# Tensor classes

release 0.3

Ton van den Boogaard

December 1, 2003

# Contents

# Chapter 1

# Introduction

The tensor classes described in this document are declared in the file `tensor.h` and defined in both `tensor.h` and `tensor.cpp`. The classes are developed with applications in the field of computational mechanics in mind. The tensor classes were applied to continuum mechanics finite element applications and to crystal plasticity analysis. The spatial dimension of all classes is always 3 and the implementation assumes a Cartesian coordinate system. The implementation therefore is easily described in index notation.

For the moment only first, second and fourth order tensors are defined. The second and fourth order tensors have general and a symmetric versions. The symmetric versions use only the essential storage and by defining different classes for the general and the symmetric tensors the implementation of algorithms can be optimized for the specific types. This has been done in some cases where this was deemed useful. Also, some functions are only available for the general tensors and some only for the symmetric tensors, e.g. the extraction of eigenvalues is only available for symmetric second order tensors and the transpose tensor only for the general second order tensors.

Automatic conversion from the symmetric to the general second and fourth order tensors is implemented, so algorithms for the general tensors can also be used for the symmetric versions, although with some copying penalty. With some compilers this sometimes results in ambiguities with the functions defined in the base classes. This is (probably) an implementation error in the compilers, because base-class functions should precede class conversions, if I interpret Stroustrups book correctly. The ambiguity can be solved by casting e.g. the symmetric second order tensor to a **const Tensor2&** base class or to a **Tensor2Gen** general second order tensor.

An extension of tensor functions is given in the file `xtensor.h` and `xtensor.cpp`. These are separated from the main files because they are not general enough, or because they only show an implementation example that you will probably like to change. These functions are shortly documented at the end of this manual.

Two test programs are included to test the basic and extended tensor functions. They are described in the next section.

# Chapter 2

# Installation

There is no installation ...

The files that must be reached when making an application program are `tensor.h` and `tensor.o`. For the extension functions you also need `xtensor.h`, `xtensor.o` and `tutils.o` and to make `tutils.o` you need `tutils.h`. You can put them anywhere you like, e.g. between your application source code. There *are* two small test programs that can be used to validate the tensor code on your machine. Compile `basetest.cpp` and `xtest.cpp` and run them. The output should be equal (apart from round-off errors) to the files `basetest.ref` and `xtest.ref`. A small C-program `testdiff` is supplied to filter round-off errors and 'numerical zeros'.

A makefile is provided for use on Unix machines and for the Borland C++ compiler on MS-Windows. For Unix machines you should adapt the symbols `CXX`, `CC`, `CXXFLAGS` and `LDFLAGS` to indicate your favorite C++ compiler, C compiler, compile flags and link/load flags. The C compiler is used for the `testdiff` program only. For Borland, use the file `makefile.b32`. Some older C++ compilers—among which the Gnu versions 2.9x—do not recognize the standard C++ `#include<limits>` statement. By adding `NO_STD_LIMITS` to the `CXXFLAGS` the include file `floats.h` is tried instead. The targets of make are:

| | |
|---|---|
| (default) | to make the tensor object |
| all | to make all object and test programs |
| basetest | to make the tensor object and basic test program |
| xtest | to make the tensor object and extended test program |
| test | to make all tests and if necessary all objects and programs |
| clean | to delete all generated objects, executables and output files |
| cleanall | (only for Borland) as clean, but also delete Borland project files |

So, shortly, just type 'make test' or for Borland 'make -f makefile.b32 test' and everything will be made and tested. If all goes well you will see at the end something like:

```
./basetest > basetest.out
./testdiff basetest.ref basetest.out
./xtest > xtest.out
./testdiff xtest.ref xtest.out
```

indicating that the output for basetest is compared with basetest.ref and the output for xtest with xtest.ref. If relevant differences are found, these will be printed after the

testdiff command.

# Chapter 3

# Class descriptions

In this chapter, the definition of the first, second and fourth order tensors is described. The **Tensor1** class is a concrete class, but the **Tensor2** and **Tensor4** classes are abstract classes. The concrete second and fourth order tensors are the general and symmetric specializations: **Tensor2Gen**, **Tensor2Sym**, **Tensor4Gen** and **Tensor4DSym** classes. The reason for not deriving the symmetric tensors from the general tensors is that, especially for the second order tensors, it was decided to allocate the data statically in the objects and not by dynamic allocation.

## 3.1 Tensor1

The **Tensor1** class represents first order tensors.

**Tensor1**()

Constructor for Tensor1 objects, leaves contents uninitialized.

*explicit* **Tensor1**( *short* **i** )

Constructor for Tensor1 objects, **i** can only be '0' and the tensor will be initialized to 0 then.

*double&* **operator()**( *int* **i** )

Returns a reference to the $i$-th value of the tensor. This value can be used as an lvalue.

*const double&* **operator()**( *int* **i** ) *const*

Returns a const reference to the $i$-th value of the tensor.

*Tensor1&* **operator+=**( *const Tensor1&* **b** )

Adds **b** to this tensor.

*Tensor1&* **operator−=**( *const Tensor1&* **b** )

Subtracts **b** from this tensor.

*Tensor1&* **operator∗=**( *double* **s** )

Multiplies this tensor with **s**.

*Tensor1&* **operator/=**( *double* **s** )

Divides this tensor by **s**.

*Tensor1&* **operator=**( *short* **i** )

The value of **i** must be '0'. The value of this tensor will be set to 0 then.

## 3.2 Tensor2

The **Tensor2** class represents second order tensors. It is a virtual base class.

*virtual* **~Tensor2**()

Destructor for the Tensor2 objects.

*virtual const double&* **operator()**( *int* **i**, *int* **j** ) *const*=0

Returns a const reference to the $(i, j)$ value of the tensor. This reference can not be used as an lvalue. It is a pure virtual function, so it must be defined in derived classes.

*virtual void* **invariants**( *double** **I**, *double** **II**, *double** **III**) *const*

Returns the invariants of a this tensor. For a second order tensor **A**, the invariants are defined by:

$$I = \mathrm{tr}(\mathbf{A})$$
$$II = \tfrac{1}{2}[(\mathrm{tr}(\mathbf{A}))^2 - \mathrm{tr}(\mathbf{A}^2)]$$
$$III = \det(\mathbf{A})$$

## 3.3 Tensor2Gen

The **Tensor2Gen** class represents a general 2nd order tensor with 9 independent values.

**Tensor2Gen**()

Constructor for Tensor2Gen objects, leaves contents uninitialized.

*explicit* **Tensor2Gen**( *short* **i** )

Constructor for Tensor2Gen objects, **i** can only be '0' or '1'. If **i** is '0' the tensor will be initialized to 0, if the value is '1' it will be initialized as an identity tensor.

**Tensor2Gen**( *const Tensor2Sym&* **i** )

Constructor for Tensor2Gen objects, initialized by a symmetric second order tensor.

*Tensor2Gen* **transpose**() *const*

Returns the transpose of this tensor.

*double&* **operator()**( *int* **i**, *int* **j** )

Returns a reference to the $(i, j)$ value of the tensor. This reference can be used as an lvalue.

*const double&* **operator()**( *int* **i**, *int* **j** ) *const*

Returns a const reference to the $(i, j)$ value of the tensor. This reference can not be used as an lvalue.

*Tensor2Gen&* **operator+=**( *const Tensor2Gen&* **B** )

Adds **B** to this tensor.

*Tensor2Gen&* **operator−=**( *const Tensor2Gen&* **B** )

Subtracts **B** from this tensor.

*Tensor2Gen&* **operator∗=**( *double* **s** )

Multiplies this tensor with **s**.

*Tensor2Gen&* **operator/=**( *double* **s** )

Divides this tensor by **s**.

*Tensor2Gen&* **operator=**( *short* **i** )

The value of **i** can only be '0' or '1'. If **i** is '0' the tensor will be set to 0, if the value is '1' it will be set to the identity tensor.

*const Tensor1* **operator()**( *int* **i** ) *const*

Returns column number **i** of this tensor.

## 3.4 Tensor2Sym

The **Tensor2Sym** class represents a symmetric 2nd order tensor. Only the 6 independent values are stored and the structure is used in many of the algorithms to optimize this class.

**Tensor2Sym**()

Constructor for Tensor2Sym objects, leaves contents uninitialized.

*explicit* **Tensor2Sym**( *short* **i** )

Constructor for Tensor2Sym objects, **i** can only be '0' or '1'. If **i** is '0' the tensor will be initialized to 0, if the value is '1' it will be initialized as an identity tensor.

*double&* **operator()**( *int* **i**, *int* **j** )

Returns a reference to the $(i, j)$ value of the tensor. A condition is that $i \geq j$. This reference can be used as an lvalue. The condition avoids that by a loop over all 9 components accidentally the off-diagonal components are incremented twice.

*const double&* **operator()**( *int* **i**, *int* **j** ) *const*

Returns a const reference to the $(i, j)$ value of the tensor. This reference can not be used as an lvalue.

*virtual void* **invariants**( *double∗* **I**, *double∗* **II**, *double∗* **III**) *const*

Specialization of the generic *Tensor2* **f**unction for more efficiency.

*void* **eigen**( *double** **lambda1**, *double** **lambda2**, *double** **lambda3**, *Tensor2Gen** **V**=*0*) *const*

Returns the eigenvalues of this tensor. If the pointer to **V** is not '0', then the eigenvectors $\mathbf{v}_1$, $\mathbf{v}_2$ and $\mathbf{v}_3$ are returned as columns of the second order tensor **V**. You can retrieve the eigenvectors most easily with the operator: V($i$) (see Section 3.3).

*Tensor2Sym&* **operator+=**( *const Tensor2Sym&* **B** )

Adds **B** to this tensor.

*Tensor2Sym&* **operator−=**( *const Tensor2Sym&* **B** )

Subtracts **B** from this tensor.

*Tensor2Sym&* **operator*=**( *double* **s** )

Multiplies this tensor with **s**.

*Tensor2Sym&* **operator/=**( *double* **s** )

Divides this tensor by **s**.

*Tensor2Sym&* **operator=**( *short* **i** )

The value of **i** can only be '0' or '1'. If **i** is '0' the tensor will be set to 0, if the value is '1' it will be set to the identity tensor.

*void* **getVector**( *double** **A** )

Stores the tensor components in an array **A** of 6 doubles as

$$\begin{bmatrix} a_{11} & a_{22} & a_{33} & \sqrt{2}a_{12} & \sqrt{2}a_{23} & \sqrt{2}a_{31} \end{bmatrix}$$

These values correspond to definitions of a matrix for the symmetric fourth order tensors. This function is intended for use of standard (fortran) linear algebra packages. The $\sqrt{2}$ values are convenient when using matrix–vector algebra with the reduced set of 6 components for the symmetric tensor. For application examples see **Tensor4DSym::getFortranMatrix**.

*void* **putVector**( *const double** **A** )

Initialize the tensor components from an array **A** of 6 doubles. The array should have the values as indicated for **getVector**. It is intended to return results from standard linear algebra packages into the tensor classes.

## 3.5 Tensor4

The **Tensor4** class represents fourth order tensors. It is a virtual base class.

*virtual* **~Tensor4**()

Destructor for the Tensor4 objects.

*virtual const double&* **operator()**( *int* **i**, *int* **j** ) *int* **k**, *int* **l** ) *const*=0

Returns a const reference to the $(i, j, k, l)$ value of the tensor. This reference can not be used as an lvalue.

## 3.6 Tensor4Gen

The **Tensor4Gen** class represents a general 4th order tensor with 81 independent values.

**Tensor4Gen**()

Constructor for Tensor4Gen objects, leaves contents uninitialized.

*explicit* **Tensor4Gen**( *short* **i** )

Constructor for Tensor2Gen objects, **i** can only be '0' or '1'. If **i** is '0' the tensor will be initialized to 0, if the value is '1' it will be initialized as a 4th order identity tensor $\delta_{ik}\delta_{jl}$.

**Tensor4Gen**( *const Tensor4DSym&* **i** )

Constructor for Tensor4Gen objects, initialized by a double symmetric 4th order tensor.

*double&* **operator()**( *int* **i**, *int* **j**, *int* **k**, *int* **l** )

Returns a reference to the $(i, j, k, l)$ value of the tensor. This reference can be used as an lvalue.

*const double&* **operator()**( *int* **i**, *int* **j**, *int* **k**, *int* **l** ) *const*

Returns a const reference to the $(i, j, k, l)$ value of the tensor. This reference can not be used as an lvalue.

*Tensor4Gen&* **operator+=**( *const Tensor4Gen&* **B** )

Adds **B** to this tensor.

*Tensor4Gen&* **operator−=**( *const Tensor4Gen&* **B** )

Subtracts **B** from this tensor.

*Tensor4Gen&* **operator∗=**( *double* **s** )

Multiplies this tensor with **s**.

*Tensor4Gen&* **operator/=**( *double* **s** )

Divides this tensor by **s**.

*Tensor4Gen&* **operator=**( *short* **i** )

The value of **i** can only be '0' or '1'. If **i** is '0' the tensor will be set to 0, if the value is '1' it will be set to a 4th order identity tensor $\delta_{ik}\delta_{jl}$.

## 3.7 Tensor4DSym

The **Tensor4DSym** class represents a doubly symmetric 4th order tensor with 21 independent values ($A_{ijkl} = A_{jikl} = A_{klij}$).

**Tensor4DSym**()

Constructor for Tensor4DSym objects, leaves contents uninitialized.

*explicit* **Tensor4DSym**( *short* **i** )

Constructor for Tensor4DSym objects, **i** can only be '0' or '1'. If **i** is '0' the tensor will be initialized to 0, if the value is '1' it will be initialized as a symmetric 4th order identity tensor $\frac{1}{2}(\delta_{ik}\delta_{jl} + \delta_{il}\delta_{jk})$.

*double&* **operator()**( *int* **i**, *int* **j**, *int* **k**, *int* **l** )

Returns a reference to the $(i, j, k, l)$ value of the tensor. A condition is that $i \geq j$, $k \geq l$ and $ij \geq kl$ if the order a combination of two indices is 11, 22, 33, 21, 32, 31 (the usual vector representation of a symmetric second order tensor). This reference can be used as an lvalue. The conditions avoid that by a loop over all 81 components of the tensor, accidentally some components are incremented more than once.

*const double&* **operator()**( *int* **i**, *int* **j**, *int* **k**, *int* **l** ) *const*

Returns a const reference to the $(i, j, k, l)$ value of the tensor. This reference can not be used as an lvalue.

*Tensor4DSym&* **operator+=**( *const Tensor4DSym&* **B** )

Adds **B** to this tensor.

*Tensor4DSym&* **operator−=**( *const Tensor4DSym&* **B** )

Subtracts **B** from this tensor.

*Tensor4DSym&* **operator∗=**( *double* **s** )

Multiplies this tensor with **s**.

*Tensor4DSym&* **operator/=**( *double* **s** )

Divides this tensor by **s**.

*Tensor4DSym&* **operator=**( *short* **i** )

The value of **i** can only be '0' or '1'. If **i** is '0' the tensor will be set to 0, if the value is '1' it will be set to a symmetric 4th order identity tensor $\frac{1}{2}(\delta_{ik}\delta_{jl} + \delta_{il}\delta_{jk})$.

*void* **getFortranMatrix**( *double∗* **A** )

Stores the tensor components in an array **A** of 36 doubles, column by column. The column and row sequence corresponds to the storage of a vector in the **Tensor2Sym** class:

$$\begin{bmatrix} a_{1111} & a_{1122} & a_{1133} & \sqrt{2}a_{1112} & \sqrt{2}a_{1123} & \sqrt{2}a_{1131} \\ a_{2211} & a_{2222} & a_{2233} & \sqrt{2}a_{2212} & \sqrt{2}a_{2223} & \sqrt{2}a_{2231} \\ a_{3311} & a_{3322} & a_{3333} & \sqrt{2}a_{3312} & \sqrt{2}a_{3323} & \sqrt{2}a_{3331} \\ \sqrt{2}a_{1211} & \sqrt{2}a_{1222} & \sqrt{2}a_{1233} & 2a_{1212} & 2a_{1223} & 2a_{1231} \\ \sqrt{2}a_{2311} & \sqrt{2}a_{2322} & \sqrt{2}a_{2333} & 2a_{2312} & 2a_{2323} & 2a_{2331} \\ \sqrt{2}a_{3111} & \sqrt{2}a_{3122} & \sqrt{2}a_{3133} & 2a_{3112} & 2a_{3123} & 2a_{3131} \end{bmatrix}$$

This function is intended for use of standard (fortran) linear algebra packages. The values are chosen, such that the tensor contraction $\mathbf{A} : \mathbf{b} = \mathbf{c}$ is represented by the matrix–vector multiplication $\mathbf{Ab} = \mathbf{c}$, with the reduced set of 36 and 6 independent components of the symmetric 4th and 2nd order tensors. E.g. matrix inversion and

solution of linear sets of equations can now be solved by standard matrix–vector algorithms.

*void* **putFortranMatrix**( *const double** **A** )

Initialize the tensor components from an array **A** of 36 doubles, column by column. The values are as indicated for **getFortranMatrix**.

## 3.8 Tensor4LSym

The **Tensor4LSym** class represents a lower symmetric 4th order tensor with 36 independent values ($A_{ijkl} = A_{jikl} = A_{ijlk}$ but $A_{ijkl} \neq A_{klij}$).

**Tensor4LSym**()

Constructor for Tensor4LSym objects, leaves contents uninitialized.

*explicit* **Tensor4LSym**( *short* **i** )

Constructor for Tensor2LSym objects, **i** can only be '0' or '1'. If **i** is '0' the tensor will be initialized to 0, if the value is '1' it will be initialized as a symmetric 4th order identity tensor $\frac{1}{2}(\delta_{ik}\delta_{jl} + \delta_{il}\delta_{jk})$.

**Tensor4LSym**(*const Tensor4DSym&* **S**)

Constructor for Tensor4LSym objects, initialized from a Tensor4DSym object.

*double&* **operator()**( *int* **i**, *int* **j**, *int* **k**, *int* **l** )

Returns a reference to the $(i, j, k, l)$ value of the tensor. A condition is that $i \geq j$ and $k \geq l$. This reference can be used as an lvalue. The conditions avoid that by a loop over all 81 components of the tensor, accidentally some components are incremented more than once.

*const double&* **operator()**( *int* **i**, *int* **j**, *int* **k**, *int* **l** ) *const*

Returns a const reference to the $(i, j, k, l)$ value of the tensor. This reference can not be used as an lvalue.

*Tensor4LSym&* **operator+=**( *const Tensor4LSym&* **B** )

Adds **B** to this tensor.

*Tensor4LSym&* **operator−=**( *const Tensor4LSym&* **B** )

Subtracts **B** from this tensor.

*Tensor4LSym&* **operator∗=**( *double* **s** )

Multiplies this tensor with **s**.

*Tensor4LSym&* **operator/=**( *double* **s** )

Divides this tensor by **s**.

*Tensor4LSym&* **operator=**( *short* **i** )

The value of **i** can only be '0' or '1'. If **i** is '0' the tensor will be set to 0, if the value is '1' it will be set to a symmetric 4th order identity tensor $\frac{1}{2}(\delta_{ik}\delta_{jl} + \delta_{il}\delta_{jk})$.

*void* **getFortranMatrix**( *double** **A** )

Stores the tensor components in an array **A** of 36 doubles, column by column, multiplied in a specific manner. See **Tensor4DSym::getFortranMatrix** for the storage and intended use.

*void* **putFortranMatrix**( *const double** **A** )

Initialize the tensor components from an array **A** of 36 doubles, column by column, multiplied in a specific manner. See **Tensor4DSym::getFortranMatrix** and **Tensor4DSym::putFortranMatrix** for the storage and intended use

# Chapter 4

# Global functions

## 4.1 Stream operators

For all classes, the <<operator is defined to put the tensor components on an output stream. The output starts on the current line and ends with an end of line, so later output will start on the next line. The width of each component can be set by the **setw** manipulator.

*ostream&* **operator**<<( *ostream&* **s**, *const Tensor1&* **a** )

Puts tensor **a** on the output stream **s** as 3 numbers in a row.

*ostream&* **operator**<<( *ostream&* **s**, *const Tensor2&* **A** )

Puts tensor **A** on the output stream **s** as 3 rows with each 3 numbers, the row represents the $i$ and the column the $j$ value of $A_{ij}$.

*ostream&* **operator**<<( *ostream&* **s**, *const Tensor4Gen&* **A** )

Puts tensor **A** on the output stream **s** as 9 rows with each 9 numbers, the row represents the $ij$ combination in the order 11, 12, 13, 21, 22, 23, 31, 32 and 33. The column represents the same order for the $kl$ pair of $A_{ijkl}$.

*ostream&* **operator**<<( *ostream&* **s**, *const Tensor4DSym&* **A** )
*ostream&* **operator**<<( *ostream&* **s**, *const Tensor4LSym&* **A** )

Puts tensor **A** on the output stream **s** as 6 rows with each 6 numbers, the row represents the $ij$ combination in the order 11, 22, 33, 12, 23 and 31 as in the usual vector presentation of a symmetric second order tensor. The column represents the same order for the $kl$ pair of $A_{ijkl}$.

## 4.2 Basic arithmetic operators

### 4.2.1 Unary operators

Unary + and – operators are used to explicitly specify (+**A**) or (−**A**).

*Tensor1* **operator+**( *const Tensor1&* **A** )
*Tensor2Gen* **operator+**( *const Tensor2Gen&* **A** )
*Tensor2Sym* **operator+**( *const Tensor2Sym&* **A** )

*Tensor4Gen* **operator+**( *const Tensor4Gen&* **A** )
*Tensor4DSym* **operator+**( *const Tensor4DSym&* **A** )
*Tensor4LSym* **operator+**( *const Tensor4LSym&* **A** )

Unary + operator, returns **A**.

*Tensor1* **operator–**( *const Tensor1&* **A** )
*Tensor2Gen* **operator–**( *const Tensor2Gen&* **A** )
*Tensor2Sym* **operator–**( *const Tensor2Sym&* **A** )
*Tensor4Gen* **operator–**( *const Tensor4Gen&* **A** )
*Tensor4DSym* **operator–**( *const Tensor4DSym&* **A** )
*Tensor4LSym* **operator–**( *const Tensor4LSym&* **A** )

Unary – operator, returns –**A**.

### 4.2.2 Binary operators

*Tensor1* **operator+**( *const Tensor1&* **A**, *const Tensor1&* **B** )
*Tensor2Gen* **operator+**( *const Tensor2Gen&* **A**, *const Tensor2Gen&* **B** )
*Tensor2Sym* **operator+**( *const Tensor2Sym&* **A**, *const Tensor2Sym&* **B** )
*Tensor4Gen* **operator+**( *const Tensor4Gen&* **A**, *const Tensor4Gen&* **B** )
*Tensor4DSym* **operator+**( *const Tensor4DSym&* **A**, *const Tensor4DSym&* **B** )
*Tensor4LSym* **operator+**( *const Tensor4LSym&* **A**, *const Tensor4LSym&* **B** )

Returns $\mathbf{R} = \mathbf{A} + \mathbf{B}$.

*Tensor1* **operator–**( *const Tensor1&* **A**, *const Tensor1&* **B** )
*Tensor2Gen* **operator–**( *const Tensor2Gen&* **A**, *const Tensor2Gen&* **B** )
*Tensor2Sym* **operator–**( *const Tensor2Sym&* **A**, *const Tensor2Sym&* **B** )
*Tensor4Gen* **operator–**( *const Tensor4Gen&* **A**, *const Tensor4Gen&* **B** )
*Tensor4DSym* **operator–**( *const Tensor4DSym&* **A**, *const Tensor4DSym&* **B** )
*Tensor4LSym* **operator–**( *const Tensor4LSym&* **A**, *const Tensor4LSym&* **B** )

Returns $\mathbf{R} = \mathbf{A} - \mathbf{B}$.

*Tensor1* **operator∗**( *const Tensor1&* **A**, *double* **s** )
*Tensor1* **operator∗**( *double* **s**, *const Tensor1&* **A** )
*Tensor2Gen* **operator∗**( *const Tensor2Gen&* **A**, *double* **s** )
*Tensor2Gen* **operator∗**( *double* **s**, *const Tensor2Gen&* **A** )
*Tensor2Sym* **operator∗**( *const Tensor2Sym&* **A**, *double* **s** )
*Tensor2Sym* **operator∗**( *double* **s**, *const Tensor2Sym&* **A** )
*Tensor4Gen* **operator∗**( *const Tensor4Gen&* **A**, *double* **s** )
*Tensor4Gen* **operator∗**( *double* **s**, *const Tensor4Gen&* **A** )
*Tensor4DSym* **operator∗**( *const Tensor4DSym&* **A**, *double* **s** )
*Tensor4DSym* **operator∗**( *double* **s**, *const Tensor4DSym&* **A** )
*Tensor4LSym* **operator∗**( *const Tensor4LSym&* **A**, *double* **s** )
*Tensor4LSym* **operator∗**( *double* **s**, *const Tensor4LSym&* **A** )

Returns $\mathbf{R} = s\mathbf{A} = \mathbf{A}s$.

*Tensor1* **operator/**( *const Tensor1&* **A**, *double* **s** )
*Tensor2Gen* **operator/**( *const Tensor2Gen&* **A**, *double* **s** )

*Tensor2Sym* **operator/**( *const Tensor2Sym&* **A**, *double* **s** )
*Tensor4Gen* **operator/**( *const Tensor4Gen&* **A**, *double* **s** )
*Tensor4DSym* **operator/**( *const Tensor4DSym&* **A**, *double* **s** )
*Tensor4LSym* **operator/**( *const Tensor4LSym&* **A**, *double* **s** )

Returns $\mathbf{R} = \frac{1}{s}\mathbf{A}$.

## 4.3 Tensor–tensor multiplications

### 4.3.1 Dot products

*double* **operator∗**( *const Tensor1&* **a**, *const Tensor1&* **b** )

Returns the (scalar) dotproduct $r = \mathbf{a} \cdot \mathbf{b}$ or in index notation $r = a_i b_i$.

*Tensor1* **operator∗**( *const Tensor2&* **A**, *const Tensor1&* **b** )

Returns the dotproduct $\mathbf{r} = \mathbf{A} \cdot \mathbf{b}$ or in index notation $r_i = A_{ij} b_j$.

*Tensor1* **operator∗**( *const Tensor1&* **a**, *const Tensor2&* **B** )

Returns the dotproduct $\mathbf{r} = \mathbf{a} \cdot \mathbf{B}$ or in index notation $r_i = a_k B_{ki}$.

*Tensor2Gen* **operator∗**( *const Tensor2&* **A**, *const Tensor2&* **B** )

Returns the dotproduct $\mathbf{R} = \mathbf{A} \cdot \mathbf{B}$ or in index notation $R_{ij} = A_{ik} B_{kj}$.

*Tensor2Sym* **multAtA**( *const Tensor2&* **A** )

Returns the (symmetric) dotproduct $\mathbf{R} = \mathbf{A}^{\mathrm{T}} \cdot \mathbf{A}$ or in index notation $R_{ij} = A_{ki} A_{kj}$.

*Tensor2Sym* **multAAt**( *const Tensor2&* **A** )

Returns the (symmetric) dotproduct $\mathbf{R} = \mathbf{A} \cdot \mathbf{A}^{\mathrm{T}}$ or in index notation $R_{ij} = A_{ik} A_{jk}$.

### 4.3.2 Double contractions

*double* **doubleContraction**( *const Tensor2&* **A**, *const Tensor2&* **B** )
*double* **doubleContraction**( *const Tensor2Gen&* **A**, *const Tensor2Gen&* **B** )
*double* **doubleContraction**( *const Tensor2Sym&* **A**, *const Tensor2Sym&* **B** )

Returns the double contraction $r = \mathbf{A} : \mathbf{B}$ or in index notation $r = A_{ij} B_{ij}$.

*Tensor2Sym* **doubleContraction**( *const Tensor4DSym&* **A**, *const Tensor2Sym&* **B** )
*Tensor2Sym* **doubleContraction**( *const Tensor4LSym&* **A**, *const Tensor2Sym&* **B** )
*Tensor2Gen* **doubleContraction**( *const Tensor4Gen&* **A**, *const Tensor2Gen&* **B** )
*Tensor2Gen* **operator∗**( *const Tensor4Gen&* **A**, *const Tensor2Gen&* **B** )
*Tensor2Sym* **operator∗**( *const Tensor4DSym&* **A**, *const Tensor2Sym&* **B** )
*Tensor2Sym* **operator∗**( *const Tensor4LSym&* **A**, *const Tensor2Sym&* **B** )

Returns the double contraction $\mathbf{R} = \mathbf{A} : \mathbf{B}$ or in index notation $R_{ij} = A_{ijkl} B_{kl}$.

*Tensor2Gen* **doubleContraction**( *const Tensor2Gen&* **A**, *const Tensor4Gen&* **B** )
*Tensor2Gen* **operator∗**( *const Tensor2Gen&* **A**, *const Tensor4Gen&* **B** )
*Tensor2Sym* **operator∗**( *const Tensor2Sym&* **A**, *const Tensor4DSym&* **B** )

*Tensor2Sym* **operator**$*$( *const Tensor2Sym&* **A**, *const Tensor4LSym&* **B** )

Returns the double contraction $\mathbf{R} = \mathbf{A} : \mathbf{B}$ or in index notation $R_{kl} = A_{ij}B_{ijkl}$. Because of the symmetry in *Tensor4DSym* **a**nd *Tensor4LSym* **,** the **doubleContraction** function $A_{ij}B_{ijkl} = B_{ijkl}B_{kl}$ need not be defined separately.

*Tensor4Gen* **doubleContraction**( *const Tensor4&* **A**, *const Tensor4&* **B** )

Returns the double contraction $\mathbf{R} = \mathbf{A} : \mathbf{B}$ or in index notation $R_{ijkl} = A_{ijmn}B_{mnkl}$.

### 4.3.3 Dyadic and vector products

*Tensor2Gen* **dyadic**( *const Tensor1&* **a**, *const Tensor1&* **b** )

Returns the dyadic product $\mathbf{R} = \mathbf{a} \otimes \mathbf{b}$ or in index notation $R_{ij} = a_i b_j$.

*Tensor2Sym* **selfDyadic**( *const Tensor1&* **a** )

Returns the (symmetric) dyadic product $\mathbf{R} = \mathbf{a} \otimes \mathbf{a}$ or in index notation $R_{ij} = a_i a_j$.

*Tensor4Gen* **dyadic**( *const Tensor2&* **A**, *const Tensor2&* **B** )

Returns the dyadic product $\mathbf{R} = \mathbf{A} \otimes \mathbf{B}$ or in index notation $R_{ijkl} = A_{ij}B_{kl}$.

*Tensor4LSym* **dyadic**( *const Tensor2Sym&* **A**, *const Tensor2Sym&* **B** )

Returns the dyadic product $\mathbf{R} = \mathbf{A} \otimes \mathbf{B}$ or in index notation $R_{ijkl} = A_{ij}B_{kl}$.

*Tensor4DSym* **selfDyadic**( *const Tensor2Sym&* **A** )

Returns the (symmetric) dyadic product $\mathbf{R} = \mathbf{A} \otimes \mathbf{A}$ or in index notation $R_{ijkl} = A_{ij}A_{kl}$.

*Tensor1* **vectorProduct**( *const Tensor1&* **a**, *const Tensor1&* **b** )

Returns the vector product $\mathbf{r} = \mathbf{a} \times \mathbf{b}$.

## 4.4 Tensor functions

*Tensor2Sym* **sym**( *const Tensor2&* **A** )

Returns the symmetric part $\mathbf{R} = \frac{1}{2}(\mathbf{A} + \mathbf{A}^{\mathrm{T}})$.

*Tensor2Gen* **skew**( *const Tensor2&* **A** )

Returns the skew symmetric part $\mathbf{R} = \frac{1}{2}(\mathbf{A} - \mathbf{A}^{\mathrm{T}})$.

*Tensor2Gen* **dev**( *const Tensor2Gen&* **A** )
*Tensor2Sym* **dev**( *const Tensor2Sym&* **A** )

Returns the deviatoric tensor $\mathbf{R} = (\mathbf{I} - \frac{1}{3}\mathbf{1} \otimes \mathbf{1}) : \mathbf{A}$.

*Tensor2Gen* **inv**( *const Tensor2Gen&* **A** )
*Tensor2Sym* **inv**( *const Tensor2Sym&* **A** )

Returns the inverse $\mathbf{R} = \mathbf{A}^{-1}$.

*Tensor2Sym* **square**( *const Tensor2Sym&* **A** )

Returns the square $\mathbf{R} = \mathbf{A} \cdot \mathbf{A}$ as a symmetric tensor.

*void* **findroots**( *const Tensor2Sym&* **S**, *Tensor2Sym∗* **U**, *Tensor2Sym∗* **U_inv** )

Returns the root and inverse root of a symmetric tensor **S**, with $\mathbf{U} \cdot \mathbf{U} = \mathbf{S}$ and $\mathbf{U}^{-1} \cdot \mathbf{U}^{-1} = \mathbf{S}^{-1}$. An algorithm by L.P. Franca, *Computers Math. Applic.*, Vol. 18, pp 459-466, 1989 is used. It is applied in the polar decompositions described in Section 4.5.

*Tensor2Sym* **log**( *const Tensor2Sym&* **A** )

Returns the natural logarithm $\mathbf{R} = \log \mathbf{A}$. The logarithm of a symmetric tensor is calculated by bringing it in a spectral format and taking the logarithms of the eigenvalues.

*double* **norm**( *const Tensor1&* **a** )

Returns the norm $\|\mathbf{a}\| = \sqrt{\mathbf{a} \cdot \mathbf{a}}$ or in index notation $\|a_i\| = \sqrt{a_i a_i}$.

*double* **norm**( *const Tensor2&* **A** )

Returns the norm $\|\mathbf{A}\| = \sqrt{\mathbf{A} : \mathbf{A}}$ or in index notation $\|A_{ij}\| = \sqrt{A_{ij} A_{ij}}$.

*double* **trace**( *const Tensor2&* **A** )

Returns the trace $\mathrm{tr}(\mathbf{A}) = \mathbf{1} : \mathbf{A}$ or in index notation $\mathrm{tr}(A_{ij}) = A_{ii}$.

*double* **det**( *const Tensor2&* **A** )

Returns the determinant $\det(\mathbf{A}) = |\mathbf{A}|$.

## 4.5 Polar decompositions

*void* **polarRight**( *const Tensor2Gen&* **F**, *Tensor2Gen∗* **R**, *Tensor2Sym∗* **U** )

Returns the polar decomposition $\mathbf{F} = \mathbf{R} \cdot \mathbf{U}$ in which **R** is a proper orthogonal tensor and **U** is a symmetric tensor. This function makes use of **findroots**.

*void* **polarLeft**( *const Tensor2Gen&* **F**, *Tensor2Sym∗* **V**, *Tensor2Gen∗* **R** )

Returns the polar decomposition $\mathbf{F} = \mathbf{V} \cdot \mathbf{R}$ in which **V** is a symmetric tensor and **R** is a proper orthogonal tensor. This function makes use of **findroots**.

# Chapter 5

# Extension

In this chapter, the extension functions, included in `xtensor.h` are described. They are included here because they are not general enough to be included in the main files.

## 5.1 Linear algebra algorithms

*Tensor2Sym* **solve**( *const Tensor4DSym&* **A**, *const Tensor2Sym&* **b** )
*Tensor2Sym* **solve**( *const Tensor4LSym&* **A**, *const Tensor2Sym&* **b** )

Solve $\mathbf{x}$ from the system $\mathbf{A} : \mathbf{x} = \mathbf{b}$ where both $\mathbf{x}$ and $\mathbf{b}$ are members of the subspace of symmetric 2nd order tensors. The current implementation of the function uses a QR-decomposition from a general linear algebra package. In a particular crystal plasticity code this performed better than a LU-decomposition. The implementation can, however, be improved, e.g. by using optimized linear algebra packages.

*Tensor2Sym* **deviatoric_solve**( *const Tensor4DSym&* **A**, *const Tensor2Sym&* **b** )
*Tensor2Sym* **deviatoric_solve**( *const Tensor4LSym&* **A**, *const Tensor2Sym&* **b** )

Solve $\mathbf{x}$ from the system $\mathbf{A} : \mathbf{x} = \mathbf{b}$ where both $\mathbf{x}$ and $\mathbf{b}$ are members of the subspace of deviatoric symmetric 2nd order tensors (trace($\mathbf{x}$) = trace($\mathbf{b} = \mathbf{0}$). The condition that $A_{11kl} + A_{22kl} + A_{33kl} = 0$ for all $kl$ is not checked. If this condition is fullfilled, the $6 \times 6$ matrix that is used in the **solve** function would be singular, but it can be solved in this subspace. A separate class for deviatoric symmetric 2nd order tensors should be included in the future to make this transparent. The current implementation of the function uses a QR-decomposition from a general linear algebra package. In a particular crystal plasticity code this performed better than a LU-decomposition. The implementation can, however, be improved, e.g. by using optimized linear algebra packages.

*Tensor4DSym* **inv**( *const Tensor4DSym&* **A** )
*Tensor4LSym* **inv**( *const Tensor4LSym&* **A** )

Return the inverse 4th order tensor, where the 4th order tensor is seen as a mapping from the subspace of symmetric 2nd order tensors to the same subspace. The current implementation of the function uses a standard general linear algebra algorithm for general square matrices. The implementation can, however, be improved, e.g. by using optimized linear algebra packages and an algorithm for symmetric systems in the **Tensor4DSym** case.

## 5.2 Computational mechanics

*Tensor4DSym* **push_forward**( *const Tensor4DSym&* **A**, *const Tensor2Gen&* **F** )
*Tensor4LSym* **push_forward**( *const Tensor4LSym&* **A**, *const Tensor2Gen&* **F** )

Return the 4th order tensor $R_{ijkl} = F_{ip}F_{jq}F_{kr}F_{ls}A_{pqrs}$, known as the push forward operation. If $\mathbf{F}^{-1}$ is used instead of **F** this results in the pull back operation.

*Tensor2Gen* **expW**( *const Tensor2Gen&* **W** )

Returns the exponent of the skew symmetric tensor **W**. If **W** is the skew symmetric part of the velocity gradient times the time increment, this results in an incremental rotation tensor. It is calculated as:

$$\mathbf{R} = \mathbf{1} + \frac{\sin\omega}{\omega}\mathbf{W} + \frac{(1-\cos\omega)}{\omega^2}\mathbf{W}\cdot\mathbf{W} \quad \text{with} \quad \omega = \sqrt{\tfrac{1}{2}\mathbf{W} : \mathbf{W}}$$

It would be more transparent to define a skew symmetric tensor class for this.

# Chapter 6

# Examples

This chapter demonstrates the use of the tensor classes by a number of examples. The examples are contained in the accompanying testfile `basetest.cpp`.

First the assignment of values and printing of some tensors is shown. Then a number of products and tensor functions is demonstrated, without much context. Some large deformation kinematics is shown in the third example, using the inverse function, polar decompositions and the logarithmic function. Finally the construction of some tensors used in elasticity and a workaround for a compiler error is presented

*assignment and printing* *example*

```
cout << setprecision(5);

Tensor1    b;
Tensor2Gen A;

b(1) = 5; b(2) = 6; b(3) = 7;

A(1,1) = 1; A(1,2) = 2; A(1,3) = 3;
A(2,1) = 4; A(2,2) = 5; A(2,3) = 6;
A(3,1) = 7; A(3,2) = 8; A(3,3) = 9;

cout << "b" << endl << setw(12) << b << endl;
cout << "A" << endl << setw(12) << A << endl;
```

*dyadic products, dot products, symmetric, skew and deviatoric parts* *example*

```
Tensor1    a, b;
Tensor2Gen B, C;

a(1) = 1; a(2) = 2; a(3) = 3;
b(1) = 5; b(2) = 6; b(3) = 7;

B = dyadic(a,b);
C = dyadic(b,a);
```

```
cout << "B = dyadic(a,b)" << endl << setw(12) << B << endl;
cout << "C = dyadic(b,a)" << endl << setw(12) << C << endl;

cout << "sym(C) " << endl << setw(12) << sym(C) << endl;
cout << "skew(C) " << endl << setw(12) << skew(C) << endl;
cout << "sym(C) + skew(C) " << endl << setw(12)
     << (sym(C)+skew(C)) << endl;

A = B*C;

cout << "B*C" << endl << setw(12) << A << endl;
cout << "trace(A) " << setw(12) <<  trace(A) << endl;

cout << "dev(A) " << endl << setw(12) << dev(A) << endl;
```

*inverse tensor, polar decompositions and logarithm* *example*

```
Tensor2Gen F;
F(1,1) =  2  ; F(1,2) =  0  ; F(1,3) = 0.5;
F(2,1) = -0.3; F(2,2) =  1  ; F(2,3) = 0.5;
F(3,1) =  0  ; F(3,2) = -0.2; F(3,3) = 0.8;

Tensor2Gen Finv;
Finv = inv(F);
cout << "F"        << endl << setw(12) << F      << endl;
cout << "inv(F)"   << endl << setw(12) << Finv   << endl;
cout << "F*inv(F)" << endl << setw(12) << F*Finv << endl;

double lambda1, lambda2, lambda3;
Tensor2Sym U;
Tensor2Gen R, EVec;

polarRight( F, &R, &U );
cout << "R" << endl << setw(12) << R << endl;
cout << "U" << endl << setw(12) << U << endl;
cout << "R.U" << endl << setw(12) << R*U << endl;

double lambda1, lambda2, lambda3;
Tensor2Gen EVec;
U.eigen( &lambda1, &lambda2, &lambda3, &EVec );

Tensor2Sym F1( lambda1*selfDyadic(EVec(1)) +
               lambda2*selfDyadic(EVec(2)) +
               lambda3*selfDyadic(EVec(3)) );
cout << "spectral composition" << endl
     << setw(12) << F1 << endl;
```

```
Tensor2Sym V;
polarLeft( F, &V, &R );
cout << "R"   << endl << setw(12) << R   << endl;
cout << "V"   << endl << setw(12) << V   << endl;
cout << "V.R" << endl << setw(12) << V*R << endl;

cout << "log(V)" << endl << setw(12) << log(V) << endl;
```

The isotropic elastic stiffness tensor can be written as

$$\mathbf{E} = 2G\mathbf{H} + \left(\kappa - \tfrac{2}{3}G\right)\mathbf{I} \otimes \mathbf{I}$$

where $\mathbf{H}$ is the fourth order symmetric identity tensor, $\mathbf{I}$ the second order identity tensor, $G$ the shear modulus and $\kappa$ the compressibility modulus. Given a strain tensor, the stress tensor can be derived as $\boldsymbol{\sigma} = \mathbf{E} : \boldsymbol{\varepsilon}$ but also as

$$\boldsymbol{\sigma} = 2G\mathbf{e} + \kappa\,\mathrm{tr}(\boldsymbol{\varepsilon})\mathbf{I}$$

where $\mathbf{e}$ is the deviatoric strain tensor. This can be implemented as follows:

*elasticity equations* *example*

```
Tensor4DSym H(1);  // fourth order symmetric identity tensor
Tensor2Sym  I(1);  // second order identity tensor

double  E, nu, G, K, p;

E=200000.0; nu=0.3;

G = E/(2*(1+nu));
K = E/(3*(1-2*nu));

Tensor4DSym E4, II;
Tensor2Sym sigma, eps, edev;

II = selfDyadic(I);

E4 = (2*G)*H + (K-2*G/3)*II;

cout << "H"  << endl << setw(12) << H << endl;
cout << "E4" << endl << setw(12) << E4 << endl;

eps(1,1) = 0.001;
eps(2,2) = -0.0003;
eps(3,3) = -0.0003;
eps(2,1) = 0.002;     // only the lower left part needs
eps(3,1) = 0.004;     // to be initialized
eps(3,2) = -0.015;
```

```
cout << "eps" << endl << setw(12) << eps << endl;

sigma = E4*eps;
cout << "sigma 1" << endl << setw(12) << sigma << endl;

edev = dev(eps);

p = -K*trace(eps);
sigma = (2*G)*edev + -p*I;
cout << "sigma 2" << endl << setw(12) << sigma << endl;
```

On some compilers an ambiguity arises because e.g. the double contraction between a symmetric and a general second order tensor can choose between the function using base classes **Tensor2** or by converting the symmetric tensor to a general second order tensor and using the function for **Tensor2Gen**. This is believed to be a compiler error. It can be solved by typecasting as follows:

*example*

```
Tensor2Gen sigma(1);
Tensor2Sym sigmas(1);

doubleContraction( sigma, sigma );   // general function
doubleContraction( sigmas, sigmas ); // symmetric function

doubleContraction( sigma, sigmas ); // error on some compilers

doubleContraction( (const Tensor2&)sigma,
                   (const Tensor2&)sigmas );
                                     // base class function

doubleContraction( sigma, (Tensor2Gen)sigmas );
                                     // general function
```

# Index