors. The focus is on the kind of tensors commonly used in e.g. continuum mechanics and fluid dynamics. Exploiting Julia's ability to overload Unicode infix operators and using Unicode in identifiers, implemented tensor expressions commonly look very similar to their mathematical writing. This possibly reduces the number of bugs in implementations. Operations on tensors are often compiled into the minimum assembly instructions required, and, when beneficial, SIMD-instructions are used. Computations involving symmetric tensors take symmetry into account to reduce computational cost. Automatic differentiation is supported, which means that most functions written in pure Julia can be efficiently differentiated without having to implement the derivative by hand. The package is useful in applications where efficient tensor operations are required, e.g. in the Finite Element Method.

Keywords: tensors; continuum mechanics; fluid dynamics; constitutive modeling; finite element method
Funding statement: Support for this research was provided by the Swedish Research Council (VR), grant no. 621-2013-3901 and grant no. 2015-05422.

(1) Overview

Introduction

Partial Differential Equations (PDEs) describing natural phenomena are modelled using tensors of different order. Two commonly studied problems are heat transfer, which include temperature and heat flux (rank-0 and rank-1 tensor, respectively), and continuum mechanics, which include stress and strain (rank-2 tensors) and the so-called tangent stiffness (rank-4 tensor). These tensors are often symmetric but may also be non-symmetric, for example in finite deformation continuum mechanics.

In the implementation of numerical solution schemes for such PDEs, a large number of expressions involving tensors typically have to be computed. As an example, in the Finite Element Method (FEM), the weak form of the PDE to be solved is evaluated multiple times in every element of the mesh. The total number of elements in a model can easily exceed millions, and it is therefore desirable to have access to a library that can perform these tensor operations efficiently. Another aspect to consider is the level of difficulty to implement tensor expressions as computer source code based on their mathematical form. A close correspondence between the source code and the mathematical writing will likely lead to a quicker, less error prone, implementation process.

The classical way of treating tensors in computational mechanics is to use what is commonly called *Voigt notation* or *Voigt format*, described in many classic FEM textbooks [1, 3, 6]. In this format, second-order tensors are stored in vectors[1] of length n_{dim}^2 or $n_{\text{dim}}\,(n_{\text{dim}}+1)/2$ for the non-symmetric and symmetric case, respectively, where n_{dim} denotes the number of spatial dimensions for which the problem is formulated. Examples of second-order tensors are the symmetric Cauchy stress $\boldsymbol{\sigma}$ and the non-symmetric Kirchoff stress $\boldsymbol{\tau}$

$$\underline{\boldsymbol{\sigma}} = \begin{bmatrix} \sigma_{xx} & \sigma_{yy} & \sigma_{zz} & \sigma_{yz} & \sigma_{xz} & \sigma_{xy} \end{bmatrix}^{\mathrm{T}} \text{ symmetric,} \quad (1a)$$

$$\underline{\boldsymbol{\tau}} = \begin{bmatrix} \tau_{xx} & \tau_{yy} & \tau_{zz} & \tau_{yz} & \tau_{xz} \\ & & & \tau_{xy} & \tau_{zy} & \tau_{zx} & \tau_{yx} \end{bmatrix}^{\mathrm{T}} \text{ non-symmetric,} \quad (1b)$$

which are here represented in Voigt notation. Similarly, fourth-order tensors are stored in square matrices where the number of columns equals the length of the vector for the second-order tensors in Voigt notation.

The purpose of Voigt notation is that some standard linear algebra operators can be used for common tensor operations such as open products, dot products and double contractions. As an example, the double contraction between a fourth-order tensor $\mathbf{C}$ and a second-order tensor $\boldsymbol{\varepsilon}$ can be formulated as a matrix-vector multiplication, and the open product between two second-order tensors as a column vector times a row vector, viz.

$$\sigma_{ij} = C_{ijkl}\varepsilon_{kl} \quad \leftrightarrow \quad \underline{\boldsymbol{\sigma}} = \underline{\mathbf{C}}\,\underline{\boldsymbol{\varepsilon}}, \quad (2a)$$

$$E_{ijkl} = \nu_{ij}\nu_{kl} \quad \leftrightarrow \quad \underline{\mathbf{E}} = \underline{\boldsymbol{\nu}}\,\underline{\boldsymbol{\nu}}^{\mathrm{T}}. \quad (2b)$$

The left column is presented using the Einstein summation convention and the right column is presented with the corresponding, standard matrix operations.

In order to preserve the relation $\sigma_{ij}\varepsilon_{ij} = \underline{\boldsymbol{\sigma}}^{\mathrm{T}}\underline{\boldsymbol{\varepsilon}}$, where $\boldsymbol{\sigma}$ and $\boldsymbol{\varepsilon}$ are symmetric second-order tensors, it is common to define

$$\underline{\boldsymbol{\varepsilon}} = [\varepsilon_{xx}, \varepsilon_{yy}, \varepsilon_{zz}, \gamma_{yz}, \gamma_{xz}, \gamma_{xy}]^{\mathrm{T}}, \quad \gamma_{ij} = 2\varepsilon_{ij}, \tag{3}$$

where γ_{ij} is usually denoted "engineering strains".

While Voigt notation is simple to adopt when using programming languages or libraries that provide linear algebra functionality, it does, however, suffer from a few drawbacks regarding both performance and clarity:

- Some operations, such as the scalar product between a rank-2 and rank-1 tensor, become difficult since the rank-2 tensor is not stored as a matrix.
- Indexing into a tensor stored in Voigt format is not straightforward. Fetching the *xy* component of a rank-2 tensor $\boldsymbol{\sigma}$ would look like $\boldsymbol{\sigma}$[6], or $\boldsymbol{\sigma}$[idx[1,2]]] where `idx` is a lookup table matching the "Cartesian index" [1,2] to the "linear index" [6] which is the location of σ_{xy} in the Voigt vector.
- Some operations will silently give wrong result when naively applied to a tensor in Voigt notation. One example is the norm function $\|\boldsymbol{\sigma}\| = \sqrt{\sigma_{ij}\sigma_{ij}}$. Using the `norm` function for vectors will give the correct answer if the tensor is stored as an non-symmetric tensor. However, it will silently give the wrong result if the tensor is stored as a symmetric tensor (since the off-diagonal components will only be accounted for once).
- Since tensors in Voigt format are usually stored in arrays that allows arbitrary number of elements, a compiler cannot know the size of the array at compile time. This prohibits optimal code to be generated.
- In order for certain operations to work as "expected" for symmetric tensors, e.g. the double contraction $\sigma_{ij}\varepsilon_{ij} \leftrightarrow \underline{\boldsymbol{\sigma}}^{\mathrm{T}}\underline{\boldsymbol{\varepsilon}}$, the strain-like tensors are commonly stored with a factor two on the off-diagonals. This is a frequent source of confusion because the same mathematical object is stored differently.

`Tensors.jl` was created in an attempt to overcome the many deficiencies of Voigt notation. It is written in the programming language Julia [2].

The main design goals have been:

1. *Performance* – Operations should compile to (close to) the bare minimum of assembly operations needed. Symmetry should be exploited for computational efficiency. SIMD-instructions should be used when computationally beneficial.
2. *Generality* – The same code should work regardless if the number types are `double` or `single` precision (denoted `Float64` and `Float32` in Julia) or even user defined numerical types, e.g. dual numbers used in forward mode automatic differentiation. It should also be dimension independent so that the same code can be used for one, two and three spatial dimensions.
3. *Clarity* – Implementation details, such as the particular way a tensor is stored, should not be visible to the user. Symmetric and non-symmetric tensors should behave the same, with the difference that operations on symmetric tensors should, if possible, be faster. Operations should be visually similar to mathematical writing. This includes using Unicode infix operators such as ⊗ for the open product and · for the scalar product.

The main purpose of the software is to be used for solving PDEs modelling physical phenomena (such as heat flux and stress equilibrium). In particular this excludes rectangular tensors, and tensors of dimension higher than 3, which are commonly used in e.g. machine learning. Currently only rank-1, rank-2 and rank-4 tensors, in up to 3 dimensions, are implemented. We note that support for rank-3 tensors fall within the scope of the package, but is not yet implemented.

The `Tensor` and `SymmetricTensor` types

The foundation of the package are the two types `Tensor` and `SymmetricTensor`. For rank-1 tensors (vectors) a typealias called `Vec` is provided. The types are parameterized according to the rank, the dimension and the number type stored in the tensor.[2] Hence, the type `Tensor{2,3,Float64}` would represent a non-symmetric second-order tensor in three dimension that stores `Float64` (64 bit floating point) numbers. A few example of creating tensors of different rank, dimension and number type is shown below:[3]

```
julia> σ = rand(SymmetricTensor{2,2})
2×2 Tensors.SymmetricTensor{2,2,Float64,3}:
  0.590845  0.766797
  0.766797  0.566237

julia> τ = rand(Tensor{2,2})
2×2 Tensors.Tensor{2,2,Float64,4}:
  0.460085  0.854147
  0.794026  0.200586

julia> x = rand(Vec{3,Float32}) # same as
rand(Tensor{1,3,Float32})
3-element Tensors.Tensor{1,3,Float32,3}:
  0.950449
  0.49083
  0.589914
```

Operations on these tensors can now be performed. When there is a corresponding mathematical symbol for the operation, it can often be used directly:

```
julia> norm(τ)
1.1747430857785317

julia> σ ⋅ σ
2×2 Tensors.Tensor{2,2,Float64,4}:
  0.0328058  0.0608441
  0.0608441  0.139497

julia> x ⊗ x
3×3 Tensors.Tensor{2,3,Float32,9}:
  0.648977  0.70272   0.478066
  0.70272   0.760913  0.517655
  0.478066  0.517655  0.352164
```

We note that these operations all get compiled into very efficient machine code, and use SIMD whenever

it is beneficial. In **Table 1** we show a non-exhaustive summary of tensor operations that are currently implemented.

Illustrative usage example

As an illustrative example of using the package, we here give the mathematical formulation of an energy potential and stress in large deformation continuum mechanics and its implementation as a function in Julia.

For a deformation gradient $\boldsymbol{F} = \boldsymbol{I} + \nabla \otimes \boldsymbol{u}$, where $\boldsymbol{u}$ is the displacement from the reference to the current configuration, the right Cauchy-Green deformation tensor is defined by $\boldsymbol{C} = \boldsymbol{F}^{\mathrm{T}} \cdot \boldsymbol{F}$. The Second Piola-Kirchoff stress tensor $\boldsymbol{S}$ is derived from the Helmholtz free energy ψ by the relation $\boldsymbol{S} = 2\frac{\partial \Psi}{\partial C}$.

The free energy for a hyperelastic material can be defined as

$$\Psi(\boldsymbol{C}) = 1/2\mu(\mathrm{tr}(\hat{\boldsymbol{C}}) - 3) + K_b(J-1)^2, \quad (4)$$

where $\hat{\boldsymbol{C}} = \det(\boldsymbol{C})^{-1/3}\boldsymbol{C}$ and $J = det(\boldsymbol{F}) = \sqrt{det(\boldsymbol{C})}$ and the shear and bulk modulus are given by μ and K_b, respectively. This free energy function can be implemented in Julia as:

```
function ψ(C, μ, Kb)
    detC = det(C)
    J = sqrt(detC)
    Ĉ = detĈ (-1/3)*C
    return 1/2*(μ * (trace(Ĉ)- 3) + Kb*(J-1)^2)
end
```

The analytic expression for the Second Piola-Kirchoff tensor is

$$\boldsymbol{S} = 2\frac{\partial \Psi}{\partial \boldsymbol{C}} = \mu \det(\boldsymbol{C})^{-1/3}(\boldsymbol{I} - 1/3\mathrm{tr}(\boldsymbol{C})\boldsymbol{C}^{-1}) + K_b(J-1)J\boldsymbol{C}^{-1}, \quad (5)$$

which can be implemented by the following function:

```
function S(C, μ, Kb)
    I = one(C)
    detC = det(C)
    J = sqrt(detC)
    invC = inv(C)
    return μ * detĈ (-1/3)*(I -
    1/3*trace(C)*invC) + Kb*(J-1)*J*invC
end
```

These functions work in 1, 2 and 3 dimensions and have good performance.

Table 1: Summary of implemented tensor operations. $\boldsymbol{u}$, $\boldsymbol{v}$ denotes vectors, $\boldsymbol{A}$, $\boldsymbol{B}$ denotes second-order symmetric or non-symmetric tensors, and $\mathbf{C}$, $\mathbf{D}$ denotes fourth-order symmetric or non-symmetric tensors. We note that instead of using : for infix double contraction we use ⊡ (written as `\boxdot`). This is because : does not have the same operator precedence as multiplication in Julia.

Operation	Julia code	infix
Single contraction		
$\boldsymbol{u}\cdot\boldsymbol{v}\,(u_i v_i)$	`dot(u, v)`	u ⋅ v
$\boldsymbol{A}\cdot\boldsymbol{v}\,(A_{ij}v_j)$	`dot(A, v)`	A ⋅ v
$\boldsymbol{A}\cdot\boldsymbol{B}\,(A_{ij}B_{jk})$	`dot(A, B)`	A ⋅ B
Double contraction		
$\boldsymbol{A}:\boldsymbol{B}\,(A_{ij}B_{ij})$	`dcontract(A, B)`	A ⊡ B
$\mathbf{C}:\boldsymbol{B}\,(C_{ijkl}B_{kl})$	`dcontract(C, B)`	C ⊡ B
$\mathbf{C}:\mathbf{D}\,(C_{ijkl}D_{klmn})$	`dcontract(A, D)`	C ⊡ D
Outer product		
$\boldsymbol{u}\otimes\boldsymbol{v}\,(u_i v_j)$	`otimes(u, v)`	u ⊗ v
$\boldsymbol{A}\otimes\boldsymbol{B}\,(A_{ij}B_{kl})$	`otimes(A, B)`	A ⊗ B
Other operations		
$\det(\boldsymbol{A})$	`det(A)`	
$\mathrm{inv}(\boldsymbol{A})$	`inv(A)`	
$\mathrm{norm}(\boldsymbol{A})$	`norm(A)`	
$\boldsymbol{A}^{\mathrm{T}}$	`transpose(A)`	
$\frac{1}{2}(\boldsymbol{A}+\boldsymbol{A}^{\mathrm{T}})$	`symmetric(A)`	
$\frac{1}{2}(\boldsymbol{A}-\boldsymbol{A}^{\mathrm{T}})$	`skew(A)`	

Automatic Differentiation

Automatic differentiation [4] (AD) is a numerical method for differentiating functions implemented in a programming language. AD has many advantages over other numerical methods for differentiation. Comparing it to numerical differentiation, where components are perturbed to compute a gradient, AD does not suffer from cancellation and can compute multiple partial derivatives in a single function call. `Tensors.jl` supports AD and as an example of its use, we here recall the function $\boldsymbol{S} = 2\frac{\partial \Psi}{\partial \boldsymbol{C}}$, compute it using AD, and compare with the analytical result:

```
julia> H = rand(Tensor{2,3}); F = one(H) + H; C
= symmetric(F' ⋅ F);

julia> 2 * gradient(C -> ψ (C, μ, Kb), C)
3×3 Tensors.SymmetricTensor{2,3,Float64,6}:
    7.35076  -2.51778   0.489453
   -2.51778   6.36214  -3.21338
    0.489453 -3.21338   8.21286

julia> S(C, μ, Kb)
3×3 Tensors.SymmetricTensor{2,3,Float64,6}:
    7.35076  -2.51778  0.489453
   -2.51778   6.36214 -3.21338
    0.489453 -3.21338  8.21286
```

The slowdown from using AD instead of the analytic version is about a factor of 3. It is also possible to compute second-order derivatives exactly in an analogous manner as the example for first order derivatives above.

The AD-functionality is built upon the *dual numbers* defined in `ForwardDiff.jl` [5].

Performance

In this section we compare the performance of a selected number of operations when using the `Tensor` types to the Voigt format, implemented with standard `Array` types.[4] The results are shown in **Table 2**, where $\boldsymbol{u}$ denotes a vector, $\boldsymbol{A}$, $\boldsymbol{A}^{\mathrm{sym}}$ denote second-order non-symmetric and symmetric tensors and $\mathbf{C}$, $\mathbf{C}^{\mathrm{sym}}$ denote fourth-order non-symmetric and symmetric tensors, respectively. All tensors presented are in three dimensions. The

Table 2: Comparison of performance for some tensor operations using `Tensors.jl` and Voigt format using Julia `Arrays`.

Operation	`Tensor`	`Array`	Speed-up
Single contraction			
$\boldsymbol{u} \cdot \boldsymbol{u}$	1.241 ns	9.795 ns	×7.9
$\boldsymbol{A} \cdot \boldsymbol{u}$	2.161 ns	58.769 ns	×27.2
$\boldsymbol{A} \cdot \boldsymbol{A}$	3.117 ns	44.395 ns	×14.2
$\boldsymbol{A}^{\text{sym}} \cdot \boldsymbol{A}^{\text{sym}}$	5.125 ns	44.498 ns	×8.7
Double contraction			
$\boldsymbol{A} : \boldsymbol{A}$	1.927 ns	12.189 ns	×6.3
$\boldsymbol{A}^{\text{sym}} : \boldsymbol{A}^{\text{sym}}$	1.927 ns	12.187 ns	×6.3
$\boldsymbol{C} : \boldsymbol{A}$	6.087 ns	78.554 ns	×12.9
$\boldsymbol{C} : \boldsymbol{C}$	60.820 ns	280.502 ns	×4.6
$\boldsymbol{C}^{\text{sym}} : \boldsymbol{C}^{\text{sym}}$	22.104 ns	281.003 ns	×12.7
$\boldsymbol{A}^{\text{sym}} : \boldsymbol{C}^{\text{sym}} : \boldsymbol{A}^{\text{sym}}$	9.466 ns	89.747 ns	×9.5
Outer product			
$\boldsymbol{u} \otimes \boldsymbol{u}$	2.167 ns	32.447 ns	×15.0
$\boldsymbol{A} \otimes \boldsymbol{A}$	9.801 ns	6.568 ns	×8.8
Other operations			
det($\boldsymbol{A}$)	1.924 ns	177.134 ns	×92.1
inv($\boldsymbol{A}^{\text{sym}}$)	4.587 ns	635.858 ns	×138.6
norm($\boldsymbol{A}$)	1.990 ns	16.752 ns	×8.4

benchmarks were performed using the benchmarking tool `BenchmarkTools.jl`.[5]

Correctness and performance testing

The package is tested for correctness using Continuous Integration (CI) on macOS, Linux and Windows versions of Julia. An extensive test suite based on unit testing is used. The testing includes tensor identities, and tests of different tensor operations. The results are compared with the matrix/vector representation of the tensors using standard Julia `Arrays`. Code coverage for the package is currently at 95%. The examples in the documentation are written as "doctests" which means that the code in the examples are running as part of CI. This is to prevent examples in the documentation from becoming stale. A comprehensive set of benchmarks are implemented and used to check that regressions in performance are not introduced.

Implementation details

The elements of a tensor are stored internally in a `Tuple`. In Julia, a `Tuple` is an immutable container, where the length is part of the type. Consequently, when compiling a function, the Julia compiler can statically know the length of the tuple-container and use that information for optimizations. Furthermore, tuples can be stored on the stack which removes any use of expensive heap allocations, alleviating the need to preallocate output buffers.

For symmetric tensors only the "lower half" of the tensor is stored. A non-symmetric fourth-order tensors in 3 dimensions has 81 elements, while the symmetric version only need to store 36 elements.

Operations on tensors are frequently implemented using, what in Julia are called, "generated functions". This allows the programmer to hook into the compilation process, at the time where the types of the arguments in the called function are known. Based on these types, the programmer can generate arbitrary Julia code which get compiled just like a normal function. This is, for example, used to generate the SIMD code for different tensor sizes.

Julia will also generate specialized code even when a user defined numeric type is used as the elements of the tensors. This is the case when doing automatic differentiation where a dual-number is used as the element type. This means that well performing code is generated even for automatic differentiation.

Since the number of dimensions and ranks for the tensors that this package support is limited, excessive compilation and code generation is prevented.

Conclusion

We have presented the Julia package `Tensors.jl`. It provides a framework for doing computations with non-symmetric and symmetric tensors of rank-1, rank-2 and rank-4 with arbitrary number types. The implementation has many advantages over using the common Voigt format, such as, higher performance, dimension generality, and allows for a more direct mapping between the source code and the mathematical notation. Automatic differentiation for first and second-order derivatives is supported and is implemented efficiently.

(2) Availability

Operating system

Any OS supported by Julia which include FreeBSD, Windows, macOS, Linux, Raspberry Pi and other ARM systems.

Programming language

Julia 0.6

Dependencies

The following packages are required to use `Tensors.jl`:

- `ForwardDiff.jl` (https://github.com/JuliaDiff/ForwardDiff.jl) – Used for dual numbers which is the engine behind the automatic differentiation functionality.
- `SIMD.jl` (https://github.com/eschnett/SIMD.jl) – Used for SIMD operations.

These dependencies are automatically installed upon installing via `Pkg.add("Tensors")`.

List of contributors

Kristoffer Carlsson, Fredrik Ekre, Keita Nakamura

Software location

Name: KristofferC/Tensors.jl (https://github.com/KristofferC/Tensors.jl)

Persistent identifier: https://doi.org/10.5281/zenodo.802357

Licence: MIT

Publisher: Kristoffer Carlsson

Version published: v0.7.1

Date published: 03/06/17

Code repository GitHub

Name: KristofferC/Tensors.jl (https://github.com/KristofferC/Tensors.jl)

Licence: MIT

Date published: 25/05/16

Language

English

(3) Reuse potential

Since tensor computations is of such fundamental use when modeling a large class of physical systems, a high quality implementation has high reuse potential in other packages modeling these physical systems, where `Tensors.jl` can serve as a "backend" for the tensor computations. One such example is the Finite Element toolbox `JuAFEM.jl`[6] where `Tensors.jl` handles all the tensor computations. Extensions to the package are welcome to be submitted as Pull Request to the GitHub repository of the package. Support is available in terms of documentation as well as an issue tracker on the repository, which is open for anyone to post questions.

Notes

[1] The term *vector* here refers to an arbitrary length, one dimensional array of numbers, in contrast to a rank-1 tensor which is also commonly called a vector.

[2] For technical reasons the total number of elements stored is also a parameter of the type, but this is rarely of interest to a user of the package.

[3] Unicode symbols like σ can be entered in the Julia REPL by entering `\sigma` and pressing `TAB`.

[4] Benchmarks were performed with an Intel(R) Core(TM) i5-7600K CPU @ 3.80 GHz CPU.

[5] https://github.com/JuliaCI/BenchmarkTools.jl.

[6] https://github.com/KristofferC/JuAFEM.jl.

Acknowledgements

We would like to thank the Julia community and, especially, Jarett Revels for creating `ForwardDiff.jl` and Erik Schnetter for creating `SIMD.jl`.

Competing Interests

The authors have no competing interests to declare.

References

1. **Bathe, K J** 1996 *Finite Element Procedures*. Prentice-Hall International Series in. Prentice Hall. ISBN: 9780133014587.
2. **Bezanson, J,** et al. Jan. 2017 "Julia: A Fresh Approach to Numerical Computing". In: *SIAM Review*, 59(1): 65–98. ISSN: 0036-1445. DOI: https://doi.org/10.1137/141000671
3. **Hughes, T J R** 1987 *The finite element method: linear static and dynamic finite element analysis*. Englewood Cliffs, N.J.: Prentice-Hall International. ISBN: 0-13-317025-X.
4. **Naumann, U** 2012 *The Art of Differentiating Computer Programs: An Introduction to Algorithmic Differentiation*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics. ISBN: 161197206X, 9781611972061.
5. **Revels, J, Lubin, M** and **Papamarkou, T** 2016 "Forward-Mode Automatic Differentiation in Julia". In: *arXiv:1607.07892 [cs.MS]*. URL: https://arxiv.org/abs/1607.07892.
6. **Zienkiewicz, O C, Taylor, R L** and **Fox, D** 2014 "Chapter 1 – General Problems in Solid Mechanics and Nonlinearity". In: *The Finite Element Method for Solid and Structural Mechanics (Seventh Edition)*, Zienkiewicz, O C, Taylor, R L and Fox, D (eds.), 1–20. Seventh Edition. Oxford: Butterworth-Heinemann. ISBN: 978-1-85617-634-7ss. DOI: https://doi.org/10.1016/B978-1-85617-634-7.00001-6.

How to cite this article: Carlsson, K and Ekre, F 2019 `Tensors.jl` — Tensor Computations in Julia. *Journal of Open Research Software,* 7: 7. DOI: https://doi.org/10.5334/jors.182

Submitted: 11 June 2017 **Accepted:** 06 June 2018 **Published:** 21 March 2019