MMTensor package documentation

Maarten Moesen

September 9, 2011

1 Introduction

1.1 What is MMTensor?

MMTensor is a collection of documented `MATLAB` scripts for micromechanical computations. It contains methods for basic computations with tensors in matrix notation, for computing and analyzing fabric tensors, and for (limited) inclusion modeling. This package should be viewed as a 'getting started' guide for micromechanical modeling. Great care was taken to implement the methods in a well-documented, easily understandable and still efficient form. The package mainly provides basic functionality. It was never our aim to be comprehensive or exhaustive.

1.2 Literature and background information

A practical description of the matrix notation used in this package and connections (isomorphisms) between tensor and matrix notation can be found in the accompanying document `tensor_representation.pdf`. This documents explains the differences with conventional matrix notation due to Voigt (1928) at a basic level. More fundamental treatments can be found in the literature.

The matrix notation used in MMTensor is in principle due to Lord Kelvin (Thomson 1856, Thomson 1878), but was expressed in contemporary linear algebra notation by Rychlewski (1984) and Mehrabadi & Cowin (1990). The latter publication and (Cowin & Mehrabadi 1992) discuss the matrix notation thoroughly and present in detail the eigenvalues and eigentensors of an elasticity tensor for different types of material symmetry. A general introduction into material symmetry is (Nye 1984). The tensor concepts and tools (such as representation surfaces) presented in this work using Voigt's matrix notation are easily transferrable to Kelvin's notation.

Kanatani (1984) and Advani & Tucker (1987) provide a framework for computing fabric tensors. A general representation of the relationship between fabric tensor and elasticity tensor is given by Cowin (1985). Moesen, Cardoso & Cowin (2011) present a practical formulation of this theory, allowing to express the material in terms of 2, 5 or 9 independent parameters, depending on the material symmetry. All proposed methods for computing the parameters numerically are implemented in this library.

The methods for inclusion modeling were largely based on the book of Mura (1987). The self-consistent model was originally due to Hill (1965) and the Mori-Tanaka scheme due to Mori & Tanaka (1973). Particularly interesting insights in the latter theory were provided by Benveniste (1987) and Weng (1990).

1.3 Licensing and acknowledging

MMTensor is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or any later version. This license guarantees that other people have the freedom and right to use and learn from this work in the same way that you are using it and learning from it right now.

If you're using the software or the ideas behind it for research purposes, then please refer in your publications to the articles from the literature list that are relevant for your research. Note that this is not an obligation from my side, but it is simply proper publishing ethics. If you're really happy about these tools, feel free to make an acknowledgment in your publication, but note that this does not replace a proper citation.

1.4 Before you start...

It is considered a prerequisite that you are familiar with tensor algebra and the matrix notation used in MMTensor. If you're not, please consult the literature in section 1.2.

Before you start I really advise you to get acquainted with `MATLAB`. To my opinion, `MATLAB` is very well-documented software. The documentation is clear and to-the-point. Therefore, if you're not already familiar with `MATLAB`, it is really worthwhile to go once through the demos, tutorials and if necessary the reference documentation.

Only two actions are to be taken before use:

1. Unzip the contents of this package to a folder on your computer. Although this can be any folder, it is advisable to use your home directory or the folder in which you store your other `MATLAB` work.

2. Add the `MMTensor` folder and its subfolders to `MATLAB`'s path. For example by using `File > Set Path... > Add with Subfolders...`. There is no need to add the doc subfolder as it does not contain `MATLAB` scripts.

2 Basic functionality

The `tensor` folder contains `MATLAB` functions for generic tensor operations. Reference information regarding the explicit usage of a function will not be given here, but can be easily obtained by entering `help <function_name>` at the `MATLAB` command line or by consulting the function's source code. What will be given here is an overview of the provided functionality. To get you started, section 7 will walk through the methods example by example.

The methods `makeS2` and `makeS4` create a second and a fourth order symmetric tensor in matrix notation. Depending on the arguments you specify, the tensor will be a zero or an identity tensor. The methods `makeIsotropicStiffnessTensor` and `makeTransversallyIsotropicStiffnessTensor` create an elasticity tensor based on engineering constants. The method `makeProjectors` creates the spherical and deviatoric projection tensors. The method `computeEngineeringConstantsSqrt2` computes the (orthogonal) engineering constants from a given elasticity tensor.

Specific elements of a tensor can be obtained and set by the methods `getS2Element`, `getS4Element`, `setS2Element` and `setS4Element`. These methods automatically

take care of the appropriate factors $\sqrt{2}$ and 2 to facilitate the conversion between tensor and matrix notation. Low-level methods providing the factors and indices for this conversion are `getSFactor` and `getSIndex`. The higher level methods `makeS2Index` and `makeS4Index` generate matrices containing the factors.

Several methods convert between other representations. For example, `matrix_to_vector_notation` and `vector_to_matrix_notation` convert orthogonal fourth order tensors between matrix and vector notation (Moesen et al. 2011). `makeMatrixFromS2` and `makeS2FromVector` interchange between tensorial matrix notation and conventional vectors and matrices.

The method `principal_axes` computes the eigenvalues of a symmetric second order tensor in ascending order. The corresponding eigenvectors constitute a right-handed axis system.

Methods `makeAngleAxisRotation` and `makeEulerRotation` create rotation matrices. Using `makeTensorTransformationMatrix` these rotation matrices can be converted to matrix notation for rotating second and fourth order tensors.

Given a rotation matrix, `rotateS4` rotates a fourth order tensor. `invertS4` inverts a fourth order tensor. `symmetrizeS4` computes the fully symmetric part of a fourth order tensor. `neglectNonorthotropicCoefficients` zeroes all terms that should be zero for a orthogonal elasticity tensor.

3 Fabric tensors

3.1 Fabric tensor computation and testing

The `fabric` folder contains `MATLAB` functions for computing the three kinds of fabric tensors based on directional data. The concepts and methods underlying the three kinds of fabric tensors are described by Kanatani (1984). Briefly, the first kind of fabric tensor is a weighted moment tensor of a directional data set. The fabric tensor of the second kind is the best tensorial approximation in the least squares sense of the directional distribution function. Decomposing second kind fabric tensors into an orthogonal series results in fabric tensors of the third kind.

The method `fabric_moment_tensor` computes from directional data a second and fourth order moment tensor. From these moment tensors, the fabric tensors of the second kind can be computed using `fabric_tensor`, or the fabric tensors of the third kind using `fabric_decomposition`. Given third kind fabric tensors, the second kind fabric tensors can be computed using `fabric_tensor_from_decomposition`. Third kind fabric tensors can also be used for testing for uniformity of the directional data set by `fabric_fitness_test`.

The method `eval_fabric_distribution_function` allows integrating the distribution density function defined by a second kind fabric tensor in a specified range of angles.

3.2 Relations between fabric tensor and elasticity tensor

The `fabric` folder also contains `MATLAB` functions for analyzing and modeling the relation between the fabric tensor and the elasticity tensor.

A typical first step in an analysis is rotating a fabric tensor and its corresponding elasticity tensor to its principal axis system, which can be done with the methods `principal_axes`, `makeTensorTransformationMatrix` and `rotateS4`, discussed in section 2.

Based on the eigenvalues of the fabric tensor, the methods `orthogonal_basis_analytical` and `orthogonal_basis_numerical` compute an orthogonal basis in vector notation, as well as the squared norms and coefficient matrices corresponding to the basis. These can be used in method `orthogonal_coefficients` to compute for a given elasticity tensor the parameters p_i. The parameters a_i, b_i and c_i as defined by Cowin (1985) can consequently be determined using the method `original_coefficients`. Based on the parameters p_i and the orthogonal basis, the predicted elasticity tensor can be reconstructed in matrix notation by the method `predicted_tensor_orthogonal_coefficients`.

The methods `cowin_coefficients_cowins_method` and `cowin_coefficients_moesens_method` compute for a fabric tensor and stiffness tensor the parameters a_i, b_i and c_i as defined by Cowin (1985). These methods only work for the case that the three eigenvalues of the fabric tensor are distinct.

4 Inclusion modeling

The `inclusions` folder provides basic methods for inclusion modeling. `makeSpheroidalEshelby` computes Eshelby's tensor for axially symmetric ellipsoidal inclusions in an isotropic matrix. `makeEshelby` computes Eshelby's tensor for general ellipsoidal inclusions in an isotropic matrix. The method `computeMoriTanakaAligned` computes the elasticity tensor for a mixture of aligned inclusions with given concentrations and inclusion stiffness tensors using Mori-Tanaka's method. `computeSelfConsistentAligned` does the same using the self-consistent method.

5 Utility functions

`euclidean_norm` computes the euclidean norm of the rows or columns of a matrix. `cell_functional` is convenient for extracting a matrix of values from a cell array consisting of similar vectors or matrices.

6 Graphical functions

The method `plotRepresentationSurface` plots for a given elasticity tensor its representation surface. A representation surface plots on a rose diagram for every direction the apparent Young's modulus in that direction. The method `plotDirectionalRoseDiagrams` takes a directional data set as input. It plots the star diagram of the directions, and the representation surfaces of second and fourth order moment and fabric tensors.

These graphical functions should be regarded as examples rather than actual functions. You can add your own customization and add bells and whistles.

7 Example walkthrough

```
%% Create a zero symmetric second order tensor
t_zero = makeS2(0)
% t_zero =
%      0
%      0
%      0
%      0
```

```
%       0
%       0

%% Create an identity second order tensor
t_identity = makeS2(1)
% t_identity =
%       1
%       1
%       1
%       0
%       0
%       0

%% Create based on an existing tensor in conventional matrix format
F = [ 0.3291    -0.0592     0.0422;
     -0.0592     0.2990     0.0816;
      0.0422     0.0816     0.3718]

F_s2 = makeS2(F)
% F_s2 =
%      0.3291
%      0.2990
%      0.3718
%      0.1154
%      0.0597
%     -0.0837

%% Convert back to matrix format. Should be the symmetric part of F
F_matrix = makeMatrixFromS2(F_s2)
% F_matrix =
%      0.3291    -0.0592     0.0422
%     -0.0592     0.2990     0.0816
%      0.0422     0.0816     0.3718

%% Manipulate individual elements with getS2Element() and setS2Element.
getS2Element(F_s2,2,2)
% ans =
%      0.2990

F_manipulated = setS2Element(F_s2,2,1,1)
% F_manipulated =
%      0.3291
%      0.2990
%      0.3718
%      0.1154
%      0.0597
%      1.4142

getS2Element(F_manipulated,1,2)
% ans =
%       1

%% Retrieve the sqrt2 factors for a S2 tensor
makeS2Index
% ans =
%      1.0000
%      1.0000
%      1.0000
%      1.4142
%      1.4142
%      1.4142
```

```
%% Create a zero fourth order tensor
T_zero = makeS4(0)
% T_zero =
%      0     0     0     0     0     0
%      0     0     0     0     0     0
%      0     0     0     0     0     0
%      0     0     0     0     0     0
%      0     0     0     0     0     0
%      0     0     0     0     0     0

%% Create the identity fourth order tensor
T_identity = makeS4(1)
% T_identity =
%      1     0     0     0     0     0
%      0     1     0     0     0     0
%      0     0     1     0     0     0
%      0     0     0     1     0     0
%      0     0     0     0     1     0
%      0     0     0     0     0     1

%% Create the sqrt2 factors for a fourth order tensor
makeS4Index
% ans =
%    1.0000    1.0000    1.0000    1.4142    1.4142    1.4142
%    1.0000    1.0000    1.0000    1.4142    1.4142    1.4142
%    1.0000    1.0000    1.0000    1.4142    1.4142    1.4142
%    1.4142    1.4142    1.4142    2.0000    2.0000    2.0000
%    1.4142    1.4142    1.4142    2.0000    2.0000    2.0000
%    1.4142    1.4142    1.4142    2.0000    2.0000    2.0000


%% Create the spherical and deviatoric projection tensor
[T_sph, T_dev] = makeProjectors
% T_sph =
%    0.3333    0.3333    0.3333         0         0         0
%    0.3333    0.3333    0.3333         0         0         0
%    0.3333    0.3333    0.3333         0         0         0
%         0         0         0         0         0         0
%         0         0         0         0         0         0
%         0         0         0         0         0         0
% T_dev =
%    0.6667   -0.3333   -0.3333         0         0         0
%   -0.3333    0.6667   -0.3333         0         0         0
%   -0.3333   -0.3333    0.6667         0         0         0
%         0         0         0    1.0000         0         0
%         0         0         0         0    1.0000         0
%         0         0         0         0         0    1.0000

%% Verify the following expressions:
% T_sph : T_sph = T_sph
% T_sph : T_dev = 0
% T_dev : T_sph = 0
% T_dev : T_dev = T_dev

T_sph*T_sph
% ans =
%    0.3333    0.3333    0.3333         0         0         0
%    0.3333    0.3333    0.3333         0         0         0
%    0.3333    0.3333    0.3333         0         0         0
%         0         0         0         0         0         0
%         0         0         0         0         0         0
%         0         0         0         0         0         0
```

```
T_dev*T_dev
% ans =
%
%      0.6667   -0.3333   -0.3333         0         0         0
%     -0.3333    0.6667   -0.3333         0         0         0
%     -0.3333   -0.3333    0.6667         0         0         0
%           0         0         0    1.0000         0         0
%           0         0         0         0    1.0000         0
%           0         0         0         0         0    1.0000

T_sph*T_dev
% ans =
%     1.0e-16 *
%      0.2776    0.2776    0.2776         0         0         0
%      0.2776    0.2776    0.2776         0         0         0
%      0.2776    0.2776    0.2776         0         0         0
%           0         0         0         0         0         0
%           0         0         0         0         0         0
%           0         0         0         0         0         0

T_dev*T_sph
% ans =
%     1.0e-16 *
%      0.2776    0.2776    0.2776         0         0         0
%      0.2776    0.2776    0.2776         0         0         0
%      0.2776    0.2776    0.2776         0         0         0
%           0         0         0         0         0         0
%           0         0         0         0         0         0
%           0         0         0         0         0         0

%% Create the isotropic stiffness tensor [in GPa] of a Ti6Al4V alloy.
%% Note: as in most computer programs, units are implicitly assumed.
C_Ti6Al4V = makeIsotropicStiffnessTensor(114, 0.34)
% C_Ti6Al4V =
%    175.4664   90.3918   90.3918         0         0         0
%     90.3918  175.4664   90.3918         0         0         0
%     90.3918   90.3918  175.4664         0         0         0
%           0         0         0   85.0746         0         0
%           0         0         0         0   85.0746         0
%           0         0         0         0         0   85.0746

%% Create an transversely anisotropic stiffness tensor [in GPa].
%% The stiffness tensor corresponds to that of a hexagonal honeycomb
%% made from Al (E=70 GPa, nu=0.3) with a relative density of 10%.
C_honey = makeTransversallyIsotropicStiffnessTensor(...
  6.7896, 0.0958, 3.3950, 0.0092, 0.004234)
% C_honey =
%     -0.0057   -0.0241   -0.0090         0         0         0
%     -0.0241   -0.0057   -0.0090         0         0         0
%     -0.0090   -0.0090    6.7842         0         0         0
%           0         0         0    6.7900         0         0
%           0         0         0         0    6.7900         0
%           0         0         0         0         0    0.0184

%% Verify the engineering constants. Note that the in-plane properties are
%% much lower than the out-of-plane properties, as expected.
[E_eng_honey,nu_eng_honey] = computeEngineeringConstantsSqrt2(C_honey)
% E_eng_honey =
%      0.0958    0.0958    6.7896    3.3950    3.3950    0.0092
% nu_eng_honey =
%      0.0042    0.0042    4.2065    0.3001    0.3001    4.2065
```

```
%% Plot the representation surface of the stiffness tensors:
plotRepresentationSurface(C_Ti6Al4V);
% An isotropic material should give a sphere.

plotRepresentationSurface(C_honey);
% A honeycomb typically is relatively compliant in its transversal plane.

%% Compute the corresponding compliance tensor.
S_honey = invertS4(C_honey)
% S_honey =
%    10.4384  -43.9094   -0.0442         0         0         0
%   -43.9094   10.4384   -0.0442         0         0         0
%    -0.0442   -0.0442    0.1473         0         0         0
%         0         0         0    0.1473         0         0
%         0         0         0         0    0.1473         0
%         0         0         0         0         0   54.3478

%% Manipulate individual elements using getS4Element and setS4Element
G_12 = 1./getS4Element(S_honey,1,2,1,2)
% G_12 =
%    0.0368

S_honey = setS4Element(S_honey,2,1,1,2,0.1)
% S_honey =
%    10.4384  -43.9094   -0.0442         0         0         0
%   -43.9094   10.4384   -0.0442         0         0         0
%    -0.0442   -0.0442    0.1473         0         0         0
%         0         0         0    0.1473         0         0
%         0         0         0         0    0.1473         0
%         0         0         0         0         0    0.2000

G_12 = 1./getS4Element(S_honey,1,2,2,1)
% G_12 =
%    10

%% Make rotation matrices
R_AA = makeAngleAxisRotation(45, [0 0 1])
% R_AA =
%    0.7071   -0.7071         0
%    0.7071    0.7071         0
%         0         0    1.0000

R_euler = makeEulerRotation(pi/4, 0, 0)
% R_euler =
%    0.7071   -0.7071         0
%    0.7071    0.7071         0
%         0         0    1.0000

%% Create a set with 500 uniformly oriented unit vectors
n_500 = randn(3,500);
n_500 = n_500./repmat(euclidean_norm(n_500), 3, 1);

%% Compute the second and fourth order moment tensor
[N2, N4] = fabric_moment_tensor(n_500)
% N2 =
%    0.3371
%    0.3341
%    0.3288
%    0.0236
%    0.0237
%    0.0090
```

```
% N4 =
%     0.2032    0.0680    0.0659   -0.0004    0.0116    0.0052
%     0.0680    0.2007    0.0654    0.0109    0.0030    0.0062
%     0.0659    0.0654    0.1976    0.0132    0.0091   -0.0024
%    -0.0004    0.0109    0.0132    0.1308   -0.0035    0.0042
%     0.0116    0.0030    0.0091   -0.0035    0.1317   -0.0006
%     0.0052    0.0062   -0.0024    0.0042   -0.0006    0.1360
% Observe that these tensors approximate closely the isotropic case.
% You can enlarge the number of vectors to see convergence (which is slow).

%% Plot the directional distribution
plotDirectionalRoseDiagrams(n_500);
% This function should be seen as an example of how plots can be generated.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% So far we've just been warming up. Let's do some real tensor analysis.%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


%% Recall the earlier defined fabric tensor F_s2.
%% Actually this tensor corresponds to a tetrahedral structure.
M2_tet = F_s2; % M stands for moment tensor, as that's what it is.

M4_tet = [ 0.1909    0.0621    0.0761    0.0127    0.0267   -0.0381
           0.0621    0.1653    0.0717    0.0417   -0.0019   -0.0369
           0.0761    0.0717    0.2240    0.0610    0.0349   -0.0087
           0.0127    0.0417    0.0610    0.1434   -0.0123   -0.0027
           0.0267   -0.0019    0.0349   -0.0123    0.1523    0.0180
          -0.0381   -0.0369   -0.0087   -0.0027    0.0180    0.1241];

%% The corresponding stiffness tensor is:
C_tet = [ 1.5645    0.5380    0.5062    0.1023    0.1004   -0.3332;
          0.5380    1.4107    0.5239    0.2623   -0.0838   -0.3209;
          0.5062    0.5239    1.6149    0.3320    0.1037   -0.0499;
          0.1023    0.2623    0.3320    1.0538   -0.1329   -0.0401;
          0.1004   -0.0838    0.1037   -0.1329    1.0542    0.1576;
         -0.3332   -0.3209   -0.0499   -0.0401    0.1576    1.0850];

%% Compute the principal axes of the second order fabric tensor
[Vv_tet, Ev_tet] = principal_axes(M2_tet)
% Vv_tet =
%     0.5019    0.8636   -0.0478
%     0.7216   -0.4486   -0.5273
%    -0.4768    0.2302   -0.8483
% Ev_tet =
%     0.2039
%     0.3711
%     0.4249
%
% Note that the eigenvalues are in ascending order and that Vv_tet is a
% righthanded axis system.

%% Rotate C_tet to the principal axis system.
C_tet = rotateS4(C_tet, Vv_tet)
% C_tet =
%     0.9612    0.4102    0.4343    0.0367   -0.0009   -0.0197
%     0.4102    1.8707    0.6201    0.1795   -0.0186    0.0125
%     0.4343    0.6201    1.9651    0.1249    0.0243    0.0128
%     0.0367    0.1795    0.1249    1.3570   -0.0131   -0.0315
%    -0.0009   -0.0186    0.0243   -0.0131    0.8052    0.0161
%    -0.0197    0.0125    0.0128   -0.0315    0.0161    0.8239
%
```

```
% Note that the first three diagonal terms are now in ascending order, and
% that the 'non-orthogonal' terms are smaller.

%% Let's analyze the fabric tensor. Compute the second kind fabric tensor
F2_tet = fabric_tensor(M2_tet)
% F2_tet =
%     0.9682
%     0.7425
%     1.2885
%     0.8655
%     0.4476
%    -0.6279

%% Similar but also computing fourth order tensors.
[F2_tet, F4_tet] = fabric_tensor(M2_tet, M4_tet)
% F2_tet =
%     0.9682
%     0.7425
%     1.2885
%     0.8655
%     0.4476
%    -0.6279
%
% F4_tet =
%     0.7544    0.3204    0.5564   -0.0038    0.2693   -0.4023
%     0.3204    0.5349    0.5126    0.1287   -0.3362   -0.3534
%     0.5564    0.5126    0.9363    0.8863    0.5893    0.0252
%    -0.0038    0.1287    0.8863    1.0251    0.0356   -0.4755
%     0.2693   -0.3362    0.5893    0.0356    1.1129   -0.0054
%    -0.4023   -0.3534    0.0252   -0.4755   -0.0054    0.6407

%% Note that the eigenvalues of the second kind fabric tensor are
%% different, but that the eigenvectors remain the same.
[Vv_tet2, Ev_tet2] = principal_axes(F2_tet)
% Vv_tet2 =
%     0.5019    0.8636   -0.0478
%     0.7216   -0.4486   -0.5273
%    -0.4768    0.2302   -0.8483
% Ev_tet2 =
%     0.0292
%     1.2832
%     1.6868

%% The second order tensor represents the statistical distribution of
%% material orientations. Compute the probability that a material element
%% makes an angle of at most 30 degrees with the horizontal plane
eval_fabric_distribution_function(F2_tet, [0 pi], [0, 2*pi])
% ans =
%     0.9998
% Over the entire interval the function should integrate to 1.

eval_fabric_distribution_function(F2_tet, [pi/3 2*pi/3], [0, 2*pi])
% ans =
%     0.4457

%% Similar for the fourth order tensor
eval_fabric_distribution_function(F4_tet, [0 pi], [0, 2*pi])
% ans =
%     1.0009
% Over the entire interval the function should integrate to 1.

eval_fabric_distribution_function(F4_tet, [pi/3 2*pi/3], [0, 2*pi])
```

```
% ans =
%      0.4255

%% Now compute the third kind fabric tensors.
[D2_tet, D4_tet] = fabric_decomposition(M2_tet, M4_tet)
% D2_tet =
%     -0.0318
%     -0.2575
%      0.2885
%      0.8655
%      0.4476
%     -0.6279
%
% D4_tet =
%     -0.2138    0.0352    0.1803   -0.1480    0.0455   -0.0883
%      0.0352   -0.2076    0.1741   -0.3041   -0.4108   -0.0394
%      0.1803    0.1741   -0.3522    0.4536    0.3655    0.1298
%     -0.1480   -0.3041    0.4536    0.3481    0.1836   -0.5810
%      0.0455   -0.4108    0.3655    0.1836    0.3606   -0.2094
%     -0.0883   -0.0394    0.1298   -0.5810   -0.2094    0.0705

%% Again the eigenvectors are the same, but the eigenvalues differ.
[Vv_tet3, Ev_tet3] = principal_axes(D2_tet)
% Vv_tet3 =
%      0.5019    0.8636   -0.0478
%      0.7216   -0.4486   -0.5273
%     -0.4768    0.2302   -0.8483
%
% Ev_tet3 =
%     -0.9708
%      0.2832
%      0.6868
% As expected, the eigenvalues sum to zero.

%% The orthogonal decomposition allows reconstructing the second kind FT
[F2d_tet] = fabric_tensor_from_decomposition(D2_tet)
% F2d_tet =
%      0.9682
%      0.7425
%      1.2885
%      0.8655
%      0.4476
%     -0.6279

[F4d_tet] = fabric_tensor_from_decomposition(D2_tet, D4_tet)
% F4d_tet =
%      0.7544    0.3204    0.5564   -0.0038    0.2693   -0.4023
%      0.3204    0.5349    0.5126    0.1287   -0.3362   -0.3534
%      0.5564    0.5126    0.9363    0.8863    0.5893    0.0252
%     -0.0038    0.1287    0.8863    1.0251    0.0356   -0.4755
%      0.2693   -0.3362    0.5893    0.0356    1.1129   -0.0054
%     -0.4023   -0.3534    0.0252   -0.4755   -0.0054    0.6407

%% The orthogonal decomposition also allows testing the fitness of
%% using a uniform distribution.
[p2, p4] = fabric_fitness_test(129, D2_tet, D4_tet)
% 129 was the number of independent data points in the tetrahedral structure.
% p_2 < 0.005 indicates a significant difference from a uniform distribution,
% so consider D2
% For testing second order fitness, D2(norm=1.2224) should be small in
% comparison to 1
% p_4 < 0.005 indicates that fourth order terms should be considered.
```

```
% p2 =
%     2.6247e-06
% p4 =
%      0.3167
% Because of the low value of p2, the fabric tensor is not uniform.
% From p4 there is no real proof that D4_tet is not uniform. Since D2_tet
% is non-uniform, the resulting tensor F4 will also be non-uniform.

%% So far the methods for computing all kinds of fabric tensors. Let's now
%% focus on the link between fabric tensor and elasticity tensor.

%% C_tet was already rotated to its principal axis system.
%% The eigenvalues in this system were Ev_tet.
%% Based on these eigenvalues we can compute an orthogonal basis

[Q, N, R, INVARIANTS] = orthogonal_basis_analytical(Ev_tet)
% Q =
%      1.0000     1.3333    -1.9412    -1.2941     0.5323     0.1510    -0.1068
0.0033          0
%      1.0000     1.3333     0.5670     0.3780    -0.0574    -0.6204    -0.2017
0.0560          0
%      1.0000     1.3333     1.3742     0.9161     0.4218     0.4694     0.1792
0.0320          0
%      1.4142    -0.9428     1.3726    -1.8302     0.1187    -0.1068    -0.0596
-0.0599          0
%      1.4142    -0.9428    -0.4010     0.5346    -0.7153     0.4387    -0.1938
0.0146          0
%      1.4142    -0.9428    -0.9717     1.2956    -0.0375    -0.3319     0.3448
-0.0193          0
%           0     2.0000          0     1.9412     0.0168          0    -0.1772
-0.0755    -0.1510
%           0     2.0000          0    -0.5670    -0.3911          0     0.1074
-0.0173     0.6204
%           0     2.0000          0    -1.3742    -0.5224          0     0.1990
0.0014    -0.4694
%
%
% N =
%      9.0000
%     20.0000
%      8.9673
%     13.9491
%      1.4179
%      0.9420
%      0.3268
%      0.0143
%      0.6280
%
%
% R =
%      1.0000     0.6667          0          0          0     0.9964          0
0.2482     0.6642
%           0     1.0000          0          0     0.1495          0    -0.1134
0.0372     0.9964
%           0          0     1.0000     1.3333          0    -0.3795     0.4982
-0.1891    -0.5060
%           0          0          0     1.0000    -0.1627          0     0.2135
-0.0405    -0.3795
%           0          0          0          0     1.0000          0    -0.6150
0.0836     0.4429
%           0          0          0          0          0     1.0000          0
0.4982     1.3333
```

```
%         0         0         0         0         0         0    1.0000
-0.4416    1.1819
%         0         0         0         0         0         0         0
1.0000   -2.6764
%         0         0         0         0         0         0         0
0    1.0000
%
%
% INVARIANTS =
%     0.0000   -0.7473   -0.1891
%
% Note that the first Invariant (I) is zero as expected.

%% The orthogonal_basis_numerical function should give the same results
[Q, N, R, INVARIANTS] = orthogonal_basis_numerical(Ev_tet)
% Q =
%     1.0000    1.3333   -1.9412   -1.2941    0.5323    0.1510   -0.1068
0.0033    0.0000
%     1.0000    1.3333    0.5670    0.3780   -0.0574   -0.6204   -0.2017
0.0560    0.0000
%     1.0000    1.3333    1.3742    0.9161    0.4218    0.4694    0.1792
0.0320    0.0000
%     1.4142   -0.9428    1.3726   -1.8302    0.1187   -0.1068   -0.0596
-0.0599   -0.0000
%     1.4142   -0.9428   -0.4010    0.5346   -0.7153    0.4387   -0.1938
0.0146    0.0000
%     1.4142   -0.9428   -0.9717    1.2956   -0.0375   -0.3319    0.3448
-0.0193   -0.0000
%         0    2.0000   -0.0000    1.9412    0.0168    0.0000   -0.1772
-0.0755   -0.1510
%         0    2.0000   -0.0000   -0.5670   -0.3911         0    0.1074
-0.0173    0.6204
%         0    2.0000   -0.0000   -1.3742   -0.5224         0    0.1990
0.0014   -0.4694
%
% N =
%     9.0000
%    20.0000
%     8.9673
%    13.9491
%     1.4179
%     0.9420
%     0.3268
%     0.0143
%     0.6280
%
% R =
%     1.0000    0.6667    0.0000    0.0000   -0.0000    0.9964    0.0000
0.2482    0.6642
%         0    1.0000    0.0000         0    0.1495         0   -0.1134
0.0372    0.9964
%         0         0    1.0000    1.3333    0.0000   -0.3795    0.4982
-0.1891   -0.5060
%         0         0         0    1.0000   -0.1627         0    0.2135
-0.0405   -0.3795
%         0         0         0         0    1.0000   -0.0000   -0.6150
0.0836    0.4429
%         0         0         0         0         0    1.0000    0.0000
0.4982    1.3333
%         0         0         0         0         0         0    1.0000
-0.4416    1.1819
%         0         0         0         0         0         0         0
```

```
1.0000   -2.6764
%         0         0         0         0         0         0         0
0    1.0000
%
% INVARIANTS =
%    0.0000   -0.7473   -0.1891

%% Q are the basis tensors in vector notation, and N are their squared norms.
%% These allow determining the orthogonal parameters can be determined for
%% a stiffness tensor as follows:
oc_tet = orthogonal_coefficients(C_tet, Q, N)
% oc_tet =
%    0.8585
%    0.5208
%    0.2552
%    0.1279
%    0.1083
%   -0.1166
%   -0.2724
%    0.0669
%   -0.1467

%% Reconstructing the stiffness tensor using oc_tet goes as follows
C_pred = predicted_tensor_orthogonal_coefficients(oc_tet,Q)
% C_pred =
%    0.9612    0.4102    0.4343         0         0         0
%    0.4102    1.8707    0.6201         0         0         0
%    0.4343    0.6201    1.9651         0         0         0
%         0         0         0    1.3570         0         0
%         0         0         0         0    0.8052         0
%         0         0         0         0         0    0.8239

%% Which is identical to the one obtained when zeroing the 'non-orthogonal'
%% elements
neglectNonorthotropicCoefficients(C_tet)
% ans =
%    0.9612    0.4102    0.4343         0         0         0
%    0.4102    1.8707    0.6201         0         0         0
%    0.4343    0.6201    1.9651         0         0         0
%         0         0         0    1.3570         0         0
%         0         0         0         0    0.8052         0
%         0         0         0         0         0    0.8239

%% Using the R matrix it is possible to obtain Cowin's (1985) coefficients
[cc_original_order, cc_orthogonal_order] = original_coefficients(oc_tet, R)
% cc_original_order =
%    0.3671
%    0.1730
%    0.2412
%    0.0512
%   -0.2428
%   -0.3257
%    0.6438
%    0.1192
%   -0.1467
% cc_orthogonal_order =
%    0.3671
%    0.6438
%    0.1730
%    0.1192
%    0.0512
%    0.2412
```

```
%    -0.2428
%    -0.3257
%    -0.1467
%% The first order is the one as was proposed by Cowin.

%% There are two methods for directly obtaining Cowin's coefficients:
%% Note that we use the eigenvalues of the orthogonal decomposition
%% Differences between these and cc_original order result from slight
%% differences in the eigenvalues.
cc_cowin = cowin_coefficients_cowins_method(C_tet, Ev_tet3)
% cc_cowin =
%     0.3672
%     0.1732
%     0.2412
%     0.0510
%    -0.2431
%    -0.3258
%     0.6439
%     0.1192
%    -0.1467
cc_moesen = cowin_coefficients_moesens_method(C_tet, Ev_tet3)
% cc_moesen =
%     0.3672
%     0.1732
%     0.2412
%     0.0510
%    -0.2431
%    -0.3258
%     0.6439
%     0.1192
%    -0.1467

%% However, to my (Moesen's) opinion, the orthogonal parameters are the ones
%% to be preferred. Especially in case of two or three equal eigenvalues.
%% For example:
[Q, N, R, INVARIANTS] = orthogonal_basis_analytical([1/4 1/2 1/4])
%
% Q =
%    1.0000    1.3333   -1.2500   -0.8333    0.3013
%    1.0000    1.3333    2.5000    1.6667    0.8036
%    1.0000    1.3333   -1.2500   -0.8333    0.3013
%    1.4142   -0.9428    0.8839   -1.1785   -0.5682
%    1.4142   -0.9428   -1.7678    2.3570    0.1421
%    1.4142   -0.9428    0.8839   -1.1785   -0.5682
%         0    2.0000         0    1.2500   -0.8036
%         0    2.0000         0   -2.5000    0.2009
%         0    2.0000         0    1.2500   -0.8036
%
% N =
%    9.0000
%   20.0000
%   14.0625
%   21.8750
%    2.8251
%
% R =
%    1.0000    0.6667         0         0         0
%         0    1.0000         0         0    0.2344
%         0         0    1.0000    1.3333         0
%         0         0         0    1.0000    0.2679
%         0         0         0         0    1.0000
%
```

```
%
% INVARIANTS =
%            0   -1.1719    0.4883

%% As you can see the rank with two equal eigenvalues is limited to 5.
%% If you don't like this you can add false as a second argument.
[Q, N, R, INVARIANTS] = orthogonal_basis_analytical([1/4 1/2 1/4], false)
%
% Q =
%     1.0000    1.3333   -1.2500   -0.8333    0.3013         0
0         0         0
%     1.0000    1.3333    2.5000    1.6667    0.8036         0
0         0         0
%     1.0000    1.3333   -1.2500   -0.8333    0.3013         0
0         0         0
%     1.4142   -0.9428    0.8839   -1.1785   -0.5682         0
0         0         0
%     1.4142   -0.9428   -1.7678    2.3570    0.1421         0
0         0         0
%     1.4142   -0.9428    0.8839   -1.1785   -0.5682         0
0         0         0
%          0    2.0000         0    1.2500   -0.8036         0         0
0         0
%          0    2.0000         0   -2.5000    0.2009         0         0
0         0
%          0    2.0000         0    1.2500   -0.8036         0         0
0         0
%
%
% N =
%     9.0000
%    20.0000
%    14.0625
%    21.8750
%     2.8251
%          0
%          0
%          0
%          0
%
%
% R =
%     1.0000    0.6667         0         0         0         0         0
0         0
%          0    1.0000         0         0    0.2344         0         0
0         0
%          0         0    1.0000    1.3333         0         0         0
0         0
%          0         0         0    1.0000    0.2679         0         0
0         0
%          0         0         0         0    1.0000         0         0
0         0
%          0         0         0         0         0         0         0
0         0
%          0         0         0         0         0         0         0
0         0
%          0         0         0         0         0         0         0
0         0
%          0         0         0         0         0         0         0
0         0
%
% INVARIANTS =
```

```
%         0   -1.1719    0.4883

%% A similar rank reduction to two in the isotropic case
[Q, N, R, INVARIANTS] = orthogonal_basis_analytical([1/3 1/3 1/3])
% Q =
%    1.0000    1.3333
%    1.0000    1.3333
%    1.0000    1.3333
%    1.4142   -0.9428
%    1.4142   -0.9428
%    1.4142   -0.9428
%         0    2.0000
%         0    2.0000
%         0    2.0000
% N =
%     9
%    20
% R =
%    1.0000    0.6667
%         0    1.0000
% INVARIANTS =
%     0     0     0

%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% INCLUSION MODELING %%
%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% This package also provides basic functions for inclusion modeling. Just
%% to show how such models can be implemented in an elegant, practical way.

%% Let's start with defining some constants
E_matrix = 3; % [GPa]
E_glass = 70; % [GPa]
nu_matrix = 0.3;
nu_glass = 0.27;

% Some aspect ratios
aspect_glass = 10;
aspect_holes = 10;
aspect_ellipsoids = [0.25 1.2 3]; % E.g. the three radii of an ellipsoid.

%% Computing Eshelby's tensor for spheroidal inclusions goes as follows:
S_spd = makeSpheroidalEshelby(aspect_glass, nu_glass)
% S_sph =
%    0.6656    0.0161    0.1705         0         0         0
%    0.0161    0.6656    0.1705         0         0         0
%   -0.0031   -0.0031    0.0416         0         0         0
%         0         0         0    0.4889         0         0
%         0         0         0         0    0.4889         0
%         0         0         0         0         0    0.6495

%% Computing the same Eshelby's tensor using a method for more general
%% ellipsoidal inclusions should yield identical results:
S_spd2 = makeEshelby([1 1 aspect_glass], nu_glass)
% S_spd2 =
%    0.6656    0.0161    0.1705         0         0         0
%    0.0161    0.6656    0.1705         0         0         0
%   -0.0031   -0.0031    0.0416         0         0         0
%         0         0         0    0.4889         0         0
%         0         0         0         0    0.4889         0
%         0         0         0         0         0    0.6495
```

```
%% Computing Eshelby's tensor for ellipsoidal inclusions:
S_ell = makeEshelby(aspect_ellipsoids, nu_glass)
% S_ell =
%     0.9139    0.2098    0.2716         0         0         0
%    -0.0291    0.2567    0.0439         0         0         0
%    -0.0096    0.0017    0.0807         0         0         0
%          0         0         0    0.1704         0         0
%          0         0         0         0    0.7938         0
%          0         0         0         0         0    0.7845


%% Now estimate the stiffness tensor of a composite material with 30% short
%% unidirectionally aligned glass fibers and 2% of spherical pores.
concentrations = [0.30 0.02];
C_inclusions = zeros(6,6,2);
C_inclusions(:,:,1) = makeIsotropicStiffnessTensor(E_glass,nu_glass);
aspect_ratios = [aspect_glass; aspect_holes];

[C_mt, C_matrix] = computeMoriTanakaAligned(...
  E_matrix, nu_matrix, concentrations, C_inclusions, aspect_ratios)
% C_mt =
%     5.9924    2.4218    2.4812         0         0         0
%     2.4218    5.9924    2.4812         0         0         0
%     2.4111    2.4111   15.9247         0         0         0
%          0         0         0    3.9421         0         0
%          0         0         0         0    3.9421         0
%          0         0         0         0         0    3.5706
% C_matrix =
%     4.0385    1.7308    1.7308         0         0         0
%     1.7308    4.0385    1.7308         0         0         0
%     1.7308    1.7308    4.0385         0         0         0
%          0         0         0    2.3077         0         0
%          0         0         0         0    2.3077         0
%          0         0         0         0         0    2.3077

%% Note that the stiffness increases in particular in the fiber direction.
%% A version of the self-consistent method was also implemented.
%% (not 100% correct, see source)
C_sc = computeSelfConsistentAligned(E_matrix, nu_matrix, ...
  concentrations, C_inclusions, aspect_ratios, 100000, 1e-6)
%Convergence: absolute error after 11930 iterations: 9.99972e-07
%C_sc =
%     5.3373    1.8854    1.6631         0         0         0
%     1.8854    5.3373    1.6631         0         0         0
%     1.7664    1.7664   15.3065         0         0         0
%          0         0         0    4.1358         0         0
%          0         0         0         0    4.1358         0
%          0         0         0         0         0    3.4519

%% Recall that the engineering constants are retrieved by
[E_mt, nu_mt] = computeEngineeringConstantsSqrt2(C_mt)
%E_mt =
%    4.8713    4.8713   14.5024    1.9711    1.9711    1.7853
%
%nu_mt =
%    0.0977    0.0977    0.3643    0.2907    0.2907    0.3643
%
% When changing the aspect ratio of the phases it is possible that
% the MT method gives asymmetric stiffness tensors. This was described by
% Benveniste (1987). For solutions for mixing phases or orientations see
% e.g. the publications by Doghri and Tinel (2005-2006).
```

References

Advani, S. & Tucker, C. (1987), 'The use of tensors to describe and predict fiber orientation in short fiber composites', *Journal of Rheology* **31**(8), 751–784.

Benveniste, Y. (1987), 'A new approach to the application of mori-tanaka theory in composite-materials', *Mechanics of Materials* **6**(2), 147–157.

Cowin, S. C. (1985), 'The relationship between the elasticity tensor and the fabric tensor', *Mechanics of Materials* **4**, 137–147.

Cowin, S. & Mehrabadi, M. (1992), 'The structure of the linear anisotropic elastic symmetries', *Journal of the Mechanics and Physics of Solids* **40**(7), 1459–1471.

Hill, R. (1965), 'A self-consistent mechanics of composite materials', *Journal of the Mechanics and Physics of Solids* **13**(4), 213–&.

Kanatani, K. (1984), 'Distribution of directional-data and fabric tensors', *International Journal of Engineering Science* **22**(2), 149–164.

Mehrabadi, M. & Cowin, S. (1990), 'Eigentensors of linear anisotropic elastic-materials', *Quarterly Journal of Mechanics and Applied Mathematics* **43**, 15–41.

Moesen, M., Cardoso, L. & Cowin, S. C. (2011), 'A symmetry-invariant formulation of the relationship between the elasticity tensor and the fabric tensor', *(to be submitted)* .

Mori, T. & Tanaka, K. (1973), 'Average stress in matrix and average elastic energy of materials with misfitting inclusions', *Acta Metallurgica* **21**(5), 571–574.

Mura, T. (1987), *Micromechanics of defects in solids*, Vol. 3, 2nd, rev. ed edn, M. Nijhoff, Dordrecht, Netherlands.

Nye, J. F. (1984), *Physical properties of crystals: their representation by tensors and matrices*, 1st published in pbk. with corrections, 1984 edn, Clarendon Press, Oxford.
URL: *http://www.loc.gov/catdir/enhancements/fy0603/84019026-d.html*

Rychlewski, J. (1984), 'On hooke law', *Pmm Journal of Applied Mathematics and Mechanics* **48**(3), 303–314.

Thomson, W. (1856), 'Elements of a mathematical theory of elasticity', *Phil. Trans. R. Soc.* **166**, 481–498.

Thomson, W. (1878), *Elasticity*, Adam and Charles Black, Edinburgh.

Voigt, W. (1928), *Lehrbuch der Kristallphysik*, Teubner, Leibzig.

Weng, G. (1990), 'The theoretical connection between mori tanaka theory and the hashin shtrikman walpole bounds', *International Journal of Engineering Science* **28**(11), 1111–1120.