FEM tools for elasticity in three dimensions

Coordinator	:	Francisco–Javier Sayas (University of Delaware)
Developers	:	Thomas Brown (UD, 2015-2018)
		Matthew Hassell (UD, 2015-2016)
		Allan Hungria (UD, 2015-)
		Tianyu Qiu (UD, 2015-2016)
		Tonatiuh Sánchez-Vizuet (UD, 2015-2016)
		Hasan Eruslu (UD, Spring 2016-)
		Shukai Du (UD, Spring 2016-)
		Ben Civiletti (UD, Fall 2016)
		Jake Jacavage (UD, Fall 2016)
Collaborators	:	Kevin Cotter (UD, Summer 2015)
		Jaspal Nijjar (UD, Summer 2015)

Last update: July 4, 2019

Contents

1 Linear elasticity **2**
- 1.1 Stiffness and mass matrices . . . 2
- 1.2 Steady state solver . . . 4
- 1.3 A discontinuous error function . . . 6

2 Viscoelasticity **7**
- 2.1 Fractional Zener model and time domain solver . . . 7
- 2.2 Time domain elasticity . . . 11
- 2.3 Transfer function for Laplace domain . . . 12
- 2.4 Creating an exact solution for fractional Zener model . . . 14

3 Tools for time domain problems **17**
- 3.1 Convolution quadrature methods . . . 17
- 3.2 CQ with multistep methods . . . 19
- 3.3 CQ with Runge-Kutta method . . . 20
- 3.4 Usage of CQ operators . . . 23
- 3.5 Smooth window function . . . 23
- 3.6 Creating a plane wave . . . 24

4 FEM computation of stresses **26**
- 4.1 Norm of the stress tensor at the barycenters . . . 26
- 4.2 Stress postprocessing . . . 29

5 Plotting utilities **32**
- 5.1 Plotting mesh . . . 32
- 5.2 Plotting the solution on the volume mesh . . . 33
- 5.3 Plotting and refining on the boundary of the mesh . . . 40

6 Needs checking **46**
6.1 Piezo elasticity 46

7 Appendix **50**
7.1 Mittag-Leffler function 50

Foreword (for internal use). This is a 'column code,' which means that geometric information is typically stored in matrices where each column corresponds to a geometric element: the coordinates of a point, the vertices of a triangle. Some of the team's recent codes are 'row codes.' It happens that access to information is very often more efficient in column codes and several annoying transpositions are avoided.

The code described in this document uses the functions in FEM3D folder. The functions that are designed for solving elasticity related problems or visualizing their solutions are presented here. For others we refer to FEM3DDocumentation file.

The code assumes the latest version of MATLAB. For example in recent functions we use overloaded algebraic operators instead of `bsxfun`.

Some calculations are kept in script format. These scripts can be run in other scripts. In that case the parameters defined as input in those scripts should be defined before running that script. Explain this more ....

1 Linear elasticity

c:elasticity

1.1 Stiffness and mass matrices

In this section we show some functions to prepare matrices and solvers for different problems in linear elasticity. We first give some general explanations about the general bilinear form for isotropic but possible heterogeneous elasticity. The associated bilinear form is

$$a(\mathbf{u},\mathbf{v}) = \int_\Omega \mu(\nabla\mathbf{u} + \nabla\mathbf{u}^\top) : \nabla\mathbf{v} + \lambda(\nabla\cdot\mathbf{u})(\nabla\cdot\mathbf{v})$$

where $\mathbf{u},\mathbf{v} : \Omega \to \mathbb{R}^3$ are componentwise H^1 vector fields and $\mu,\lambda : \Omega \to \mathbb{R}$ are variable Lamé parameters. We look at the components of $\mathbf{u} = (u,v,w) \in V_h^3$ and test with functions where only one component is not zero. The first 'scalar' variational equation we obtain is

$$\begin{aligned}
a(\mathbf{u},(\omega,0,0)) =& \int_\Omega \mu \begin{bmatrix} 2u_x & u_y+v_x & u_z+w_x \\ v_x+u_y & 2v_y & v_z+w_y \\ w_x+u_z & w_y+v_z & 2w_z \end{bmatrix} : \begin{bmatrix} \omega_x & \omega_y & \omega_z \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} + \lambda(u_x+v_y+w_z)\omega_x \\
=& 2\int_\Omega \mu u_x\omega_x + \int_\Omega \mu u_y\omega_y + \int_\Omega \mu v_x\omega_y + \int_\Omega \mu u_z\omega_z + \int_\Omega \mu w_x\omega_z \\
&+ \int_\Omega \lambda u_x\omega_x + \int_\Omega \lambda v_y\omega_x + \int_\Omega \lambda w_z\omega_x \\
=& \left(2\int_\Omega \mu u_x\omega_x + \int_\Omega \lambda u_x\omega_x + \int_\Omega \mu u_y\omega_y + \int_\Omega \mu u_z\omega_z\right) \\
&+ \left(\int_\Omega \lambda v_y\omega_x + \int_\Omega \mu v_x\omega_y\right) + \left(\int_\Omega \lambda w_z\omega_x + \int_\Omega \mu w_x\omega_z\right).
\end{aligned}$$

The second one is

$$\begin{aligned} a(\mathbf{u},(0,\omega,0)) &= \int_\Omega \mu \begin{bmatrix} 2u_x & u_y+v_x & u_z+w_x \\ v_x+u_y & 2v_y & v_z+w_y \\ w_x+u_z & w_y+v_z & 2w_z \end{bmatrix} : \begin{bmatrix} 0 & 0 & 0 \\ \omega_x & \omega_y & \omega_z \\ 0 & 0 & 0 \end{bmatrix} + \lambda(u_x+v_y+w_z)\omega_y \\ &= \left(\int_\Omega \mu u_y\omega_x + \int_\Omega \lambda u_x\omega_y\right) + \left(2\int_\Omega \mu v_y\omega_y + \int_\Omega \mu v_x\omega_x + \int_\Omega \lambda v_y\omega_y + \int_\Omega \mu v_z\omega_z\right) \\ &\quad + \left(\int_\Omega \lambda w_z\omega_y + \int_\Omega \mu w_y\omega_z\right), \end{aligned}$$

and finally we have

$$\begin{aligned} a(\mathbf{u},(0,0,\omega)) &= \int_\Omega \mu \begin{bmatrix} 2u_x & u_y+v_x & u_z+w_x \\ v_x+u_y & 2v_y & v_z+w_y \\ w_x+u_z & w_y+v_z & 2w_z \end{bmatrix} : \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ \omega_x & \omega_y & \omega_z \end{bmatrix} + \lambda(u_x+v_y+w_z)\omega_z \\ &= \left(\int_\Omega \mu u_z\omega_x + \int_\Omega \lambda u_x\omega_z\right) + \left(\int_\Omega \mu v_z\omega_y + \int_\Omega \lambda v_y\omega_z\right) \\ &\quad + \left(2\int_\Omega \mu w_z\omega_z + \int_\Omega \mu w_x\omega_x + \int_\Omega \mu w_y\omega_y + \int_\Omega \lambda w_z\omega_z\right). \end{aligned}$$

The elastic stiffness matrix. At the time of discretizing the three components of $\mathbf{u}$ are discretized as separate unknowns. Therefore the $3\dim V_h \times 3\dim V_h$ matrix corresponding to the bilinear form a is set as a 3×3 block matrix, whose blocks are of $\dim V_h \times \dim V_h$ size. Using `stiffnessMatrices3D` twice (once with all six coefficients equal to μ and the second time with all coefficients equal to λ. If we have the matrices sorted as components of a cell array (the output of `stifnessMatrices3D`)

$$\begin{cases} \mathtt{Sm}\{1,1\} \equiv \int_\Omega \mu P_x^j P_x^i \\ \mathtt{Sm}\{1,2\} \equiv \int_\Omega \mu P_y^j P_x^i \\ \mathtt{Sm}\{1,3\} \equiv \int_\Omega \mu P_z^j P_x^i \\ \mathtt{Sm}\{2,2\} \equiv \int_\Omega \mu P_y^j P_y^i \\ \mathtt{Sm}\{2,3\} \equiv \int_\Omega \mu P_z^j P_y^i \\ \mathtt{Sm}\{3,3\} \equiv \int_\Omega \mu P_z^j P_z^i \end{cases} \quad \text{and} \quad \begin{cases} \mathtt{Sl}\{1,1\} \equiv \int_\Omega \lambda P_x^j P_x^i \\ \mathtt{Sl}\{1,2\} \equiv \int_\Omega \lambda P_y^j P_x^i \\ \mathtt{Sl}\{1,3\} \equiv \int_\Omega \lambda P_z^j P_x^i \\ \mathtt{Sl}\{2,2\} \equiv \int_\Omega \lambda P_y^j P_y^i \\ \mathtt{Sl}\{2,3\} \equiv \int_\Omega \lambda P_z^j P_y^i \\ \mathtt{Sl}\{3,3\} \equiv \int_\Omega \lambda P_z^j P_z^i \end{cases}$$

then

$$\begin{bmatrix} 2\mathtt{Sm}\{1,1\}+\mathtt{Sl}\{1,1\} & \mathtt{Sl}\{1,2\}+\mathtt{Sm}\{1,2\}^{\mathtt{T}} & \mathtt{Sl}\{1,3\}+\mathtt{Sm}\{1,3\}^{\mathtt{T}} \\ +\mathtt{Sm}\{2,2\}+\mathtt{Sm}\{3,3\} & & \\ \mathtt{Sm}\{1,2\}+\mathtt{Sl}\{1,2\}^{\mathtt{T}} & 2\mathtt{Sm}\{2,2\}+\mathtt{Sm}\{1,1\} & \mathtt{Sl}\{2,3\}+\mathtt{Sm}\{2,3\}^{\mathtt{T}} \\ & +\mathtt{Sl}\{2,2\}+\mathtt{Sm}\{3,3\} & \\ \mathtt{Sm}\{1,3\}+\mathtt{Sl}\{1,3\}^{\mathtt{T}} & \mathtt{Sm}\{2,3\}+\mathtt{Sl}\{2,3\}^{\mathtt{T}} & 2\mathtt{Sm}\{3,3\}+\mathtt{Sm}\{1,1\} \\ & & +\mathtt{Sm}\{2,2\}+\mathtt{Sl}\{3,3\} \end{bmatrix}$$

is the global stiffness matrix for the elasticity bilinear form.

The vector mass matrix. In time-harmonic and dynamical elasticity problem, the mass bilinear form

$$\int_\Omega \rho\mathbf{u}\cdot\mathbf{v}$$

makes an appearance. This is an easy matrix to compute by making three copies of a scalar mass matrix

$$\mathrm{M} = \left[\int_\Omega \rho P_i P_j\right]_{i,j=1}^{\dim V_h},$$

and placing them on the diagonals of a 3×3 block matrix

$$\begin{bmatrix} \mathrm{M} & \mathrm{O} & \mathrm{O} \\ \mathrm{O} & \mathrm{M} & \mathrm{O} \\ \mathrm{O} & \mathrm{O} & \mathrm{M} \end{bmatrix}.$$

1.2 Steady state solver

sec:1.2

The function `FEMelasticity3D.m` computes the V_h approximation to the linear elasticity problem

$$\begin{aligned} -\operatorname{div} \boldsymbol{\sigma} + s^2 \rho \, \mathbf{u} &= \mathbf{f} && \text{in } \Omega, \\ \boldsymbol{\sigma} &= \mu \boldsymbol{\varepsilon}(\mathbf{u}) + \lambda (\nabla \cdot \mathbf{u}) \mathrm{I} && \text{in } \Omega, \\ \mathbf{u} &= \mathbf{u}_D && \text{on } \Gamma_D, \\ \boldsymbol{\sigma} \, \boldsymbol{\nu} &= \boldsymbol{\sigma}_N \, \boldsymbol{\nu} && \text{on } \Gamma_N. \end{aligned}$$

The coefficients are the following: s is a complex parameter ($s = -\imath \, \omega$ corresponds to time-harmonic elasticity, while $s = 0$ corresponds to quasi-static elasticity); ρ is a strictly positive function (density); λ and μ are functions. The way the function is built $\boldsymbol{\sigma}_N$ is input as a cell array with six functions from which the symmetric tensor is derived in the following form

$$\boldsymbol{\sigma}_N = \begin{bmatrix} \sigma_1 & \sigma_2 & \sigma_3 \\ \sigma_2 & \sigma_4 & \sigma_5 \\ \sigma_3 & \sigma_5 & \sigma_6 \end{bmatrix}.$$

Note that this is not Voigt's ordering (which will be used for the output), but the ordering that is consistent with the Neumann tester `NeumannBC3D.m`.
The process is carried out as follows:

- We first compute the three load vectors

  $$\int_\Omega f_x \varphi_i, \qquad \int_\Omega f_y \varphi_i, \qquad \int_\Omega f_z \varphi_i, \qquad i = 1, \ldots, \dim V_h,$$

  using `loadVector3D.m`

- We then compute the traction vectors

  $$\int_{\Gamma_N} (\sigma_{xx}, \sigma_{xy}, \sigma_{xz}) \cdot \boldsymbol{\nu} \varphi_i, \quad \int_{\Gamma_N} (\sigma_{yx}, \sigma_{yy}, \sigma_{yz}) \cdot \boldsymbol{\nu} \varphi_i, \quad \int_{\Gamma_N} (\sigma_{zx}, \sigma_{zy}, \sigma_{zz}) \cdot \boldsymbol{\nu} \varphi_i, \qquad i = 1, \ldots, \dim V_h.$$

- We then compute the $3 \dim V_h \times 3 \dim V_h$ stiffness matrix S and the associated mass matrix M. This process was detailed at the beginning of this section (Section 1). The combined matrix

  $$\mathrm{MAT} := \mathrm{S} + s^2 \, \mathrm{M}$$

  is then produced.

- In the next step, we compute a $\dim V_h \times 3$ matrix whose columns are the Dirichlet approximations of the three components of $\mathbf{u}_D$. Note that this is automatically done by `DirichletBC.m` when the input is a 3-cell array with function handles. Each of these columns has zeros in the free nodes and values in the Dirichlet nodes. As a reminder, the approximation is done with a global $L^2(\Gamma_D)$ orthogonal projection onto the space $V_h|_{\Gamma_D}$. This function also provides the list of free nodes (in V_h).

- The right-hand-side is then computed

  $$\mathrm{RHS} := \begin{bmatrix} \mathbf{f}_x + \mathbf{t}_x \\ \mathbf{f}_y + \mathbf{t}_y \\ \mathbf{f}_z + \mathbf{t}_z \end{bmatrix} - \mathrm{MAT} \begin{bmatrix} \mathbf{u}_{D,x} \\ \mathbf{u}_{D,y} \\ \mathbf{u}_{D,z} \end{bmatrix}$$

- The list of free nodes in the space $V_h \times V_h \times V_h$ is then computed.
- We then solve in the free nodes

$$\mathrm{MAT}_{\mathrm{Free},\mathrm{Free}}\mathbf{U}_{\mathrm{Free}} = \mathrm{RHS}_{\mathrm{Free}}.$$

  The solution vector $\mathbf{U}$ (of size $3 \dim V_h$) is separated into three $\dim V_h$ vectors corresponding to the approximations of the three components of the displacement vector.

- Finally, `stressPostprocessing.m` is used to produce $\mathbb{P}_{k-1}$ discontinuous approximations of the six components of the stress tensor.

```
function [u1h,u2h,u3h,sxxh,syyh,szzh,syzh,sxzh,sxyh]...
                  = FEMelasticity3D(mu,lam,rho,f,uD,sigma,T,k,s)
% [u1h,u2h,u3h,sxxh,syyh,szzh,syzh,sxzh,sxyh]...
%                   = FEMelasticity3D(mu,lam,rho,{fx,fy,fz},{u,v,w},...
%                                     {sxx,sxy,sxz,syy,syz,szz},T,k,s)
% Input:
% mu,lam, rho    : Vectorized function handles of three variables
% fx, fy, fz     : Vectorized functions of three variables (load vector)
% ux, uy, uz     : Vectorized functions of three variables (Dirichlet BC)
% sigxx,...      : Six vectorized functions of three variables
% T              : Data structure. Basic FEM triangulation
% k              : scalar. Polynomial degree
% s              : Complex wave number (s=-1i k)
%
% Output:
% u1h,u2h,u3h    : Ndof x 1 Vectors. Vh coefficients of the displacement
%                  components.
% sxxh,...       : dim P_{k-1} times N_elts vectors.  Coefficients of the DG
%                  stress post-processed using the FEM displacement solution
%
% Last modified: November 21, 2017

T = edgesAndFaces(T);
T = enhanceGrid3D(T);

fh =loadVector3D(f,T,k);
fx = fh(:,1);
fy = fh(:,2);
fz = fh(:,3);

traction=neumannBC3D(sigma,T,k);
tx=traction(:,1);
ty=traction(:,2);
tz=traction(:,3);

Sm = stiffnessMatrices3D(mu,T,k);
Sl = stiffnessMatrices3D(lam,T,k);

MAT=[2*Sm{1,1}+Sl{1,1}+Sm{2,2}+Sm{3,3}  Sl{1,2}+Sm{1,2}.'   Sl{1,3}+Sm{1,3}.';...
     Sm{1,2}+Sl{1,2}.'  2*Sm{2,2}+Sm{1,1}+Sl{2,2}+Sm{3,3}   Sl{2,3}+Sm{2,3}.';...
     Sl{1,3}.'+Sm{1,3}  Sl{2,3}.'+Sm{2,3}  2*Sm{3,3}+Sm{1,1}+Sm{2,2}+Sl{3,3}];

Mh = massMatrix3D(rho,T,k);
O=sparse(size(Mh,1),size(Mh,2));
MAT=MAT+s^2*[Mh O O;...
             O Mh O;...
             O O Mh];

[uh,~,free] = dirichletBC3D(uD,T,k);
uh=uh(:);
RHS = [fx+tx;fy+ty;fz+tz]-MAT*uh;
```

```
Ndof = dimFEMspace(T,k);
free = [free(:); Ndof+free(:);2*Ndof+free(:)];
uh(free) = MAT(free,free)\RHS(free);
u1h = reshape(uh,[Ndof,3]);
u2h = u1h(:,2);
u3h = u1h(:,3);
u1h(:,2:3) = [];

[sxxh,syyh,szzh,syzh,sxzh,sxyh]=stressPostprocessing(u1h,u2h,u3h,lam,mu,T,k);

end
```

1.3 A discontinuous error function

The function `errorDFEM3D.m` is similar to `errorFEM3D.m` (see Section 5.8 the error function in FEM3DDocumentation), except that the computed solution is now assumed to be a discontinuous solution calculated locally on the degrees of freedom for each element. CAUTION: this function was written with `stressPostprocessing.m` in mind (see Section Stress Postprosessing). This means that the function takes as an input a polynomial degree of k, but then uses polynomials of degree $k-1$ to compute the L^2 error. Given a function u, and a discrete function u^h (in the space $\prod_K \mathbb{P}_{k-1}(K)$, input as a column vector), we compute

$$\int_\Omega |u-u^h| = \sum_K \int_K |u-u^h|^2,$$

which we approximate using quadrature as

$$\sum_K \sum_q |K||u_K(\mathbf{p}_q) - u_K^h(\mathbf{p}_q)|^2.$$

```
function eh=errorDFEM3D(u,uh,T,k)

% eh=errorDFEM3D(u,uh,T,k)
%
% Input:
%    u    : vectorized function of three variables
%    uh   : dim P_k-1 X NElts vector for a discontinuous function
%    T    : basic triangulation
%    k    : polynomial degree of continuous space
%           CAUTION: we really need k-1, but we input k!
% Output:
%    eh   : L2 error
%
% Last modified: July 8, 2016

T=edgesAndFaces(T);
T=enhanceGrid3D(T);

formula=quadratureFEM(4*k-4,3);
x=T.coordinates(1,:); x=formula(:,1:4)*x(T.elements);
y=T.coordinates(2,:); y=formula(:,1:4)*y(T.elements);
z=T.coordinates(3,:); z=formula(:,1:4)*z(T.elements);
U=u(x,y,z);

Nelts = size(T.elements,2);
dkM1 = size(uh,1)/Nelts;
Uh=reshape(uh, dkM1, Nelts);
uh=bernstein3D(formula(:,2),formula(:,3),formula(:,4),k-1)*Uh;

eh=sqrt(formula(:,5)'*abs(U-uh).^2*T.volume.');
return
```

2 Viscoelasticity

2.1 Fractional Zener model and time domain solver

sec:2.1

The function `FEMViscoelasticity3D.m` computes the FEM approximation of the viscoelastic wave propagation problem with Fractional Zener model: for $t \geq 0$

eq:2.0

$$-\operatorname{div} \boldsymbol{\sigma}(t) + \rho\, \ddot{\mathbf{u}}(t) = \mathbf{f}(t) \qquad \text{in } \Omega, \tag{1a}$$

$$\mathbf{u}(t) = \mathbf{u}_D(t) \qquad \text{on } \Gamma_D, \tag{1b}$$

$$\boldsymbol{\sigma}(t)\, \boldsymbol{\nu} = \boldsymbol{\sigma}_N(t)\, \boldsymbol{\nu} \qquad \text{on } \Gamma_N, \tag{1c}$$

where the stress-strain relation can be written in Laplace domain as

$$\mathbf{S}(s) = \mu(s)\boldsymbol{\varepsilon}(\mathbf{U}(s)) + \lambda(s)(\nabla \cdot \mathbf{U}(s))\mathrm{I} \qquad \forall s \in \mathbb{C}_+ := \{s \in \mathbb{C} : \mathrm{Re}\, s > 0\}, \tag{1d}$$

{eq:2.0d}

with

$$\mu(s) = \frac{m_\mu + b_\mu s^{\nu_\mu}}{1 + a s^{\nu_\mu}}, \qquad \lambda(s) = \frac{m_\lambda + b_\lambda s^{\nu_\lambda}}{1 + a s^{\nu_\lambda}} \tag{1e}$$

for $m_\mu, b_\mu, \nu_\mu, m_\lambda, b_\lambda, \nu_\lambda \in L^\infty(\Omega)$ strictly positive satisfying

$$a m_\mu \leq b_\mu, \qquad a m_\lambda \leq b_\lambda.$$

Here we assume homogeneous initial conditions

$$\mathbf{u}(0) = \mathbf{0}, \qquad \dot{\mathbf{u}}(0) = \mathbf{0}. \tag{1f}$$

The function `FEMViscoelasticity3D.m` works as follows

- Sample the forcing term, Dirichlet and Neumann data at the time steps. This is done in a parallel loop using the helper function `sampleData` (see the note at the end of this section). Depending on the time stepping method, we sample a little differently:

  (a) **Sampling for the CQ method.** We create $3N_{\mathrm{dof}} \times 1$ vectors

  $$\mathbf{f}^n, \quad \mathbf{u}^n_{h,D}, \quad \mathbf{t}^n, \qquad n = 1, \ldots, N_t$$

  corresponding to load, Dirichlet and Neumann approximations at time steps $t_n = n\Delta$ (uniform time steps). Then we combine all these samples and construct the $6N_{\mathrm{dof}} \times N_t$ matrix X in the following way

  $$\mathrm{X} = \left[\begin{array}{c|c|c|c} & & & \\ X_1 & X_2 & \cdots & X_{N_t} \\ & & & \end{array}\right],$$

  where each column is

  $$X_n = \begin{bmatrix} \mathbf{u}^n_{h,D} \\ \mathbf{f}^n + \mathbf{t}^n \end{bmatrix} \qquad n = 1, \ldots, N_t.$$

  (b) **Sampling for the RKCQ method.** We run a parallel loop over time steps $n = 1, \ldots, N_t - 1$, and a serial loop over stages $\ell = 1, \ldots, S$ to create $3N_{\mathrm{dof}} \times 1$ vectors

  $$\mathbf{f}^{n,\ell}, \quad \mathbf{u}^{n,\ell}_{h,D}, \quad \mathbf{t}^{n,\ell},$$

corresponding to load, Dirichlet and Neumann approximations at time steps

$$t_{n,\ell} = (n + c_\ell)\Delta, \qquad n = 1, \dots, N_t, \quad \ell = 1, \dots, S.$$

Here the index ℓ corresponds to the variable `ns` in the code. The vector $\mathbf{c}$, on the other hand, is from the Butcher tableau and equal to the row sums of the $S \times S$ matrix $\mathbf{A}$ (see Section 3.3 for details). This matrix is an optional input if one wants to use Runge-Kutta as time-stepping method.

Now we first create $6N_{\mathrm{dof}} \times N_t$ matrices

$$\mathrm{X}^\ell = \left[\begin{array}{c|c|c|c} & & & \\ X_1^\ell & X_2^\ell & \cdots & X_{N_t}^\ell \\ & & & \end{array}\right] \qquad \ell = 1, \dots, S$$

where each column is

$$X_n^\ell = \begin{bmatrix} \mathbf{u}_{h,D}^{n,\ell} \\ \mathbf{f}^{n,\ell} + \mathbf{t}^{n,\ell} \end{bmatrix} \qquad n = 1, \dots, N_t, \quad \ell = 1, \dots, S.$$

Then we vertically stack matrices X^ℓ for $\ell = 1, \dots, S$

$$\mathrm{X} = \begin{bmatrix} \mathrm{X}^1 \\ \vdots \\ \mathrm{X}^S \end{bmatrix},$$

to obtain a single $6N_{\mathrm{dof}}S \times N_t$ matrix.

- Using `TFFEMelasticity3D.m`, we construct the transfer function $\mathcal{F}_h$ (`Transfer` in the code) in the Laplace domain such that

$$\mathcal{F}_h(s, \mathbf{u}_D, \mathbf{f}, \boldsymbol{\sigma}_N) = (\mathbf{u}^h, \boldsymbol{\sigma}^h)$$

is FEM approximation of

$$\begin{aligned} -\operatorname{div} \boldsymbol{\sigma} + s^2 \rho\, \mathbf{u} &= \mathbf{f} && \text{in } \Omega, \\ \mathbf{u} &= \mathbf{u}_D && \text{on } \Gamma_D, \\ \boldsymbol{\sigma}\, \boldsymbol{\nu} &= \boldsymbol{\sigma}_N\, \boldsymbol{\nu} && \text{on } \Gamma_N. \end{aligned}$$

This solver is same as the solver of `FEMelasticity3D.m`.

- Using the transfer function $\mathcal{F}_h$ and sampled data X we perform an all-at-once time stepping. This is done via one of the following, where time stepping method is specified by the input `p`.
  (a) A multistep method with the function `CQoperator.m`. In this case `p` should be given as a function handle.
  (b) A Runge-Kutta method with the function `RKCQoperator.m`. In this case `p` should be given as a matrix corresponding to $\mathbf{A}$ in the Butcher tableau.

  If time stepping method is not specified we use trapezoidal rule as default.

- We finally output the FEM approximation of displacement and post-processed stress

$$\mathbf{u}_h^n, \quad \boldsymbol{\sigma}_h^n$$

at the time steps t_n for $n = 1, \cdots, N$. For the stress part, we use `stressPostprocessing.m` to output

(a) Either $\mathbb{P}_{k-1}$ discontinuous approximations of the stress,

(b) Or averaged stress for each element,

These options are determined by whether the input variable `flag` is 0 or 1.

Note on the sampling data with the helper function. The second function `sampleData` at the end of the file containing `FEMViscoelasticity3D.m` is designed to make the parallel loops more readable. Given the time domain functions

$$\mathbf{f}(x,y,z,t), \quad \mathbf{u}_D(x,y,z,t), \quad \mathbf{t}(x,y,z,t),$$

corresponding to the forcing term, Neumann and Dirichlet data and a time value τ, we test the spatial functions

$$\mathbf{f}(x,y,z,\tau), \quad \mathbf{u}_D(x,y,z,\tau), \quad \mathbf{t}(x,y,z,\tau),$$

which is done as described in `FEMelasticity3D.m` in Section 1.2. Finally we return $3N_{\text{dof}} \times 1$ vectors

$$\mathbf{f}_h, \quad \mathbf{u}_{h,D}, \quad \mathbf{t}_h.$$

```
function [Uh,Vh,Wh,Sxxh,Syyh,Szzh,Syzh,Sxzh,Sxyh]= ...
    FEMViscoelasticity3D(mu,lam,rho,f,uD,sigma,T,k,time,varargin)
% [Uh,Vh,Wh,Sxxh,Syyh,Szzh,Syzh,Sxzh,Sxyh]= ...
%     FEMViscoelasticity3D(mu,lam,rho,f,uD,sigma,T,k,time)
%
% [Uh,Vh,Wh,Sxxh,Syyh,Szzh,Syzh,Sxzh,Sxyh]= ...
%     FEMViscoelasticity3D(mu,lam,rho,f,uD,sigma,T,k,time,flag)
%
% [Uh,Vh,Wh,Sxxh,Syyh,Szzh,Syzh,Sxzh,Sxyh]= ...
%     FEMViscoelasticity3D(mu,lam,rho,f,uD,sigma,T,k,time,flag,p)
%
% Input:
% mu        : First Lame parameter. Cell array {m_mu,b_mu,a_mu,nu_mu} of
%             function handles of (x,y,z)
% lam       : Second Lame parameter. Cell array {m_lam,b_lam,a_lam,nu_lam}
%             of function handles of (x,y,z)
% rho       : Mass density. Function handle of (x,y,z)
% uD        : Dirichlet BC. 3x1 cell array of function handles of (x,y,z,t)
% sigma     : Neumann BC. 3x3 cell array {Sxx,Sxy,Sxz;
%                                         Syx,Syy,Syz;
%                                         Szx,Szy,Szz}
%             of function handles of (x,y,z,t)
% f         : Force term. 3x1 cell array of function handles of (x,y,z,t)
% T         : Enhanced tetrahedrization
% k         : Polynomial degree
% time      : Equipartitioned temporal grid of [0,T] (vector of 1 x Nt)
% flag      : Optional. 0 (Default) for stress at nodes, 1 for ave. stress
% p         : Optional. Time stepping method. Function handle for the CQ
%             method or matrix for RKCQ. Default is trapezoidal rule
%
% Output:
% Uh,Vh,Wh  : Ndof x Nt array of FE cofficients for displacement in TD.
% Sxxh,...  : Components of stress in TD. dimP_{k-1}*Nelt x Nt or Nelt x Nt
%             array depending on whether flag is 0 or 1
%
% Last modified: Oct 24, 2018

nofarg = 9; % number of non optional arguments
% If 1st optional argument exists and = 1, then average stress is computed.
avestressFlag = 0;
if nargin ≥ nofarg+1
    avestressFlag = varargin{1};
end
```

```
% Handling 2nd optional argument for CQ/RKCQ
p = @(z) 2*(1-z)./(1+z); % Trapezoidal rule is default method
if nargin>=nofarg+2
    p = varargin{2};
end
% Determining the time stepping method
if isa(p,'function_handle')
    multistepMethod = true;
elseif isa(p,'numeric')
    A = p; % Runge-Kutta matrix
    multistepMethod = false;
    S = size(A,1); % number of stages
    c = sum(A,2);
end
% Dimensions
Nelt = size(T.elements,2);
Ndof = dimFEMspace(T,k);
km1 = k-1;
dimPkm1Nelt = (km1+3)*(km1+2)*(km1+1)/6*size(T.elements,2);

% Time-step
dt = time(2) - time(1);

% Sampling the data for CQ or RKCQ methods
if multistepMethod % implementing CQ
    Nt = length(time);
    % Generate input data for transfer function
    Y = zeros(3*Ndof,Nt); Z = zeros(3*Ndof,Nt);
    parfor n=1:Nt
        t=time(n);
        [fh,trh,uDh]=sampleData(f,sigma,uD,T,k,t)
        Y(:,n) = uDh;
        Z(:,n) = fh + trh;
    end
    X = [Y;Z];
else % implementing RKCQ
    Nt = length(time) - 1;
    Y = zeros(3*Ndof,S,Nt); Z = zeros(3*Ndof,S,Nt);
    X = zeros(6*Ndof,S,Nt);
    parfor n=1:Nt
        tn=time(n);
        for ns=1:S
            t = tn + dt * c(ns); %#ok
            [fh,trh,uDh]=sampleData(f,sigma,uD,T,k,t);
            Z(:,ns,n) = fh+trh;
            Y(:,ns,n) = uDh;
        end
    end
    % Reshape the data
    for ns=1:S
        X(:,ns,:) = [Y(:,ns,:);Z(:,ns,:)];
    end
    X = reshape(X,[S*6*Ndof,Nt]);
end

% Lame parameters are unpacked
m_mu = mu{1}; b_mu = mu{2}; a_mu = mu{3}; nu_mu = mu{4};
m_lam = lam{1}; b_lam = lam{2}; a_lam = lam{3}; nu_lam = lam{4};
muts = @(x,y,z,s) (m_mu(x,y,z) + ...
    s.^nu_mu(x,y,z).*b_mu(x,y,z))./(1 + s.^nu_mu(x,y,z).*a_mu(x,y,z));
lamts = @(x,y,z,s) (m_lam(x,y,z) + ...
    s.^nu_lam(x,y,z).*b_lam(x,y,z))./(1 + s.^nu_lam(x,y,z).*a_lam(x,y,z));
% Construct transfer function for time-stepping
Transfer = @(s,X) TFFEMelasticity3D(muts,lamts,rho,X,T,k,s,avestressFlag);

if avestressFlag
```

```
    dimStress = Nelt;
else
    dimStress = dimPkm1Nelt;
end
% CQ outputs all the information in one long vector
if multistepMethod
    allOut=CQoperator(Transfer,X,dt,3*Ndof+6*dimStress,p);
else % Runge-Kutta method is being implemented
    allOut=RKCQoperator(Transfer,X,dt,3*Ndof+6*dimStress,0,p);
end
% Indices to extract displacement and stress
block1=@(x) (1+(x-1)*Ndof):(x*Ndof);
block2=@(x) (1+(x-1)*dimPkm1Nelt):(x*dimStress);
% Unpacking displacement and stess
Uh = allOut(block1(1),:);
Vh = allOut(block1(2),:);
Wh = allOut(block1(3),:);
Sxxh = allOut(Ndof*3+block2(1),:);
Syyh = allOut(Ndof*3+block2(2),:);
Szzh = allOut(Ndof*3+block2(3),:);
Syzh = allOut(Ndof*3+block2(4),:);
Sxzh = allOut(Ndof*3+block2(5),:);
Sxyh = allOut(Ndof*3+block2(6),:);
end

function [fh,trh,uDh]=sampleData(f,sigma,uD,T,k,t)
% This is a helper function. Given time value t, it returns the sampled
% data for force, traction and dirichlet tests

% Load function
ft = cell(1,3);
for i=1:3
    ft{i} = @(x,y,z) f{i}(x,y,z,t);
end
fh = loadVector3D(ft,T,k);
fh = fh(:);

% Tranction
sigmat = cell(3,3);
for i=1:3
    for j=1:3
        sigmat{i,j} = @(x,y,z) sigma{i,j} (x,y,z,t);
    end
end
trh=neumannBC3D(sigmat,T,k);
trh = trh(:);

% Dirichlet condition
uDt = cell(1,3);
for i=1:3
    uDt{i} = @(x,y,z) uD{i} (x,y,z,t);
end
[uDh,~,~] = dirichletBC3D(uDt,T,k);
uDh = uDh(:);
end
```

2.2 Time domain elasticity

A note on how to perform time domain elasticity using `FEMViscoleasticity3D.m`.

2.3 Transfer function for Laplace domain

s

ec:2.3 Using `TFFEMelasticity3D.m`, we construct the transfer function $\mathcal{F}_h$ in the Laplace domain such that

$$\mathcal{F}_h(s, \mathbf{u}_D, \mathbf{f}, \boldsymbol{\sigma}_N) = (\mathbf{u}_h, \boldsymbol{\sigma}_h)$$

is FEM approximation of

$$\begin{aligned} -\operatorname{div} \boldsymbol{\sigma} + s^2 \rho \, \mathbf{u} &= \mathbf{f} && \text{in } \Omega, \\ \mathbf{u} &= \mathbf{u}_D && \text{on } \Gamma_D, \\ \boldsymbol{\sigma} \, \boldsymbol{\nu} &= \boldsymbol{\sigma}_N \, \boldsymbol{\nu} && \text{on } \Gamma_N. \end{aligned}$$

Since this solver is almost same as the solver of `FEMelasticity3D.m`, we will only describe the differences.

- Lame parameters here $\mu(s), \lambda(s)$ are functions of s. To compute stiffness matrices we first define the functions
$$\mu_s(x, y, z) = \mu(x, y, z, s), \quad \lambda_s(x, y, z) = \lambda(x, y, z, s),$$
then use them as input in `stiffnessMatrices3D`.

- The information of the approximations of load vector, Dirichlet and Neumann data are already given in the input `Data`. Although in `FEMelasticity3D.m` we run `loadVector3D, neumannBC3D, dirichletBC3D` to obtain these approximations, here we only need to decompose the $6N_{\text{dof}} \times 1$ vector `Data`:
$$\text{Data} = \begin{bmatrix} \mathbf{u}_{h,D} \\ \\ \text{rhs} \end{bmatrix}$$
where
$$\mathbf{u}_{h,D}, \qquad \text{rhs} = \mathbf{f}_h + \mathbf{t}_h,$$
are $3N_{\text{dof}} \times 1$ vectors. Here $\mathbf{f}_h, \mathbf{u}_{h,D}, \mathbf{t}_h$ are load, Dirichlet and Neumann approximations respectively. Note also that since $\mathbf{f}_h, \mathbf{t}_h$ go together, we do not need to decompose rhs.

- We use `bdDOF3D` to obtain the list of free, and Dirichlet nodes.

- We build the solver and obtain the approximation of the displacement vector exactly as it is done in `FEMelasticity3D.m`. Here stress is computed using `stressPostprocessing.m`. We refer to the last item in Section 2.1 for the different ways of outputting numerical stress.

```
function sol = TFFEMelasticity3D(muts,lamts,rho,Data,T,k,s,varargin)
% [Uhx;Uhy;Uhz;Sigxx;Sigyy;Sigzz;Sigyz;Sigxz;Sigxy] ...
%           = TFFEMelasticity3D(muts,lamts,rho,[uDh;rhs],T,k,s,flag)
%
% Input:
% muts, lamts   : Viscoelastic Lame parameters. Functions of (x,y,z,s)
% rho           : Mass density. Function of (x,y,z)
% [uDh;rhs]     : 6*Ndof x 1. Dirichlet BC and RHS (force term + Neumann BC)
% T             : Enhanced tetrahedrization
% k             : Polynomial degree
% s             : Complex number. Frequency parameter in Laplace domain
% flag          : 0 for stress at nodes, 1 for avereged stress at each
%                 element
%
% Output:
% [Uhx;...;Sigxx;...] : (3*Ndof + 6*dimStress) x 1 vector where
%                       Uhx,... : Ndof x 1 vectors for FE coefficients of
```

```
%                           the displacement.
%                        Sigxx,...: Coefficients of stress in piecewise
%                           P_{k-1} space or averaged stress on each element
%                           where dimStress is either dimP_{k-1}*Nelt or
%                           Nelt depending on whether flag is 0 or 1.
%
% Last modified : Oct 24, 2018

% Extracting data
Ndof = dimFEMspace(T,k);
uh = Data(1:3*Ndof);
rhs = Data(3*Ndof+1:6*Ndof);
% Constructing stiffness and mass matrices
mu = @(x,y,z) muts(x,y,z,s);
lam = @(x,y,z) lamts(x,y,z,s);
Sm = stiffnessMatrices3D(mu,T,k);
Sl = stiffnessMatrices3D(lam,T,k);
Mh = massMatrix3D(rho,T,k);

Sh = [2*Sm{1,1}+Sl{1,1}+Sm{2,2}+Sm{3,3}     Sl{1,2}+Sm{1,2}.'      Sl{1,3}+Sm{1,3}.';...
      Sm{1,2}+Sl{1,2}.'      2*Sm{2,2}+Sm{1,1}+Sl{2,2}+Sm{3,3}     Sl{2,3}+Sm{2,3}.';...
      Sl{1,3}.'+Sm{1,3}      Sl{2,3}.'+Sm{2,3}     2*Sm{3,3}+Sm{1,1}+Sm{2,2}+Sl{3,3}];

O = sparse(size(Mh,1),size(Mh,2));
Mh = [Mh O O;...
      O Mh O;...
      O O Mh];
% Getting Dricihlet and free indices
[~,~,~,~,~,dir,free]= bdDOF3D(T,k);
dir  = [dir(:) ; Ndof+dir(:) ; 2*Ndof+dir(:)];
free = [free(:); Ndof+free(:); 2*Ndof+free(:)];
% Solving for the displacement
MSh = s^2*Mh + Sh;
uh(free) = MSh(free,free)\...
    (rhs(free) - MSh(free,dir)*uh(dir));

u1h = reshape(uh,[Ndof,3]);
u2h = u1h(:,2);
u3h = u1h(:,3);
u1h(:,2:3) = [];
% In case of the optional argument is passed in and is equal to 1 then
% average stress is computed
nofarg = 7; % Number of non-optional arguments
if nargin >= nofarg+1
    avestress_flag = varargin{1};
else
    avestress_flag = 0;
end
if avestress_flag % compute average stress
    [sigxxh,sigyyh,sigzzh,sigyzh,sigxzh,sigxyh] = ...
        stressPostprocessing(u1h,u2h,u3h,lam,mu,T,k,1);
    % Combining displacement and stress as a single vector output
    sol = [u1h;u2h;u3h;sigxxh.';sigyyh.';sigzzh.';sigyzh.';sigxzh.';sigxyh.'];
    return
else % compute stress at nodes
    [sigxxh,sigyyh,sigzzh,sigyzh,sigxzh,sigxyh] = ...
        stressPostprocessing(u1h,u2h,u3h,lam,mu,T,k);
end
% Combining displacement and stress as a single vector output
sol = [u1h;u2h;u3h;sigxxh;sigyyh;sigzzh;sigyzh;sigxzh;sigxyh];
end
```

2.4 Creating an exact solution for fractional Zener model

Goal. Given space functions $\mathbf{u}_X, \rho, \mu_X, \lambda_X$, we want to construct a separable analytical solution

$$\mathbf{u}(x, y, z, t) = \mathbf{u}_X(x, y, z)t^p$$

for the problem (1), where for the strain-stress relation we assume

$$\begin{aligned} m_\mu &= m\,\mu_X, \quad b_\mu = b\,\mu_X \\ m_\lambda &= m\,\lambda_X, \quad b_\lambda = b\,\lambda_X \end{aligned}$$

and

$$\nu_\mu = \nu_\lambda = \nu, \qquad a_\mu = a_\lambda = a.$$

Here m, b, a, ν are positive scalars with the restrictions $b \geq m\,a$ and $\nu \in [0, 1]$. Then the stress in Laplace domain described in (1d) can be written as

$$\boldsymbol{\sigma} = \frac{m + bs^\nu}{1 + as^\nu}\boldsymbol{\sigma}_X,$$

where

$$\boldsymbol{\sigma}_X = \mu_X \boldsymbol{\varepsilon}(\mathbf{u}_X) + \lambda_X(\nabla \cdot \mathbf{u}_X)\mathrm{I}.$$

Construction. We compute all components of spatial parts of stress and the forcing term

$$\begin{aligned} \boldsymbol{\epsilon}_X &= \frac{1}{2}(\nabla \mathbf{u}_X + \nabla \mathbf{u}_X^\top) \\ \boldsymbol{\sigma}_X &= 2\mu\,\boldsymbol{\epsilon}_X + \lambda\,(\nabla \cdot \mathbf{u}_X)\,\mathrm{I} \end{aligned}$$

$$\mathbf{f}_X = -\mathrm{div}\boldsymbol{\sigma}_X$$

where components of $\boldsymbol{\sigma}_X$ is denoted as

$$\begin{bmatrix} \boldsymbol{\sigma}_{xx} & \boldsymbol{\sigma}_{xy} & \boldsymbol{\sigma}_{xz} \\ \times & \boldsymbol{\sigma}_{yy} & \boldsymbol{\sigma}_{yz} \\ \times & \times & \boldsymbol{\sigma}_{zz} \end{bmatrix}.$$

To compute the temporal parts, we use the following results.

Proposition 1. *For $\boldsymbol{\epsilon}(x, y, z, t) = \boldsymbol{\epsilon}_X(x, y, z)\,t^p$ with $p \geq 3$, the stress in Laplace domain becomes*

$$\boldsymbol{\sigma}(x, y, z, s) = \boldsymbol{\sigma}_X(x, y, z)\sigma_T(s)$$

where σ_T is defined in the Laplace domain in the following way

$$\sigma_T(s) := \frac{m\Gamma(\gamma + 1)}{a}\frac{s^{\nu-(\nu+\gamma+1)}}{s^\nu + \frac{1}{a}} + \frac{b\Gamma(\gamma + 1)}{a}\frac{s^{\nu-(\gamma+1)}}{s^\nu + \frac{1}{a}}.$$

and $\boldsymbol{\sigma}_X$ is as constructed earlier.

Here $p \geq 3$ makes the solution causal enough, and proposition follows from (**??**). To find the stress in time domain we utilize use the following lemma.

Lemma 1. *If $Re(\alpha), Re(\beta) > 0$, then*

$$\mathcal{L}\left\{t^{\beta-1}E_{\alpha,\beta}(-a\,t^\alpha)\right\} = \frac{s^{\alpha-\beta}}{s^\alpha + a},$$

where $E_{\alpha,\beta}$ is the Mittag-Leffler function.

With these results we compute time domain function

$$\sigma_T(t) = \frac{m\Gamma(\gamma+1)}{a} t^{\nu+\gamma} E_{\nu,\nu+\gamma+1}(\frac{-t^\nu}{a}) + \frac{b_1\Gamma(\gamma+1)}{a} t^\gamma E_{\nu,\gamma+1}(\frac{-t^\nu}{a}). \tag{2}$$

{eq:sig1}

Here terms

$$E_{\nu,\nu+\gamma+1}(\frac{-t^\nu}{a}), \qquad E_{\nu,\gamma+1}(\frac{-t^\nu}{a})$$

are computed using `mlf.m` function (see Section 7.1 Mittag-Leffler function). Next, we implement the temporal functions

$$h(t) = t^p, \qquad h''(t) = p(p-1)t^{p-2}$$

and construct the following space-time functions

$$\begin{aligned}
\mathbf{u}(x,y,z,t) &= \mathbf{u}_X(x,y,z)h(t),\\
\boldsymbol{\sigma}(x,y,z,t) &= \boldsymbol{\sigma}_X(x,y,z)\boldsymbol{\sigma}_T(t),\\
\mathbf{f}(x,y,z,t) &= \rho(x,y,z)\,\mathbf{u}_X(x,y,z)\,h''(t) + \mathbf{f}_X(x,y,z)\,\sigma_T(t).
\end{aligned}$$

How to use the script. The script `scriptExactSolutionFracVisco.m` can be used in the following way.

(a) **Input**

- a, b, m: Scalar viscoelastic parameters such that $b \geq a\,m$.
- ν: Fractional power in $[0, 1]$.
- p: Degree of the temporal solution, `deg` in the code. Should be an integer not less than 3.
- Spatial functions: The functions ρ, μ_X, λ_X and $\mathbf{u}_X$ should be defined before running this script. We assume these functions and their necessary derivatives are defined in the following way.

Spatial function	Definition in the code
Mass density (> 0)	`rho`
Lame parameters (> 0)	`mu, lam`
Lame parameters' 1st derivatives	`mux, muy, muz, lamx, lamy, lamz`
Displacement	`u, v, w`
Displacement 1st, 2nd derivatives	`ux, uy, uz, uxx, uxy, uxz, uyz, uzz`, same for `v, w`

The scripts `scriptExactDispCinf, scriptExactDispPoly` can be used to generate these functions.

(b) **Runnig externally.** The parameters described above should be defined before, if `scriptExactSolutionFracVisco.m` is being run in another script. In that case a parameter named `external` should be created too. (For example `external = 1`). Otherwise user will be prompted to enter the input values from the command line.

(c) **Output.** All output functions are vectorized in either x, y, z or t. For example if x, y, z are vector valued, then to evaluate these function one needs to insert scalar values of t, and vice versa.

- Displacement, $\mathbf{u}$: `U,V,W` in the code. Function handles of x, y, z, t.
- Stress, $\boldsymbol{\sigma}$: `Sxx, Syy, Szz, Sxy, Sxz, Syz`. Function handles of x, y, z, t.
- Forcing term, $\mathbf{f}$: `Fx, Fy, Fz`. Function handle of x, y, z, t.

- Lame parameters: `Mu, Lam`. 1×4 cell arrays of function handles of (x, y, z) (not t). They are packed so that

$$\text{Mu} = \{m_\mu, b_\mu, a_\mu, \nu_\mu\} = \{m\mu_X, b\mu_X, a, \nu\}$$
$$\text{Lam} = \{m_\lambda, b_\lambda, a_\lambda, \nu_\lambda\} = \{m\lambda_X, b\lambda_X, a, \nu\}$$

  which is the order `FEMViscoelasticity.m` uses.

```
%% scriptExactSolutionFracVisco.m
% A script to generate an exact solution for fractional isotropic
% generalized Zener model for viscoelastic wave propagation problem.
% Different viscoelastic models can be tested via different input.
%
% Input:
% a,b,m         : Viscoelastic parameters. Scalars such that b > a*m
% nu            : Fractional power. Scalar in [0,1]
% deg           : Degree of H(t). Integer ≥ 3
% Spatial func.s: Should be defined before calling the script
%                 Mass density : rho
%                 Lame and derv: mu, lam, mux, muy, muz, lamx, lamy, lamz
%                 Displacement : u, v, w,
%                 Derv. of disp: ux, uy, uz, uxx, uxy, uxx, uyy, uyz, uzz,
%                                and same for v, w
%
% Output:
% U,V,W     : Seperable exact solution. Function handles of x,y,z,t
%             U = u(x)h(t), V=v(x)h(t), W=w(x)h(t)
% Sxx,...   : Symmetric stress tensor. 6 function handles of x,y,z,t.
% Fx,Fy,Fz  : Forcing term. Function handles of x,y,z,t
% Mu, Lam   : Lame parameters. 4 x 1 cell array of function handles of
%             (x,y,z,t). See FEMViscoelasticity3D's input
%
% Last modified: October 23, 2018

%% fractional Zener model and temporal solution parameters
% These parameters help to obtain different models
% c1 − a*c0 > 0, c1=b, c0=m, hence b − a*m > 0
% Example input:  a = 1; b = 2.5; m = 1; nu = 0.5; deg = 3;
if ¬exist('external','var')
    a = input('Viscoparameter a: ');
    b = input('Viscoparameter b: ');
    m = input('Viscoparameter m (so that b>a*m): ');
    nu = input('Fractional power nu: ');
    deg = input('Degree of temporal solution: ');
end
%% Derived quantities: Spatial part of forcing term and stress
div=@(x,y,z) ux(x,y,z)+vy(x,y,z)+wz(x,y,z);
divx=@(x,y,z) uxx(x,y,z)+vxy(x,y,z)+ wxz(x,y,z);
divy=@(x,y,z) uxy(x,y,z)+vyy(x,y,z)+wyz(x,y,z);
divz=@(x,y,z) uxz(x,y,z)+vyz(x,y,z)+wzz(x,y,z);

sxx=@(x,y,z) 2*mu(x,y,z).*ux(x,y,z)+lam(x,y,z).*div(x,y,z);
syy=@(x,y,z) 2*mu(x,y,z).*vy(x,y,z)+lam(x,y,z).*div(x,y,z);
szz=@(x,y,z) 2*mu(x,y,z).*wz(x,y,z)+lam(x,y,z).*div(x,y,z);
sxy=@(x,y,z) mu(x,y,z).*(vx(x,y,z)+uy(x,y,z));
sxz=@(x,y,z) mu(x,y,z).*(wx(x,y,z)+uz(x,y,z));
syz=@(x,y,z) mu(x,y,z).*(wy(x,y,z)+vz(x,y,z));
fx=@(x,y,z) −(lam(x,y,z).*divx(x,y,z)+lamx(x,y,z).*div(x,y,z)...
    +2*mu(x,y,z).*uxx(x,y,z)+2*mux(x,y,z).*ux(x,y,z)...
    +mu(x,y,z).*(vxy(x,y,z)+uyy(x,y,z))+muy(x,y,z).*(vx(x,y,z)+uy(x,y,z))...
    +mu(x,y,z).*(wxz(x,y,z)+uzz(x,y,z))+muz(x,y,z).*(wx(x,y,z)+uz(x,y,z)));
fy=@(x,y,z) −(lam(x,y,z).*divy(x,y,z)+lamy(x,y,z).*div(x,y,z)...
    +2*mu(x,y,z).*vyy(x,y,z)+2*muy(x,y,z).*vy(x,y,z)...
    +mu(x,y,z).*(wyz(x,y,z)+vzz(x,y,z))+muz(x,y,z).*(wy(x,y,z)+vz(x,y,z))...
```

```
    +mu(x,y,z).*(uxy(x,y,z)+vxx(x,y,z))+mux(x,y,z).*(uy(x,y,z)+vx(x,y,z)));
fz=@(x,y,z) -(lam(x,y,z).*divz(x,y,z)+lamz(x,y,z).*div(x,y,z)...
    +2*mu(x,y,z).*wzz(x,y,z)+2*muz(x,y,z).*wz(x,y,z)...
    +mu(x,y,z).*(uxz(x,y,z)+wxx(x,y,z))+mux(x,y,z).*(uz(x,y,z)+wx(x,y,z))...
    +mu(x,y,z).*(vyz(x,y,z)+wyy(x,y,z))+muy(x,y,z).*(vz(x,y,z)+wy(x,y,z)));

%% Temporal part of exact solution
h = @(t) t.^deg;
hp = @(t) deg*t.^(deg-1);
hpp = @(t) deg*(deg-1)*t.^(deg-2);
prec_mlf = 10; % accuracy of the computation of mlf. 10 should be enough.
if a == 0
    sigmaT = @(t) m*t.^(deg) + b*gamma(deg+1)/gamma(deg+1-nu)*t.^(deg-nu);
else
    sigmaT = @(t) m*gamma(deg+1)/a*t.^(nu+deg).*mlf(nu,nu+deg+1,-t.^nu/a,prec_mlf)...
        +b*gamma(deg+1)/a*t.^deg.*mlf(nu,deg+1,-t.^nu/a,prec_mlf);
end

%% Combining spatial and temporal solutions
% exact stress
Sxx = @(x,y,z,t) sxx(x,y,z).*sigmaT(t);
Syy = @(x,y,z,t) syy(x,y,z).*sigmaT(t);
Szz = @(x,y,z,t) szz(x,y,z).*sigmaT(t);
Sxy = @(x,y,z,t) sxy(x,y,z).*sigmaT(t);
Sxz = @(x,y,z,t) sxz(x,y,z).*sigmaT(t);
Syz = @(x,y,z,t) syz(x,y,z).*sigmaT(t);

% exact forcing term
Fx = @(x,y,z,t) rho(x,y,z)*hpp(t).*u(x,y,z) + fx(x,y,z).*sigmaT(t);
Fy = @(x,y,z,t) rho(x,y,z)*hpp(t).*v(x,y,z) + fy(x,y,z).*sigmaT(t);
Fz = @(x,y,z,t) rho(x,y,z)*hpp(t).*w(x,y,z) + fz(x,y,z).*sigmaT(t);
% exact displacement
U = @(x,y,z,t) u(x,y,z)*h(t);
V = @(x,y,z,t) v(x,y,z)*h(t);
W = @(x,y,z,t) w(x,y,z)*h(t);

Ux = @(x,y,z,t) ux(x,y,z)*h(t);
Vx = @(x,y,z,t) vx(x,y,z)*h(t);
Wx = @(x,y,z,t) wx(x,y,z)*h(t);

Uy = @(x,y,z,t) uy(x,y,z)*h(t);
Vy = @(x,y,z,t) vy(x,y,z)*h(t);
Wy = @(x,y,z,t) wy(x,y,z)*h(t);

Mu = cell(1,4);
Lam = cell(1,4);
Mu{1} = @ (x,y,z) m*mu(x,y,z);
Mu{2} = @ (x,y,z) b*mu(x,y,z);
Mu{3} = @ (x,y,z) a + 0*x;
Mu{4} = @ (x,y,z) nu + 0*x;
Lam{1} = @ (x,y,z) m*lam(x,y,z);
Lam{2} = @ (x,y,z) b*lam(x,y,z);
Lam{3} = @ (x,y,z) a + 0*x;
Lam{4} = @ (x,y,z) nu + 0*x;
```

3 Tools for time domain problems

3.1 Convolution quadrature methods

sec:3.1

The goal of these methods are approximating the convolution

$$\mathbf{u} = \mathcal{F} * \mathbf{g}.$$

We use a similar notation used in [1] and assume the time values correspond to the columns of the matrices that appear in our computations. Given number of time steps N_t, we will use N as the number of columns of input and output which has different definition depending on the method.

- Multistep: $N = N_t + 1$,
- Runge-Kutta: $N = N_t$.

Our input consists of

- F: $N_1 \times N$ matrix valued transfer function corresponding to $\mathcal{L}\{\mathcal{F}\}$. Here F is a function handle of s and $\mathbf{h}$ where
  - s: Complex scalar. Represents frequency in Laplace domain.
  - $\mathbf{h}$: $N_2 \times 1$ vector.
- $\mathbf{g}$: Matrix of time values of the input data. Each column represents the data evaluated at the corresponding time-step. We apply CQ scheme to the columns of this matrix with the help of the transfer function.
- κ: Scalar time-step. This corresponds to `k` in the code.
- CQ method option: Either a rational function $p(z)$ or a matrix $\mathbf{A}$ to determine the time-stepping method for CQ.
- Output dimension: We input a number which helps us to built the quantity $\mathbf{u}$ which corresponds to the approximation $\mathcal{F} * \mathbf{g}$. The definition of this number will be clarified for each method.

In both multistep and Runge-Kutta methods we follow a similar pattern.

1. We start with precomputing
$$\omega = \zeta_{N+1}, \qquad R = \epsilon^{\frac{1}{2(N+1)}}$$
where ϵ is the floating point relative accuracy constant `eps` in MATLAB which is approximately 10^{-16}.
2. The matrix $\mathbf{g}$ contains all the information at all time steps where each time step represents a column. Then using `fft`, we take DFT of the columns to obtain the following matrix
$$\widehat{\mathbf{h}} = \mathcal{F}\{\mathbf{g} \odot [R^0, R^1, \ldots, R^N]\}.$$
3. [Main computation] We compute $\widehat{\mathbf{u}}$ by running a parallel loop over the indices $\ell = 1, \ldots, \lfloor \frac{N}{2} \rfloor + 1$ (This is actually $\widehat{\mathbf{v}}$ in the paper). In the code $\widehat{u}_\ell$ corresponds to `u(:,l+1)`.
4. Then for $\ell > \lfloor \frac{N}{2} \rfloor + 1$ we use the identity
$$\widehat{u}_{N-\ell} = \overline{\widehat{u}}_\ell.$$
5. Finally applying `ifft` to columns we compute
$$\mathbf{u} = \mathrm{Re}\, \mathcal{F}^{-1}\{\widehat{\mathbf{u}} \odot [R^0, R^{-1}, \ldots, R^{-N}]\}.$$

Usage.

Some comments....

Note. A small change of order 10^{-15} at early stages of the CQ computation may amplify to an order 10^{-7} of a final change because of the last column of $[R^0, R^{-1}, \ldots, R^{-N}]$ which is
$$R^{-N} = \epsilon^{\frac{-N}{2(N+1)}} \approx 10^8.$$

3.2 CQ with multistep methods

The function `CQoperator.m` implements various multi-steps methods with CQ. The desired multistep method can be given via the input $p(z)$.

Examples of multistep methods

(1) Backward Euler (BE)

$$p(z) = 1 - z.$$

(2) BDF2

$$p(z) = \frac{3}{2} - 2z + \frac{1}{2}z^2.$$

(3) Trapezoidal (TR)

$$p(z) = 2\frac{1-z}{1+z}.$$

Implementing CQ. The input $F, \mathbf{g}$ and the output $\mathbf{u}$ take the following form

- F: $d_1 \times (N_t + 1)$ matrix valued transfer function of s and $\mathbf{h}$ where $\mathbf{h}$ is a $d_2 \times 1$ vector,
- $\mathbf{g}$: $d_2 \times (N_t + 1)$ matrix,
- $\mathbf{u}$: $d_1 \times (N_t + 1)$ matrix.

Main computation. To compute the ℓ-th column of $\widehat{\mathbf{u}}$, we use the transfer function F and ℓ-th column of $\widehat{\mathbf{h}}$ in the following way

$$\widehat{u}_\ell = F\big(p(R\omega^\ell)/\kappa\big)\,\widehat{h}_\ell.$$

Rest of the computations are done as explained in Section 3.1.

```
function u = CQoperator(F,g,k,dim,varargin)
% u = CQoperator(F,g,k,dim)
% u = CQoperator(F,g,k,dim,p)
%
% Input:
%    F : dim x 1 valued transfer function. Function handle of
%        s : complex number, X : d2 x 1 complex valued vector
%    g : Time values of input (d2 x (M+1) matrix)
%    k : Time-step
%  dim : Dimension of output for F
%    p : Transfer function of multistep method. Function handle of s
%
% Output:
%    u : dim x (M+1) matrix corresponding to F(\partial_k) g
%
% Last Modified : September 19, 2018

if nargin==4
    p = @(z) 2*(1-z)./(1+z);     % The default is TR
else
    p=varargin{1};
end

M  = size(g,2)-1;              % M = number of time-steps
omega = exp(2*pi*1i/(M+1));
R = eps^(0.5/(M+1));

h = bsxfun(@times,g,R.^(0:M));
h = fft(h,[],2);               % DFT by columns (\hat H)
```

```
u = zeros(dim,M+1);
parfor l=0:floor((M+1)/2)
    u(:,l+1)=F(p(R*omega^(-l))/k,h(:,l+1));%#ok      % \hat v
end
u(:,M+2-(1:floor(M/2)))=conj(u(:,2:floor(M/2)+1));
u=real(ifft(u,[],2));                              % v
u=bsxfun(@times,u,R.^(-(0:M)));

return
```

3.3 CQ with Runge-Kutta method

sec:3.3

A subclass of implicit Runge-Kutta methods. We will be using the ones which can be represented via Butcher tableau in the following way

$$\begin{array}{c|c} \mathbf{c} & \mathbf{A} \\ \hline & \mathbf{b}^\top \end{array} \qquad \mathbf{b}, \mathbf{c} \in \mathbb{R}^S, \qquad \mathbf{A} \in \mathbb{R}^{S\times S},$$

where

$$\mathbf{A1} = \mathbf{c}, \qquad \mathbf{e}_S^\top \mathbf{A} = \mathbf{b}^\top, \qquad \mathbf{b}^\top \mathbf{1} = 1, \qquad .$$

with

$$\mathbf{1} = (1, \dots, 1)^\top, \qquad \mathbf{e}_S^\top = (0, \dots, 0, 1).$$

Examples

(1) Radau IIA, 3rd order

$$\mathbf{A} = \begin{bmatrix} 5/12 & -1/12 \\ 3/4 & 1/4 \end{bmatrix}$$

(2) Lobatto IIID, 4th order

$$\mathbf{A} = \begin{bmatrix} 1/6 & -1/3 & 1/6 \\ 1/6 & 5/12 & -1/12 \\ 1/6 & 2/3 & 1/6 \end{bmatrix}$$

(3) Radau IIA, 5th order

$$\mathbf{A} = \begin{bmatrix} 11/45 - 7\sqrt{6}/360 & 37/225 - 169\sqrt{6}/1800 & -2/225 + \sqrt{6}/75 \\ 37/225 + 169\sqrt{6}/1800 & 11/45 + 7\sqrt{6}/360 & -2/225 - \sqrt{6}/75 \\ 4/9 - \sqrt{6}/36 & 4/9 + \sqrt{6}/36 & 1/9 \end{bmatrix}$$

Note. These matrices can be also found in the script `scriptRKCQXXX`

Implementing Runge Kutta. We implement Algorithm 5.II from [1] using the above notation. We assume the input and output spaces have the dimensions d_1 and d_2 respectively and we have a Runge-Kutta method (which is given as a $S \times S$ matrix $\mathbf{A}$) with S stages. The input $F, \mathbf{g}$ and the output $\mathbf{U}$ take the following form

- F: $S\,d_1 \times N$ matrix valued transfer function of s and $\mathbf{h}$ where $\mathbf{h}$ is a $d_2 \times 1$ vector.

- $\mathbf{g}$: $S\,d_2 \times N$ matrix.
- $\mathbf{U}$: $Sd_1 \times N$ or $d_1 \times N$ matrix.

Using matrix $\mathbf{A}$, we define $S \times 1$ vectors

$$\mathbf{1} = [1, \ldots, 1]^\top, \qquad \mathbf{b}^\top = [0, 0, \ldots, 0, 1]\mathbf{A}.$$

The overall computation follows the steps described in 3.1. We will only explain the computation in the parallel loop.

Main computation. The following is how the iteration number ℓ of the parallel loop is handled. We start with defining

$$z_\ell = R\omega^\ell, \qquad \Delta_\ell = (A + \frac{z_\ell}{1 - z_\ell})\mathbf{1}\mathbf{b}^\top,$$

then using MATLAB `eig` function we compute the spectral decomposition of

$$\frac{1}{\kappa}\Delta_\ell = P_\ell \Lambda_\ell P_\ell^{-1}.$$

We denote the diagonal of Λ_ℓ as $\lambda_1, \lambda_2, \ldots, \lambda_S$ (clearly all depend on ℓ). Now our goal is computing

$$(P_\ell \otimes I_{d_1})\mathrm{diag}(F(\lambda_1), \ldots, F(\lambda_S))(P_\ell^{-1} \otimes I_{d_2})\widehat{\mathbf{h}}_\ell.$$

where $\widehat{\mathbf{h}}_\ell$ is the ℓ-th column of the $d_2\,S \times N$ matrix $\widehat{\mathbf{h}}$. We do that in the following way

- We compute the $S\,d_2 \times 1$ vector
$$\mathbf{G}_\ell = (P_\ell^{-1} \otimes I_{d_2})\widehat{\mathbf{h}}_\ell.$$
- Then slice $\mathbf{G}_\ell$ to $d_2 \times 1$ vectors $\mathbf{g}_\ell^s$ for $s = 1, 2, \ldots, S$ such that
$$\mathbf{G}_\ell = \begin{bmatrix} \mathbf{g}_\ell^1 \\ \hline \vdots \\ \hline \mathbf{g}_\ell^S \end{bmatrix}.$$
We find the locations of vectors $\mathbf{g}_\ell^1, \ldots, \mathbf{g}_\ell^S$ with the help of the function handle `idy`. For each s, `idy(s)` gives us the indices of $\mathbf{G}_\ell$ that corresponds to the location of $\mathbf{g}_\ell^s$.
- Loop over $s = 1, 2, \ldots, S$ to apply the transfer function F to obtain $d_1 \times 1$ vectors
$$\mathbf{w}_\ell^s = F(\lambda_s, \mathbf{h}_\ell^s) \qquad s = 1, 2, \ldots, S$$
and glue them together
$$\mathbf{W}_\ell = \begin{bmatrix} \mathbf{w}_\ell^1 \\ \hline \vdots \\ \hline \mathbf{w}_\ell^S \end{bmatrix}.$$
To locate $\mathbf{w}_\ell^1, \ldots, \mathbf{w}_\ell^S$ we use `idx` in a similar fashion to `idy`.
- Finally ℓ-th column of $\mathbf{U}$ can be computed as
$$\mathbf{U}_\ell = (P_\ell \otimes I_{d_1})\mathbf{W}_\ell.$$

Output. In the end,

- If the indicator variable `flag=1`, we output $\mathbf{U}$, all stage values.

- Otherwise we output the last d_2 rows of $\mathbf{U}$, the values corresponding to the last stage.

```
function U = RKCQoperator(F,g,k,d1,flag,varargin)

% U = RKCQoperator(F,g,k,d1,flag)
% U = RKCQoperator(F,g,k,d1,flag,A)
%
% Input:
%   F(z,h)  : d1 x 1 valued function handle of ...
%             z : complex number, h : d2 x 1 complex valued vector
%   g       : S*d2 x (N+1) matrix (S=2 if RadauIIa is used)
%   d1      : Dimension of function space of the output
%   k       : Real number, length of the time step
%   flag    : 1, return all stages, 0 return last stage (step) only
%   A       : S x S real matrix for RK methods. If not given 2-stage
%             RadauIIa is implemented
%
% Output:
%   U    : S*d1 x (N+1) matrix (flag=1)
%   U    : d1 x (N+1) matrix (flag=0)
%
% Last modified: September 19, 2018

% If there is no A given, RadauIIa method is implemented
if nargin == 5
    A=[5/12 -1/12; 3/4 1/4];
else
    A=varargin{1};
end
S   = size(A,1); % number of stages
bT = A(end,:);
one = ones(S,1);

N  = size(g,2)-1; % number of time steps
d2 = size(g,1)/S; % dimension of the input space
omega= exp(2*pi*1i/(N+1));
R = eps^(0.5/(N+1));

g=bsxfun(@times,g,R.^(0:N));
g=fft(g,[],2); % hhat

idx=@(s) (s-1)*d1+1:(s*d1);
idy=@(s) (s-1)*d2+1:(s*d2);

U=zeros(S*d1,N+1);
% Compute half the hermitian sequence
parfor l=0:floor((N+1)/2)
    z = R*omega^(-l);
    Delta = inv(A + z/(1-z) * one * bT);
    [P,Lambda]=eig(Delta);
    Lambda=diag(Lambda)/k;
    gl=kron(inv(P),speye(d2))*g(:,l+1);
    ul=zeros(S*d1,1);
    for s=1:S
        ul(idx(s))=F(Lambda(s), gl(idy(s))); %#ok
    end
    U(:,l+1)=kron(P,speye(d1))*ul;
end
% Mirror the hermitian sequence
U(:,N+2-(1:floor(N/2)))=conj(U(:,2:floor(N/2)+1));
U=ifft(U,[],2);

U=bsxfun(@times,U,R.^(-(0:N)));
U=real(U);
% If all stages are asked, all rows are returned
```

```
if flag
    return
end
% If only the last stage was asked, the last block of the rows are returned
U=U((S-1)*d1+1:end,:);
return
```

3.4 Usage of CQ operators

Some guidelines to write a script using (RK)CQoperator

- Design a transfer function as a MATLAB function. More precisely, write a separate function that takes two necessary parameters `data` and complex number `s`, and other parameters, and outputs a `solution`. This function is your Laplace domain transfer function. The important part here is that `data` and `solution` should be vector quantities.

  A good example for a design is `TFFEMelasticity3D.m`. In that code `data` contains the information of tested Dirichlet, Neumann boundary data and load vector. They are put together as an input and in the code they are extracted and used as they are needed.

- Write an anonymous simplified transfer function out of your MATLAB transfer function. Here is an example. Suppose you have a MATLAB transfer function defined in the following way

  ```
  function solution = transfer(s,data,otherParam)
  # do stuff
  end
  ```

  Next, in the script that you use `CQoperator`, define all other parameters

  ```
  otherParam = ...
  ```

  Finally your inline transfer function should be defined as

  ```
  transferFunc = @(s,data) transfer(s,data,otherParam)
  ```

  A good example for this can be seen on line 107 of `FEMViscoelasticity3D.m`.

- Simply call `CQoperator`

  ```
  allOut = CQoperator(transferFunc,data,k,dim)
  ```

  and extract the variable `allOut`. The extraction depends on how you construct your transfer function.

3.5 Smooth window function

`window(a,b,c,d)` creates a window function $w(t)$ and its derivatives $w'(t), w''(t)$ where

$$w(t) = H\left(\frac{t-a}{b-a}\right) H\left(\frac{c-t}{d-c}\right)$$

with

$$H(t) = \begin{cases} 0, & \text{if} \quad t < 0. \\ t^5(1-5(t-1)+15(t-1)^2-35(t-1)^3+70(t-1)^4-126(t-1)^5), & \text{if} \quad 0 < t < 1, \\ 1, & \text{if} \quad 1 \le t. \end{cases}$$

As a result we have

$$w(t) = \begin{cases} 0, & \text{if} \quad t < a, \\ 1, & \text{if} \quad b < t < c, \\ 0, & \text{if} \quad d < t. \end{cases}$$

where $w(t)$ is increasing smoothly on (a, b) and decreasing smoothly on (b, c).

```
function [wf,wfp,wfpp]=window(a,b,c,d)
%
% [wf,wfp,wfpp]=window(a,b,c,d)
%
% Input:
% a,b,c,d       : Numbers such that
%                 wf(x)=0 on (-infty,a]
%                 wf(x)=inc on [a,b]
%                 wf(x)=1 on [b,c]
%                 wf(x)=dec on [c,d]
%                 wf(x)=0 on [d, infty]
%
% Output:
% wf            : A function handle
% wfp, wfpp     : Function handles (derivatives of wf)
%
% Last modified: November 21, 2017

HH =  @(x) x.^5.*(1-5*(x-1)+15*(x-1).^2-35*(x-1).^3+70*(x-1).^4-...
    126*(x-1).^5).*(x>0).*(x<1)+(x>=1);
HHp = @(x) (5*x.^4.*...
    (1-5*(x-1)+15*(x-1).^2-35*(x-1).^3+70*(x-1).^4-126*(x-1).^5)...
    +x.^5.*(-5+30*(x-1)-105*(x-1).^2+280*(x-1).^3-630*(x-1).^4))...
    .*(x>0).*(x<1);
HHpp= @(x) -1260*(x-1).^4.*(9*x-4).*(x.^3).*(x>0).*(x<1);
wf  = @(x) HH((x-a)./(b-a)).*HH((d-x)./(d-c));
wfp = @(x) 1/(b-a)*HHp((x-a)/(b-a)).*HH((d-x)./(d-c))...
    -1/(d-c)*HH((x-a)/(b-a)).*HHp((d-x)./(d-c));
wfpp = @(x) 1/(b-a)^2*HHpp((x-a)/(b-a)).*HH((d-x)./(d-c))...
    -2/(b-a)*1/(d-c)*HHp((x-a)/(b-a)).*HHp((d-x)./(d-c))...
    +1/(d-c)^2*HH((x-a)/(b-a)).*HHpp((d-x)./(d-c));
```

3.6 Creating a plane wave

The function `vectorPlaneWave.m` creates function handles which give the three components of a three dimensional plane wave and its first and second derivatives in space and in time.
In order to use this function, you need to know the amplitude and propagation direction (vector quantities), as well as the speed of propagation and lag in time. The cell array input `signal` needs to contain the signal function f as the first cell, its first derivative f' as the second cell, and its second derivative f'' as the third cell.
The output

$$\mathbf{u} = (u, v, w) = f(c(t - t_{\text{lag}}) - \mathbf{x} \cdot \mathbf{d})\mathbf{a},$$

where

$$\begin{aligned} \mathbf{a}, \mathbf{d} &\in \mathbb{R}^3, & c, t_{\text{lag}} &\in \mathbb{R}, \\ |\mathbf{d}| &= 1, & c &> 0, \end{aligned}$$

and f is a causal function, meaning that $f(t) = 0$, for all $t \leq 0$.

```
function [u,v,w,ut,vt,wt,utt,vtt,wtt,...
                ux,uy,uz,vx,vy,vz,wx,wy,wz,...
                uxx,uxy,uxz,uyy,uyz,uzz,...
                vxx,vxy,vxz,vyy,vyz,vzz,...
                wxx,wxy,wxz,wyy,wyz,wzz] ...
                = vectorPlaneWave(a,d,c,tlag,signal)

%       [u,v,w,ut,vt,wt,utt,vtt,wtt,...
%               ux,uy,uz,vx,vy,vz,wx,wy,wz,...
%               uxx,uxy,uxz,uyy,uyz,uzz,...
%               vxx,vxy,vxz,vyy,vyz,vzz,...
```

```
%               wxx,wxy,wxz,wyy,wyz,wzz]    ...
%               = vectorPlaneWave(a,d,c,tlag,signal)
%
% Input:
%     a       : vector amplitude of wave (3-vector)
%     d       : unit propagation direction vector (3-vector)
%     c       : wave speed
%     tlag    : lag time
%     signal  : 3-cell array with f, f', f''
%
%  Output:
%     u,v,w                   : function handles for a plane wave
%                               (u,v,w) = f(d.x-c(t-tlag))a
%     ux,uy,...,wz            : function handles with first derivatives
%     ut,vt,wt,utt,vtt,wtt    : function handles with time derivs
%     uxx,uxy,...,wzz         : function handles with second derivatives
%
% Last modified May 20, 2016.

f = signal{1};
fp = signal{2};
fpp = signal{3};

% u
u=@(x,y,z,t)  a(1)*f(-(x*d(1)+y*d(2)+z*d(3))+c*(t-tlag));
ux=@(x,y,z,t) -d(1)*a(1)*fp(-(x*d(1)+y*d(2)+z*d(3))+c*(t-tlag));
uy=@(x,y,z,t) -d(2)*a(1)*fp(-(x*d(1)+y*d(2)+z*d(3))+c*(t-tlag));
uz=@(x,y,z,t) -d(3)*a(1)*fp(-(x*d(1)+y*d(2)+z*d(3))+c*(t-tlag));
uxx=@(x,y,z,t) d(1)^2*a(1)*fpp(-(x*d(1)+y*d(2)+z*d(3))+c*(t-tlag));
uyy=@(x,y,z,t) d(2)^2*a(1)*fpp(-(x*d(1)+y*d(2)+z*d(3))+c*(t-tlag));
uzz=@(x,y,z,t) d(3)^2*a(1)*fpp(-(x*d(1)+y*d(2)+z*d(3))+c*(t-tlag));
uxy=@(x,y,z,t) d(1)*d(2)*a(1)*fpp(-(x*d(1)+y*d(2)+z*d(3))+c*(t-tlag));
uxz=@(x,y,z,t) d(1)*d(3)*a(1)*fpp(-(x*d(1)+y*d(2)+z*d(3))+c*(t-tlag));
uyz=@(x,y,z,t) d(2)*d(3)*a(1)*fpp(-(x*d(1)+y*d(2)+z*d(3))+c*(t-tlag));
ut =@(x,y,z,t) a(1)*c*fp(-(x*d(1)+y*d(2)+z*d(3))+c*(t-tlag));
utt=@(x,y,z,t) a(1)*c^2*fpp(-(x*d(1)+y*d(2)+z*d(3))+c*(t-tlag));

% v
v=@(x,y,z,t)  a(2)*f(-(x*d(1)+y*d(2)+z*d(3))+c*(t-tlag));
vx=@(x,y,z,t) -d(1)*a(2)*fp(-(x*d(1)+y*d(2)+z*d(3))+c*(t-tlag));
vy=@(x,y,z,t) -d(2)*a(2)*fp(-(x*d(1)+y*d(2)+z*d(3))+c*(t-tlag));
vz=@(x,y,z,t) -d(3)*a(2)*fp(-(x*d(1)+y*d(2)+z*d(3))+c*(t-tlag));
vxx=@(x,y,z,t) d(1)^2*a(2)*fpp(-(x*d(1)+y*d(2)+z*d(3))+c*(t-tlag));
vyy=@(x,y,z,t) d(2)^2*a(2)*fpp(-(x*d(1)+y*d(2)+z*d(3))+c*(t-tlag));
vzz=@(x,y,z,t) d(3)^2*a(2)*fpp(-(x*d(1)+y*d(2)+z*d(3))+c*(t-tlag));
vxy=@(x,y,z,t) d(1)*d(2)*a(2)*fpp(-(x*d(1)+y*d(2)+z*d(3))+c*(t-tlag));
vxz=@(x,y,z,t) d(1)*d(3)*a(2)*fpp(-(x*d(1)+y*d(2)+z*d(3))+c*(t-tlag));
vyz=@(x,y,z,t) d(2)*d(3)*a(2)*fpp(-(x*d(1)+y*d(2)+z*d(3))+c*(t-tlag));
vt =@(x,y,z,t) a(2)*c*fp(-(x*d(1)+y*d(2)+z*d(3))+c*(t-tlag));
vtt=@(x,y,z,t) a(2)*c^2*fpp(-(x*d(1)+y*d(2)+z*d(3))+c*(t-tlag));

% w
w=@(x,y,z,t)  a(3)*f(-(x*d(1)+y*d(2)+z*d(3))+c*(t-tlag));
wx=@(x,y,z,t) -d(1)*a(3)*fp(-(x*d(1)+y*d(2)+z*d(3))+c*(t-tlag));
wy=@(x,y,z,t) -d(2)*a(3)*fp(-(x*d(1)+y*d(2)+z*d(3))+c*(t-tlag));
wz=@(x,y,z,t) -d(3)*a(3)*fp(-(x*d(1)+y*d(2)+z*d(3))+c*(t-tlag));
wxx=@(x,y,z,t) d(1)^2*a(3)*fpp(-(x*d(1)+y*d(2)+z*d(3))+c*(t-tlag));
wyy=@(x,y,z,t) d(2)^2*a(3)*fpp(-(x*d(1)+y*d(2)+z*d(3))+c*(t-tlag));
wzz=@(x,y,z,t) d(3)^2*a(3)*fpp(-(x*d(1)+y*d(2)+z*d(3))+c*(t-tlag));
wxy=@(x,y,z,t) d(1)*d(2)*a(3)*fpp(-(x*d(1)+y*d(2)+z*d(3))+c*(t-tlag));
wxz=@(x,y,z,t) d(1)*d(3)*a(3)*fpp(-(x*d(1)+y*d(2)+z*d(3))+c*(t-tlag));
wyz=@(x,y,z,t) d(2)*d(3)*a(3)*fpp(-(x*d(1)+y*d(2)+z*d(3))+c*(t-tlag));
wt =@(x,y,z,t) a(3)*c*fp(-(x*d(1)+y*d(2)+z*d(3))+c*(t-tlag));
wtt=@(x,y,z,t) a(3)*c^2*fpp(-(x*d(1)+y*d(2)+z*d(3))+c*(t-tlag));

end
```

4 FEM computation of stresses

4.1 Norm of the stress tensor at the barycenters

Goal. With this function we numerically compute

$$\|\sigma_h\|_2 = \sqrt{\sigma_{hxx}^2 + \sigma_{hyy}^2 + \sigma_{hzz}^2 + 2\sigma_{hxy}^2 + 2\sigma_{hxz}^2 + 2\sigma_{hyz}^2}$$

at the barycenters of the elements for given $\mathbf{u}_h \in \mathbf{V}_h$, and variable material properties μ, λ. We are using the same idea used in `errorFEM3D(u,uh,T,k)` while calculating the H^1 seminorm. First we start from u_{tot}^h, and decompose it to u_h, v_h, w_h. Then we define $U_h = \mathtt{u_h}(\mathtt{DOF3D}(\mathtt{T},\mathtt{k}))$ and V_h, W_h same way.

Computation of partial derivatives Instead of evaluating $\partial_j u$ at all the quadrature points, we are going to evaluate it at the barycenters. To differentiate u_h note that if

$$\begin{aligned} \mathrm{C}^K &= (\mathrm{B}^K)^{-1} = \begin{bmatrix} c_{11}^K & c_{12}^K & c_{13}^K \\ c_{21}^K & c_{22}^K & c_{23}^K \\ c_{31}^K & c_{32}^K & c_{33}^K \end{bmatrix} \\ &= \frac{1}{6|K|} \begin{bmatrix} y_{13}z_{14} - y_{14}z_{13} & z_{13}x_{14} - z_{14}x_{13} & x_{13}y_{14} - x_{14}y_{13} \\ y_{14}z_{12} - y_{12}z_{14} & z_{14}x_{12} - z_{12}x_{14} & x_{14}y_{12} - x_{12}y_{14} \\ y_{12}z_{13} - y_{13}z_{12} & z_{12}x_{13} - z_{13}x_{12} & x_{12}y_{13} - x_{13}y_{12} \end{bmatrix}, \end{aligned}$$

(the matrix C^K used in the computation of the stiffness matrices is scaled slightly differently), then

$$(\nabla u_h|_K)(\mathbf{b}^K) = \sum_{\ell=1}^{d_k} u_\ell^K \mathbf{C}_K^\top \nabla P_\ell(\widehat{\mathbf{b}})$$

and therefore, denoting

$$\partial_j u_h|_K(\mathbf{b}^K) = \sum_{\ell=1}^{d_k} u_\ell^K \sum_{i=1}^{3} c_{ij}^K \partial_i P_\ell(\widehat{\mathbf{b}}).$$

where $\mathbf{b}^K$ is the barycenter of the element K. So the barycenter of reference element is $\widehat{\mathbf{b}} = (\frac{1}{4}, \frac{1}{4}, \frac{1}{4})$ Storing the coefficients c_{ik}^K in *row vectors* $\mathbf{c}_{ij} \in \mathbb{R}^{N_{\mathrm{elt}}}$ and evaluating the derivatives of the basis functions in matrices

$$\mathrm{P}_\ell^i = \partial_i P_\ell(\widehat{\mathbf{b}}) \quad \ell = 1, \ldots, d_k, \qquad i = 1, 2, 3,$$

we can compute everything quite fast. The N_{elt} sized row vector with the values of $\partial_j u_h$ at all the quadrature points is given by

$$u_{hj}(:,k) = \sum_{i=1}^{3} \mathrm{P}^i(\mathbf{c}_{ji} \odot \mathrm{U}_h(:,k)) \quad k = 1, \ldots, N_{\mathrm{time}} \text{ and } \; j = x, y, z$$

Using this idea we loop over time to calculate $u_{hx}, u_{hy}, u_{hz}, v_{hx}, v_{hy}, v_{hz}, w_{hx}, w_{hy}, w_{hz}$ which are matrices of size $N_{\mathrm{elt}} \times N_{\mathrm{time}}$ then we calculate the components of numerical stress tensor evaluated at barycenters.

$$\begin{aligned} \sigma_{hxx} &= 2\mu(\mathbf{b}) \odot u_{hx} + \lambda(\mathbf{b}) \odot (u_{hx} + v_{hy} + w_{hz}) \\ \sigma_{hyy} &= 2\mu(\mathbf{b}) \odot v_{hy} + \lambda(\mathbf{b}) \odot (u_{hx} + v_{hy} + w_{hz}) \\ \sigma_{hzz} &= 2\mu(\mathbf{b}) \odot w_{hz} + \lambda(\mathbf{b}) \odot (u_{hx} + v_{hy} + w_{hz}) \\ \sigma_{hxy} &= \lambda(\mathbf{b}) \odot (v_{hx} + u_{hy}) \\ \sigma_{hxz} &= \lambda(\mathbf{b}) \odot (w_{hx} + u_{hz}) \\ \sigma_{hyz} &= \lambda(\mathbf{b}) \odot (w_{hy} + v_{hz}) \end{aligned}$$

where $\mu(\mathbf{b})$ and $\lambda(\mathbf{b})$ are N_{elt} sized vectors corresponding to the values of the functions μ and λ evaluated at barycenters of elements, $\mathbf{b}$. Then the function returns the norm of the stress tensor which is

$$\|\sigma_h\|_2 = \sqrt{\sigma_{hxx}^2 + \sigma_{hyy}^2 + \sigma_{hzz}^2 + 2\sigma_{hxy}^2 + 2\sigma_{hxz}^2 + 2\sigma_{hyz}^2}$$

```
function sigma=stressAtBaryc(mu,lam,uhtot,T,k)

% sigma=stressAtBaryc(mu,lambda,uhtot,T,k)
% sigma=stressAtBaryc(mu,lambda,{uhtot, Ntime},T,k)
%
% Input:
%    mu,lam: Vectorized function handles of three variables
%    uh    : dim FE-vector for a FE function
%    T     : basic triangulation
%    k     : polynomial degree
% Output:
%    sigma: Nelt x Ntime sized matrix valued stress
%           at barycenter of elements at each time steps
%
% Last modified: July 1, 2016 by HE

Ne = size(T.elements,2);

% Geometric coefficients for change of variables
x12=T.coordinates(1,T.elements(2,:))-T.coordinates(1,T.elements(1,:));
y12=T.coordinates(2,T.elements(2,:))-T.coordinates(2,T.elements(1,:));
z12=T.coordinates(3,T.elements(2,:))-T.coordinates(3,T.elements(1,:));
x13=T.coordinates(1,T.elements(3,:))-T.coordinates(1,T.elements(1,:));
y13=T.coordinates(2,T.elements(3,:))-T.coordinates(2,T.elements(1,:));
z13=T.coordinates(3,T.elements(3,:))-T.coordinates(3,T.elements(1,:));
x14=T.coordinates(1,T.elements(4,:))-T.coordinates(1,T.elements(1,:));
y14=T.coordinates(2,T.elements(4,:))-T.coordinates(2,T.elements(1,:));
z14=T.coordinates(3,T.elements(4,:))-T.coordinates(3,T.elements(1,:));

% Entries of CK
sqdet=1./(6*T.volume);
c11=sqdet.*(y13.*z14 - y14.*z13);
c12=sqdet.*(x14.*z13 - x13.*z14);
c13=sqdet.*(x13.*y14 - x14.*y13);
c21=sqdet.*(y14.*z12 - y12.*z14);
c22=sqdet.*(x12.*z14 - x14.*z12);
c23=sqdet.*(x14.*y12 - x12.*y14);
c31=sqdet.*(y12.*z13 - z12.*y13);
c32=sqdet.*(x13.*z12 - x12.*z13);
c33=sqdet.*(x12.*y13 - x13.*y12);

% Calculating the coordinates of the barycenter
baryc=1/4*(T.coordinates(:,T.elements(1,:))+...
    T.coordinates(:,T.elements(2,:))+...
    T.coordinates(:,T.elements(3,:))+...
    T.coordinates(:,T.elements(4,:)));
barycx=baryc(1,:).'; barycy=baryc(2,:).'; barycz=baryc(3,:).';
[Px,Py,Pz]=bernsteinDer3D(1/4,1/4,1/4,k);

% Decomposing uhtot for several cases
if iscell(uhtot)
    switch length(uhtot)
        case 2
            uhtotal = uhtot{1};
            Ntime = uhtot{2};
            t=2*pi*(1:Ntime)./Ntime;
            len = size(uhtotal,1)/3;
            u = uhtotal(1:len,:);
            v = uhtotal(len+1:2*len,:);
```

```
            w = uhtotal(2*len+1:end,:);
            uh=bsxfun(@times,real(u),cos(t))+bsxfun(@times,imag(u),sin(t));
            vh=bsxfun(@times,real(v),cos(t))+bsxfun(@times,imag(v),sin(t));
            wh=bsxfun(@times,real(w),cos(t))+bsxfun(@times,imag(w),sin(t));
        case 3
            uh = uhtot{1};
            vh = uhtot{2};
            wh = uhtot{3};
            Ntime = size(uh,2);
        case 4
            u = uhtot{1};
            v = uhtot{2};
            w = uhtot{3};
            Ntime = uhtot{4};
            t=2*pi*(1:Ntime)./Ntime;
            uh=bsxfun(@times,real(u),cos(t))+bsxfun(@times,imag(u),sin(t));
            vh=bsxfun(@times,real(v),cos(t))+bsxfun(@times,imag(v),sin(t));
            wh=bsxfun(@times,real(w),cos(t))+bsxfun(@times,imag(w),sin(t));
    end
else
    len = size(uhtot,1)/3;
    uh = uhtot(1:len,:);
    vh = uhtot(len+1:2*len,:);
    wh = uhtot(2*len+1:end,:);
    Ntime = size(uh,2);
end
dof = DOF3D(T,k);

% Calculating the partial derivatives at each time step
uhx = zeros(Ne,Ntime);
uhy = zeros(Ne,Ntime);
uhz = zeros(Ne,Ntime);
vhx = zeros(Ne,Ntime);
vhy = zeros(Ne,Ntime);
vhz = zeros(Ne,Ntime);
whx = zeros(Ne,Ntime);
why = zeros(Ne,Ntime);
whz = zeros(Ne,Ntime);

for j=1:Ntime
    Uh=uh(:,j); Uh = Uh(dof);
    Vh=vh(:,j); Vh = Vh(dof);
    Wh=wh(:,j); Wh = Wh(dof);

    uhx(:,j)=((Px*bsxfun(@times,c11,Uh)+Py*bsxfun(@times,c21,Uh)+...
        Pz*bsxfun(@times,c31,Uh))).';
    uhy(:,j)=((Px*bsxfun(@times,c12,Uh)+Py*bsxfun(@times,c22,Uh)+...
        Pz*bsxfun(@times,c32,Uh))).';
    uhz(:,j)=((Px*bsxfun(@times,c13,Uh)+Py*bsxfun(@times,c23,Uh)+...
        Pz*bsxfun(@times,c33,Uh))).';

    vhx(:,j)=((Px*bsxfun(@times,c11,Vh)+Py*bsxfun(@times,c21,Vh)+...
        Pz*bsxfun(@times,c31,Vh))).';
    vhy(:,j)=((Px*bsxfun(@times,c12,Vh)+Py*bsxfun(@times,c22,Vh)+...
        Pz*bsxfun(@times,c32,Vh))).';
    vhz(:,j)=((Px*bsxfun(@times,c13,Vh)+Py*bsxfun(@times,c23,Vh)+...
        Pz*bsxfun(@times,c33,Vh))).';

    whx(:,j)=((Px*bsxfun(@times,c11,Wh)+Py*bsxfun(@times,c21,Wh)+...
        Pz*bsxfun(@times,c31,Wh))).';
    why(:,j)=((Px*bsxfun(@times,c12,Wh)+Py*bsxfun(@times,c22,Wh)+...
        Pz*bsxfun(@times,c32,Wh))).';
    whz(:,j)=((Px*bsxfun(@times,c13,Wh)+Py*bsxfun(@times,c23,Wh)+...
        Pz*bsxfun(@times,c33,Wh))).';
end
```

```
% Using the partial derivatives calculating the stress
mubaryc = mu(barycx,barycy,barycz);
lambdabaryc = lam(barycx,barycy,barycz);
div=uhx+vhy+whz;

sxx=2*bsxfun(@times,mubaryc,uhx) + bsxfun(@times,lambdabaryc,div);
syy=2*bsxfun(@times,mubaryc,vhy) + bsxfun(@times,lambdabaryc,div);
szz=2*bsxfun(@times,mubaryc,whz) + bsxfun(@times,lambdabaryc,div);

sxy=bsxfun(@times,mubaryc,(vhx+uhy));
sxz=bsxfun(@times,mubaryc,(whx+uhz));
syz=bsxfun(@times,mubaryc,(why+vhz));


sigma=sqrt(sxx.^2+syy.^2+szz.^2+2*sxy.^2+2*sxz.^2+2*syz.^2);

return
```

4.2 Stress postprocessing

Note. The order of the components of the output stress may be different than the usual ordering in other functions. Please be aware of that.

The function `stressPostprocessing.m` takes an FEM approximation $\mathbf{u}_h \in V_h^3$ of the displacement field, and computes an approximation of the stress components as a function in the space

$$\{v : \Omega \to \mathbb{R} \,:\, v|_K \in \mathbb{P}_{k-1}(K) \quad \forall K \in \mathcal{T}_h\} \equiv \prod_{K\in\mathcal{T}_h} \mathbb{P}_{k-1}(K),$$

using the $L^2(\Omega)$ orthogonal projection of the components of

$$\boldsymbol{\sigma}_h = \mu(\nabla \mathbf{u}_h + \nabla \mathbf{u}_h^\top) + \lambda(\nabla \cdot \mathbf{u}_h)\mathrm{I}.$$

As a first step the stress is computed in all quadrature points of all the elements

$$\boldsymbol{\sigma}_h(\mathbf{p}_q^K) = 2\mu(\mathbf{p}_q^K)\varepsilon(\mathbf{u}(\mathbf{p}_q^K)) + \lambda(\mathbf{p}_q^K)\mathrm{tr}\ \varepsilon(\mathbf{u}(\mathbf{p}_q^K))I_{3\times 3}.$$

This evaluation is done in several steps:

- First λ and μ are evaluated in the $N_{\text{quad}} \times N_{\text{elt}}$ collection of all quadrature points.
- The derivatives of u_h, v_h and w_h with respect to the three variables are then evaluated at all quadrature points. This is done in the same was as in `errorFEM3D` (see Section 5.8 The error function in FEM3DDocumentation), by evaluating the basis functions in the quadrature points of the reference element, and using geometric coefficients.
- Finally the six components of $\boldsymbol{\sigma}_h(\mathbf{p}_q^K)$ are produced. We thus have six $N_{\text{quad}} \times N_{\text{elt}}$ matrices.

The L^2 projection is local, element by element, and can be carried out simultaneously for all elements and all components of σ_h. Mathematically speaking, we need to find approximations (for all $\alpha, \beta \in \{1,2,3\}$ and $K \in \mathcal{T}_h$)

$$\sigma_{\alpha,\beta}^{h,K} \in \mathbb{P}_{k-1} \qquad \int_K R_i^K \sigma_{\alpha,\beta}^{h,K} = \int_K R_i^K \sigma_{\alpha,\beta} \qquad i = 1,\ldots,d_{k-1},$$

where $\{R_i\}$ is a basis for $\mathbb{P}_{k-1}$. This is equivalent to solving systems

$$\sum_q |K|\widehat{\omega}_q R_i(\widehat{\mathbf{p}}_q) R_j(\widehat{\mathbf{p}}_q)\sigma_{\alpha\beta}^{ij} = \sum_q |K|\widehat{\omega}_q R_i(\widehat{\mathbf{p}}_q)\sigma_{\alpha\beta}(\mathbf{p}_q^K).$$

(Note that $|K|$ can be simplified in the above expression, making the matrix in the left-hand side independent of K.) Let R and R_ω be the matrices

$$\mathrm{R}_{q,i} = R_i(\widehat{\mathbf{p}}_q), \qquad (\mathrm{R}_\omega)_{q,i} = \widehat{\omega}_q R_i(\widehat{\mathbf{p}}_q), \qquad q = 1, \ldots, N_{\mathrm{quad}}, \quad i = 1, \ldots, d_{k-1}.$$

We then have to compute

$$\boldsymbol{\Sigma}_{\mathrm{proj}} := (\mathrm{R}_\omega^\top \mathrm{R})^{-1} \mathrm{R}_\omega^\top \boldsymbol{\Sigma},$$

where $\boldsymbol{\Sigma}$ is the $N_{\mathrm{quad}} \times (6N_{\mathrm{elt}})$ matrix with the values of all $\sigma_{\alpha\beta}$ at all the quadrature points of all the elements. This small system with a large number of right-hand-sides provides a $d_{k-1} \times (6N_{\mathrm{elt}})$ matrix with the coefficients of the approximations $\sigma^h_{\alpha\beta}$ at each of the elements. The order of the columns of $\boldsymbol{\Sigma}$ is not relevant for the computation and only needs to be kept to unpack the approximations of the six components that we have computed in $\boldsymbol{\Sigma}_{\mathrm{proj}}$.
Note finally, that we output these quantities as 6 vectors of size $d_{k-1}\, N_{\mathrm{elt}}$ and that we order them using Voigt's notation

$$\begin{bmatrix} \sigma_1 & \sigma_6 & \sigma_5 \\ \times & \sigma_2 & \sigma_4 \\ \times & \times & \sigma_3 \end{bmatrix}$$

Alternatively, we can pass an eight argument into `stressPostprocessing.m`, which is used as a flag, indicating that we should output the element-by-element average value of these approximations, i.e.

$$\frac{1}{|K|} \int_K \sigma^h_{\alpha,\beta} \approx \sum_q \sum_{i=1}^{\dim \mathbb{P}_{k-1}} \widehat{\omega}_q R_i(\widehat{\mathbf{p}}^q) \sigma^K_{\alpha,\beta,i}.$$

This computation can be easily done by adding the rows of the $N_{\mathrm{quad}} \times (6N_{\mathrm{elt}})$ matrix

$$\mathrm{R}_\omega \boldsymbol{\Sigma}_{\mathrm{proj}}.$$

```
function [sigmaxx,sigmayy,sigmazz,sigmayz,sigmaxz,sigmaxy]=...
                     stressPostprocessing(uh,vh,wh,lambda,mu,T,k,varargin)
% [sigmaxx,sigmayy,sigmazz,sigmayz,sigmaxz,sigmaxy]=...
%                     stressPostprocessing(uh,vh,wh,lambda,mu,T,k)
% [sigmaxx,sigmayy,sigmazz,sigmayz,sigmaxz,sigmaxy]=...
%                     stressPostprocessing(uh,vh,wh,lambda,mu,T,k,tag)
%
% Input:
%    uh          : dim FE-vector for a FE function (1st component)
%    vh          : dim FE-vector for a FE function (2nd component)
%    wh          : dim FE-vector for a FE function (3rd component)
%    lambda      : vectorized function of three variables
%    mu          : vectorized function of three variables
%    T           : basic triangulation
%    k           : polynomial degree
%    tag         : an indicator that you want the averaged stress on each
%                  element rather than the value on nodes (can be anything)
%
% Output:
%    sigmaxx,... : dimP_{k-1}*Nelt x 1. Each represents for coefficients
%                  of stress in piecewise P_{k-1} polynomial space
%                  or
%                  1 x Nelt. Each is a row vector of
%                  averaged stress on each element
%
% Last modified: Nov 21, 2017

T=edgesAndFaces(T);
T=enhanceGrid3D(T);
Nelt=size(T.elements,2);
```

```
formula=quadratureFEM(4*k,3);
x=T.coordinates(1,:); x=formula(:,1:4)*x(T.elements);
y=T.coordinates(2,:); y=formula(:,1:4)*y(T.elements);
z=T.coordinates(3,:); z=formula(:,1:4)*z(T.elements);

Mu=mu(x,y,z);
La=lambda(x,y,z);

% Geometric coefficients for change of variables

x12=T.coordinates(1,T.elements(2,:))-T.coordinates(1,T.elements(1,:));
y12=T.coordinates(2,T.elements(2,:))-T.coordinates(2,T.elements(1,:));
z12=T.coordinates(3,T.elements(2,:))-T.coordinates(3,T.elements(1,:));
x13=T.coordinates(1,T.elements(3,:))-T.coordinates(1,T.elements(1,:));
y13=T.coordinates(2,T.elements(3,:))-T.coordinates(2,T.elements(1,:));
z13=T.coordinates(3,T.elements(3,:))-T.coordinates(3,T.elements(1,:));
x14=T.coordinates(1,T.elements(4,:))-T.coordinates(1,T.elements(1,:));
y14=T.coordinates(2,T.elements(4,:))-T.coordinates(2,T.elements(1,:));
z14=T.coordinates(3,T.elements(4,:))-T.coordinates(3,T.elements(1,:));

sqdet=1./(6*T.volume);
c11=sqdet.*(y13.*z14 - y14.*z13);
c12=sqdet.*(x14.*z13 - x13.*z14);
c13=sqdet.*(x13.*y14 - x14.*y13);
c21=sqdet.*(y14.*z12 - y12.*z14);
c22=sqdet.*(x12.*z14 - x14.*z12);
c23=sqdet.*(x14.*y12 - x12.*y14);
c31=sqdet.*(y12.*z13 - z12.*y13);
c32=sqdet.*(x13.*z12 - x12.*z13);
c33=sqdet.*(x12.*y13 - x13.*y12);

% Evaluation of all stresses at all quadrature points

Uh=uh(DOF3D(T,k));
Vh=vh(DOF3D(T,k));
Wh=wh(DOF3D(T,k));
[Px,Py,Pz]=bernsteinDer3D(formula(:,2),formula(:,3),formula(:,4),k);

uhx=Px*bsxfun(@times,c11,Uh)+Py*bsxfun(@times,c21,Uh)...
        +Pz*bsxfun(@times,c31,Uh);
uhy=Px*bsxfun(@times,c12,Uh)+Py*bsxfun(@times,c22,Uh)...
        +Pz*bsxfun(@times,c32,Uh);
uhz=Px*bsxfun(@times,c13,Uh)+Py*bsxfun(@times,c23,Uh)...
        +Pz*bsxfun(@times,c33,Uh);

vhx=Px*bsxfun(@times,c11,Vh)+Py*bsxfun(@times,c21,Vh)...
        +Pz*bsxfun(@times,c31,Vh);
vhy=Px*bsxfun(@times,c12,Vh)+Py*bsxfun(@times,c22,Vh)...
        +Pz*bsxfun(@times,c32,Vh);
vhz=Px*bsxfun(@times,c13,Vh)+Py*bsxfun(@times,c23,Vh)...
        +Pz*bsxfun(@times,c33,Vh);

whx=Px*bsxfun(@times,c11,Wh)+Py*bsxfun(@times,c21,Wh)...
        +Pz*bsxfun(@times,c31,Wh);
why=Px*bsxfun(@times,c12,Wh)+Py*bsxfun(@times,c22,Wh)...
        +Pz*bsxfun(@times,c32,Wh);
whz=Px*bsxfun(@times,c13,Wh)+Py*bsxfun(@times,c23,Wh)...
        +Pz*bsxfun(@times,c33,Wh);

sigmaxx=2*Mu.*uhx+La.*(uhx+vhy+whz);
sigmayy=2*Mu.*vhy+La.*(uhx+vhy+whz);
sigmazz=2*Mu.*whz+La.*(uhx+vhy+whz);
sigmaxy=Mu.*(uhy+vhx);
sigmaxz=Mu.*(uhz+whx);
sigmayz=Mu.*(vhz+why);
```

```
Sigmas=[sigmaxx,sigmayy,sigmazz,sigmaxy,sigmaxz,sigmayz];

% L2 projection element by element

R =bernstein3D(formula(:,2),formula(:,3),formula(:,4),k-1);
wR=bsxfun(@times,formula(:,5),R);
SigmaProj=(R'*wR)\(wR'*Sigmas);

if nargin > 7
    SigmaProj = sum(wR*SigmaProj);
    sigmaxx = SigmaProj(1:Nelt);
    sigmayy=SigmaProj(Nelt+1:2*Nelt);
    sigmazz=SigmaProj(2*Nelt+1:3*Nelt);
    sigmaxy=SigmaProj(3*Nelt+1:4*Nelt);
    sigmaxz=SigmaProj(4*Nelt+1:5*Nelt);
    sigmayz=SigmaProj(5*Nelt+1:6*Nelt);
else
    sigmaxx=SigmaProj(:,1:Nelt);
    sigmayy=SigmaProj(:,Nelt+1:2*Nelt);
    sigmazz=SigmaProj(:,2*Nelt+1:3*Nelt);
    sigmaxy=SigmaProj(:,3*Nelt+1:4*Nelt);
    sigmaxz=SigmaProj(:,4*Nelt+1:5*Nelt);
    sigmayz=SigmaProj(:,5*Nelt+1:6*Nelt);
    sigmaxx=sigmaxx(:);
    sigmayy=sigmayy(:);
    sigmazz=sigmazz(:);
    sigmaxy=sigmaxy(:);
    sigmaxz=sigmaxz(:);
    sigmayz=sigmayz(:);
end


return
```

5 Plotting utilities

5.1 Plotting mesh

If `T` is a tetrahedrization and f is a function, `meshPlot(T,f)` plots the tetrahedrization (using `tetramesh`) where each element gets the color from the value of f at the barycenter of the element. Another way to use this function is by bringing a row vector $1 \times N_{\text{elt}}$. In addition, if a 1 is passed in as the third argument, i.e. `meshPlot(T,f,1)`, then the plot will show the normal vectors of the mesh as well.

```
function meshPlot(T, f, varargin )
%meshPlot(T, f )
%meshPlot(T, f, 1)
% Inputs:
%       T : an enhanced tetrahedrization
%       f : either a function of three variables or a row vector which
%           is used as the colormap of the plot
%       varargin : type 1 to plot the normals
%
% Last Modified: November 4, 2016

if size(f,2) > 1
    tetramesh(T.elements',T.coordinates',f);
else
    bary = (T.coordinates(:, T.elements(1,:))...
          + T.coordinates(:, T.elements(2,:))...
          + T.coordinates(:, T.elements(3,:))...
          + T.coordinates(:, T.elements(4,:)))/4;
```

```
    tetramesh(T.elements',T.coordinates',f(bary(1,:),bary(2,:),bary(3,:)));
end

if nargin == 3 && varargin{1} == 1
    baryFace = (T.coordinates(:,T.faces(1,:))...
              + T.coordinates(:,T.faces(2,:))...
              + T.coordinates(:,T.faces(3,:)))/3;
    hold on
    quiver3(baryFace(1,:),baryFace(2,:),baryFace(3,:),...
            T.normals(1,:),T.normals(2,:),T.normals(3,:));
    hold off
end


end
```

5.2 Plotting the solution on the volume mesh

Goal. Given a 3D FEM solution ($3 \dim V_h \times N_{\text{time}}$ matrix) and numerical stress σ_h ($\dim V_h \times N_{\text{time}}$ matrix) creating an animation which presents the displacement of the mesh and color change on the elements. To make the plotting function faster, we are only plotting the boundary of the mesh. Also see `stressPostprosessing` for how to obtain numerical stress per element. This function can do the following:

- Plotting the displacement at the equilibrium state
- Saving the animation of the evolution of the displacement
- Saving some snapshots of the displacement

For each of these options the plotting can be done in two ways

- With a fixed color per element
- With a color showing the average change of stress per element

Now we will explain the process and options in details.

Decomposing. Our input is $3 \dim V_h \times N_{\text{time}}$ matrix

$$\left[\begin{array}{c} u_h \\ \hline v_h \\ \hline w_h \end{array}\right]$$

which is decomposed it into 3 sub-matrices u_h, v_h, w_h.

Vertices from DOF. In order to plot the displacement of the mesh, we need to add only the displacements of the vertices to the mesh coordinates i.e. the values of u_h, v_h, w_h which are corresponding the vertex indices of DOF to the x, y, z coordinates of the mesh. In order to do that we take the first 4 rows of the `DOF3D(T,k)` which corresponds the vertex indices. Then, consider the values of u_h, v_h, w_h at those indices.

Finding faces. To find the face coloring from given numerical stress per element we do the following.

- Create an *outer face list* which contains first Dirichlet then Neumann faces. This information is given in enhanced triangulation of a mesh. Size of this list is $3 \times N_{\text{of}}$ where N_{of} is the number of outer faces of the mesh. Each column of the outer face list corresponds to the vertices of a face.

- We obtain the global numbering of these faces from the face list of the mesh. To do that we simply look at the intersection of the columns of the outer face list and the general face list.
- Using that global numbering of outer face list and `facebyelt` component of the mesh we obtain a $1 \times N_{\text{of}}$ vector which gives us the elements contain the outer faces. Note that there is only one element per outer face. Finally, we assign the color of the corresponding element to that outer face.
- These steps are done without looping, using the MATLAB function `intersect`.

Coloring. To obtain a uniform color change through the animation, we calculate the maximum and minimum color of all outer faces at all times. Then we scale the coloring with the function `caxis`.

Saving the movie. The movies are saved in the format of MPEG-4 with 100% quality. It is saved in '/movies' folder if it exists or in the current path. If you want to save the movie with the default settings you can simply pass an empty cell array in the `movie` parameter. Otherwise the following options are given

- **Rotation.** It is either 1 or 0 (on or off). When rotation is on, it gives a one complete 360° rotation around the object by the time movie ends. Default is off. (Note that with the current version of this function during the rotation, some of the camera properties are automatically scaled by `view` function which causes some continuous zooming during rotation. For details, see camera graphics terminology of MATLAB and properties of camera position of `view` function)
- **Camera Angle.** This is a 2 dimensional row vector corresponding to azimuth and elevation angles in radians. Default is $(-37.5°, 30°)$. (See MATLAB `view`)
- **Color.** There are several options
  - `'flat'`: This assumes that there is a given coloring (numerical stress) and from that coloring one color per face is assigned. (See MATLAB `'FaceColor'` properties.)
  - Color scale: This also assumes there is a given coloring. It assigns the face colors using the given scaling option; e.g. `'grayscale', 'hotscale'`. You simply put `'scale'` after the name of the scale you want. Default scale is `'parula'`. (See MATLAB `colormap` names)
  - Name of a color: A unique color for the entire surface. You can pass the color names defined by MATLAB; e.g. `'red', 'green'`. (See `ColorSpec`)
  - RGB vector: A unique color for the entire surface. You need to pass a 1×3 vector with real entires from $[0, 1]$; e.g. $(1, 0.3, 0.8)$.

  Default is `'flat'`.
- **Axis lines.** Either 1 or 0 (on or off). When it is on, it also shows the axis labels x, y, z. Default is off.
- **Edge line style.** This determines how to display the edges of the mesh. When it is on, the edge color is always black. You can pass in the following
  - `'none'`: No edge display
  - `'-'`: Edges are straight lines
  - `':'`: Edges are dotted lines

  Default is straight lines. (See MATLAB Line Style Specifiers for more options)
- **Frame rate.** This specifies the number of frames per second. It has to be a natural number. Smaller numbers make the movie slower. Default is 30 frames per second.

Saving the snapshots. When sixth input is given, the snapshot saving mode is on. Instead of saving a video, some of the snapshots are saved in the designated folder in *png* format (*eps* requires incredible amount of storage which causes a crash of the system). There are two options

- Saving all snapshots. A string `'all'` should be passed in for this option.
- Saving specific snapshots. An integer vector array $(k_1, \cdots, k_N)$ should be passed in. In that case the function saves the following snapshots

$$(u_h^{k_i}, v_h^{k_i}, w_h^{k_i}) \qquad i = 1, \cdots, N$$

using the colors

$$\sigma_h^{k_i} \qquad i = 1, \cdots, N$$

where $\star_h^k$ indicates the value of $\star_h$ at the time step k for $\star \in \{u, v, w, \sigma\}$.

```
function plotElasticity3D(uhtot,T,k,varargin)
% plotElasticity3D(uhtot,T,k)
% plotElasticity3D(uhtot,T,k,sigmah)
% plotElasticity3D(uhtot,T,k,sigmah,{})
%     OR     plotElasticity3D(uhtot,T,k,sigmah,movie)
% plotElasticity3D(uhtot,T,k,sigmah,movie,snapshots)
%
% Input:
%   uhtot     :      3*dimVh x Ntimes complex vector (containing uh,vh,wh)
%   T         :      Data structure. Enhanced FEM triangulation
%   k         :      Scalar. Polynomial degree
%   sigmah    :      Nelts x Ntimes numerical norm of stress at barycenter
%                    of elements. If stress is unknown, and uniform coloring
%                    is desired pass in a scalar i.e. 1.
%   movie     :      Cell element containing the following
%                    (for default you can pass in {})
%   movie{1} :       Rotation. This can be 1 or 0, enables or disables
%                      camera rotation.
%                      Default: no rotation
%   movie{2} :       Camera angle. 1x2 vector corresponding to
%                      [azimuth elevation] in radians.
%                      Input option: 'default'
%                      Default: [-37.5,30]
%   movie{3} :       Color for the faces in MATLAB RGB vector,
%                      Input options: 'default' OR 'flat',
%                       OR 'grayscale', 'hotscale, ''winterscale',...
%                       OR 'green','red',...
%                       OR [1,0.3,0.8]
%                      'flat': Uniform colors taken from stress data
%                      'grayscale': Computes the grayscale of the
%                                   given coloring in the stress data
%                      'green': Uniform red color ignoring stress
%                      Default: 'flat'
%   movie{4} :       Axis on or off. This can be 1 or 0. When
%                      it is on it shows the labels too.
%                      Default: on
%   movie{5} :       Edge line style.
%                      Input options: '-'(solid line), 'none', '--'(dashed),
%                      ':' (dotted)
%                      Default: '-'
%   movie{6} :       Number of frames per second. A positive integer.
%                      Default: 30
%   snapshots:       Enables to save snaphots.
%                    Input options: 'all',index vector
%                      'all': Saves all time steps of the snaphots of the given
%                      data
```

```
%                     index vector: Saves the snapshots at the time steps
%                     corresponding to the index vector
%                   When you save snapshots, they are all saved in the
%                   designated folder with names: snapshot_1,snapshot_2,...
%                   in png format.
%
% Action: plot of ...
%   displaced mesh with constant coloring
%   displaced mesh colored by norm of stress
%   displaced mesh colored by norm of stress and saves the video
%   displaced mesh colored by norm of stress and saves the snapshots
%
% Last modified: February 1, 2018

% Finding vertices from DOF
Ne = size(T.elements,2);
dofuh = DOF3D(T,k);
vdof = unique(dofuh(1:4,:));
% in case of snapshots
iwantsnapshots=false;
if nargin≥6
    snapshots=varargin{3};
    if isnumeric(snapshots)
        uhtot=uhtot(:,snapshots);
    else
        snapshots=1:size(uhtot,2);
    end
    iwantsnapshots=true;
end
% Decomposing uhtot
len = size(uhtot,1)/3;
uh = uhtot(1:len,:);
vh = uhtot(len+1:2*len,:);
wh = uhtot(2*len+1:end,:);
Ntime = size(uh,2);
% Taking the values of uh which contributed by the vertices and adding the
% mesh coordinates
Uh=bsxfun(@plus,uh(vdof,:),T.coordinates(1,:)');
Vh=bsxfun(@plus,vh(vdof,:),T.coordinates(2,:)');
Wh=bsxfun(@plus,wh(vdof,:),T.coordinates(3,:)');

% Finding face numbering
faceList=[T.dirichlet T.neumann];
Nf=size(faceList,2);
[¬,¬,ib]=intersect(faceList.',T.faces(1:3,:).','rows','stable');
[¬,¬,ibb]=intersect(ib,T.facebyelt(:),'stable');
colno=(ceil(ibb/4));
% Determining the coloring via using stress
if nargin≥4
    C = varargin{1};
    if size(C,1)==Ne
        C=C(colno,:);
        if nargin≥6
            C=C(:,snapshots);
        end
        disp('Face coloring is derived from given col set.');
    end
    if size(C,2)≠Ntime
        disp('Uniform coloring is set.');
        C = ones(Nf,Ntime);
        if nargin>6
            C=C(:,snapshots);
        end
    end
else
    disp('Uniform coloring is set.');
```

```
    C = ones(Nf,Ntime);
end
maxc = max(max(C)) + eps; %adding a small number to avoid minc=maxc case
minc = min(min(C));

% Determining the movie options
iwantmovie=false;
rot=0;
angle=[-37.5,30];
facecol='flat';
axismode=true;
linestyle='-';
framerate=30;
colorscale=parula;
if nargin>5
    movie=varargin{2};
    iwantmovie=true;
    if ~isempty(movie)
        rot=movie{1};
        if length(movie)>1
            angle=movie{2};
            if strcmp(angle,'default')==1
                angle=[-37.5, 30];
            end
            if length(movie)>2
                facecol=movie{3};
                if strcmp(facecol,'default')==1
                    facecol='flat';
                end
                if ischar(facecol) && ~isempty(strfind(facecol,'scale'))%#ok
                    colorscale=strrep(facecol,'scale','');
                    facecol='flat';
                end
                if length(movie)>3
                    axismode=movie{4};
                    if length(movie)>4
                        linestyle=movie{5};
                        if length(movie)>5
                            framerate=movie{6};
                        end
                    end
                end
            end
        end
    end
end

if iwantsnapshots
    iwantmovie=false;
end

%creating the movie in the designated folder
if iwantmovie
    close all;
    DateAndTime=datetime('now');
    TimeStamp=datestr(datenum(DateAndTime),'yy-mm-dd-HH-MM');
    if exist([pwd '/movies'],'file') ~= 7
        disp('We can not find the /movies folder in current path.');
        FileStr=strcat(TimeStamp,'_FastMovie_k=',...
            num2str(k),'_M=',num2str(Ntime));
        writerObj = VideoWriter(FileStr,'MPEG-4');
        writerObj.Quality = 100;
        writerObj.FrameRate = framerate;
        open(writerObj);
        disp('Therefore, your videos is being saved in:');
        display(pwd);
```

```
    else
        FileStr=strcat('movies/',TimeStamp,'_FastMovie_k=',...
            num2str(k),'_M=',num2str(Ntime));
        writerObj = VideoWriter(FileStr,'MPEG-4');
        writerObj.Quality = 100;
        writerObj.FrameRate = framerate;
        open(writerObj);
        disp('Your videos is being saved in:');
        disp([pwd '/movies']);
    end
end

% Animation part
currentcolor=C(:,1);
plotHandle = @(u,v,w,Col) trisurf(faceList', u, v, w, Col,...
    'LineStyle',linestyle,...
    'FaceLighting','flat',...
    'FaceColor',facecol,...
    'AmbientStrength',0.3,...
    'DiffuseStrength', 0.8,...
    'SpecularStrength', 0.9,...
    'SpecularExponent', 10);
% Creating a full size figure
figure('units','normalized','outerposition',[0 0 1 1]);
shading interp
colormap(colorscale);
figH=plotHandle(Uh(:,1), Vh(:,1), Wh(:,1), currentcolor);
axis equal;
if ¬iwantsnapshots
    title(['time step = ' num2str(1) '/' num2str(Ntime)]);
end
plotmaxx=max(max(Uh)); plotminx=min(min(Uh));
plotmaxy=max(max(Vh)); plotminy=min(min(Vh));
plotmaxz=max(max(Wh)); plotminz=min(min(Wh));
xlim([plotminx,plotmaxx])
ylim([plotminy,plotmaxy])
zlim([plotminz,plotmaxz])
caxis([minc,maxc]);
current_angle = [(1-1)/(Ntime)*rot*360,0] + angle;
%if no rotation is given, angle of view is fixed
%if rotation = 1, then it rotates as the movie goes on
view(current_angle);
if axismode==false
    axis off;
else
    ax = gca;
    ax.XLabel.String = 'x';
    ax.XLabel.FontSize = 12;
    ax.YLabel.String = 'y';
    ax.YLabel.FontSize = 12;
    ax.ZLabel.String = 'z';
    ax.ZLabel.FontSize = 12;
    ax.ZLabel.Rotation = 0;
end
camlight('headlight');

if iwantmovie
    Myvideo.F(1) =  getframe(gcf);
    writeVideo(writerObj,Myvideo.F(1));
elseif iwantsnapshots
    if exist([pwd '/movies'],'file') ≠ 7
        figurename=['snapshot_' num2str(1) '.png'];
    else
        figurename=['movies/snapshot_' num2str(1) '.png'];
    end
    saveas(gcf,figurename);
```

```
else
    pause(0.1);
end
for j=2:Ntime
    if ¬ishghandle(figH)
        return
    end
    shading interp
    colormap(colorscale);
    currentcolor=C(:,j);
    figH=plotHandle(Uh(:,j), Vh(:,j), Wh(:,j), currentcolor);
    axis equal;
    xlim([plotminx,plotmaxx])
    ylim([plotminy,plotmaxy])
    zlim([plotminz,plotmaxz])
    caxis([minc,maxc]);
    if ¬iwantsnapshots
        title(['time step = ' num2str(j) '/' num2str(Ntime)]);
    end
    current_angle = [(j-1)/(Ntime)*rot*360,0] + angle;
    %if no rotation is given, angle of view is fixed
    %if rotation = 1, then it rotates as the movie goes on
    view(current_angle);
    if axismode==false
        axis off;
    else
        ax = gca;
        ax.XLabel.String = 'x';
        ax.XLabel.FontSize = 12;
        ax.YLabel.String = 'y';
        ax.YLabel.FontSize = 12;
        ax.ZLabel.String = 'z';
        ax.ZLabel.FontSize = 12;
        ax.ZLabel.Rotation = 0;
    end
    camlight('headlight');
    if iwantmovie
        Myvideo.F(j) =  getframe(gcf);
        writeVideo(writerObj,Myvideo.F(j));
    elseif iwantsnapshots
        if exist([pwd '/movies'],'file') ≠ 7
            figurename=['snapshot_' num2str(j) '.png'];
        else
            figurename=['movies/snapshot_' num2str(j) '.png'];
        end
        saveas(gcf,figurename);
    else
        pause(0.1);
    end
end
if iwantmovie
    close(writerObj)
    close
elseif iwantsnapshots
    clc;close all;
    if exist([pwd '/movies'],'file') ≠ 7
        disp('Your snapshots are being saved in:');
        disp(pwd);
    else
        disp('Your snapshots are being saved in:');
        disp([pwd '/movies']);
    end
end


end
```

5.3 Plotting and refining on the boundary of the mesh

Plotting a scalar function. Given a tetrahedral mesh, polynomial degree k and a $\mathcal{P}_k$ FEM solution u_h, the function `FEMsurfaceplot` refines each boundary face of the mesh into k^2 boundary faces and plots the refinement using u_h as the color. To see how this is done, consider the case on the reference element where we create a cardinal grid (the case for $k = 4$ is shown in see Figure 1).

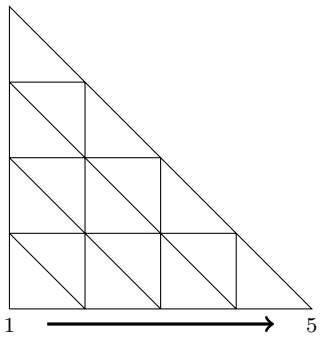

Figure 1: The cardinal grid on the reference triangle for $k = 4$, showing the way in which degrees of freedom and elements are counted. `fig:surfplot`

Still need to describe the process, make a diagram of the cardinal mesh on the reference element including the way degrees of freedom are counted, etc

```
function FEMsurfaceplot(uh,T,k)

% FEMsurfaceplot(uh,T,k)
%
% Input:
%       uh    : scalar FEM solution, used for coloring the plot
%       T     : enhanced tetrahedrization
%       k     : polynomial degree
%
% Output:
%       The result of this function is a plot on a k-refinement of the
%       surface of the mesh contained in T colored by the function uh
%
% Last Modified: December 2, 2017

pt = [];
ind =1;
for i = 0:k
    pt = [pt; ...
          (0:k-i)', i*ones(k-i+1,1)];
    list{i+1} = ind+(0:k-i);
    ind = ind+k+1-i; % index number for beginning of next row
end

pt = pt/k;
pt = [1-pt(:,1)-pt(:,2), pt];

ell = k+1:-1:1; % ell(i) = length(list{i})

Tri =[];
```

```
for i = 1:k
    t = list{i}(1:ell(i)-1); % bottom left vertex
    Tri = [Tri, [t;1+t;ell(i) + t]];
    t(end) = [];
    Tri = [Tri, [t+1; 1+ell(i)+t; ell(i)+t]];
end

LT = T.faces(1:3, T.faces(4,:)~=0);
X = T.coordinates(1,:); X=pt*X(LT);
Y = T.coordinates(2,:); Y=pt*Y(LT);
Z = T.coordinates(3,:); Z=pt*Z(LT);

d2 = nchoosek(k+2,2);
TRI = bsxfun(@plus, d2*(0:size(LT,2)-1), Tri(:));
TRI = reshape(TRI, 3, k^2*size(LT,2));

faceList = find(T.faces(4,:)~=0);
DOF=computeBDDOF3D(T,k,faceList);

P=bernstein2D(pt(:,2),pt(:,3),k);
U=P*uh(DOF);
trisurf(TRI', X(:), Y(:), Z(:),U(:))
xlabel('x');ylabel('y');
shading interp
```

Plotting the displaced mesh. Using the technique that `FEMsurfaceplot.m` uses which is described above, we can plot the displaced mesh with a better resolution. Function `FEMsurfacePlotDisplacement.m` uses the similar input as `plotElasticity3D.m` does and produces the following:

- Plot of the displacement at the equilibrium state
- Animation of the evolution of the displacement
- Some snapshots of the displacement

on the refined mesh. Details of the refinement is explained above. As an example, if you have an FEM solution $\mathbf{u}_h$ using $\mathcal{P}_k$ elements with $k \geq 2$ on a cubic mesh which has a ℓ–level of refinement, this function produces a plot which is displayed on a mesh with $2 \times \ell$–refinement. Here when we say ℓ–level refinement we mean a cubic mesh where each edge is partitioned into ℓ smaller edges.
This function is still in the development process, and at this point it cannot refine the colormap. Therefore the animation and plots have a fixed color per element. We refer to `plotElasticity3D.m` for the details and the usage of the input with an exception of the input σ_h. Since this function assigns single color per element, average stress is not asked as an input, instead a color option should be given as a part of the `movie` input.

A big part of this function is exactly same as `plotElasticity3D.m`. When the color refinement project is finished we will replace `plotElasticity3D.m` with this function. At the moment both of them is being kept in the documentation.

```
function FEMsurfacePlotDisplacement3D(uhtot,T,k,varargin)
% FEMsurfacePlotDisplacement3D(uhtot,T,k)
% FEMsurfacePlotDisplacement3D(uhtot,T,k,{})
```

```
%     OR     FEMsurfacePlotDisplacement3D(uhtot,T,k,movie)
% FEMsurfacePlotDisplacement3D(uhtot,T,k,movie,snapshots)
%
% Input:
%   uhtot    :      3*dimVh x Ntimes complex vector (containing uh,vh,wh)
%   T        :      Data structure. Enhanced FEM triangulation
%   k        :      Scalar. Polynomial degree
%                   is desired pass in a scalar i.e. 1.
%   movie    :      Cell element containing the following
%                   (for default you can pass in {})
%   movie{1}:       Rotation. This can be 1 or 0, enables or disables
%                      camera rotation.
%                      Default: no rotation
%   movie{2}:       Camera angle. 1x2 vector corresponding to
%                      [azimuth elevation] in radians.
%                      Input option: 'default'
%                      Default: [-37.5,30]
%   movie{3}:       Color for the faces in MATLAB RGB vector,
%                      Input options: 'default' OR 'flat',
%                       OR 'grayscale', 'hotscale, ''winterscale',...
%                       OR 'green','red',...
%                       OR [1,0.3,0.8]
%                      'flat': Uniform colors taken from stress data
%                      'grayscale': Computes the grayscale of the
%                                   given coloring in the stress data
%                      'green': Uniform red color ignoring stress
%                      Default: 'flat'
%   movie{4}:       Axis on or off. This can be 1 or 0. When
%                      it is on it shows the labels too.
%                      Default: on
%   movie{5}:       Edge line style.
%                      Input options: '-'(solid line), 'none', '--'(dashed),
%                      ':' (dotted)
%                      Default: '-'
%   movie{6}:       Number of frames per second. A positive integer.
%                      Default: 30
%   snapshots:       Enables to save snaphots.
%                    Input options: 'all',index vector
%                      'all': Saves all time steps of the snaphots of the given
%                      data
%                      index vector: Saves the snapshots at the time steps
%                      corresponding to the index vector
%                    When you save snapshots, they are all saved in the
%                    designated folder with names: snapshot_1,snapshot_2,...
%                    in png format.
% Action: plot of ...
%   displaced mesh with constant coloring
%   displaced mesh with constant coloring and saves the video
%   displaced mesh with constant coloring and saves some snapshots
%
% Last modified: February 2, 2018

%Surface refining
pt = [];
ind =1;
for i = 0:k
    pt = [pt; ...
          (0:k-i)', i*ones(k-i+1,1)];%#ok
    list{i+1} = ind+(0:k-i);%#ok
    ind = ind+k+1-i; % index number for beginning of next row
end

pt = pt/k;
pt = [1-pt(:,1)-pt(:,2), pt];

ell = k+1:-1:1; % ell(i) = length(list{i})
```

```
Tri =[];
for i = 1:k
    t = list{i}(1:ell(i)-1); % bottom left vertex
    Tri = [Tri, [t;1+t;ell(i) + t]];%#ok
    t(end) = [];
    Tri = [Tri, [t+1; 1+ell(i)+t; ell(i)+t]];%#ok
end

LT = T.faces(1:3, T.faces(4,:)~=0);
X = T.coordinates(1,:); X=pt*X(LT);
Y = T.coordinates(2,:); Y=pt*Y(LT);
Z = T.coordinates(3,:); Z=pt*Z(LT);

d2 = nchoosek(k+2,2);
TRI = bsxfun(@plus, d2*(0:size(LT,2)-1), Tri(:));
TRI = reshape(TRI, 3, k^2*size(LT,2));

faceList = find(T.faces(4,:)~=0);
DOF=computeBDDOF3D(T,k,faceList);
Nf=size(faceList,2);
% in case of snapshots
iwantsnapshots=false;
if nargin>=5
    snapshots=varargin{2};
    if isnumeric(snapshots)
        uhtot=uhtot(:,snapshots);
    end
    iwantsnapshots=true;
end
P=bernstein2D(pt(:,2),pt(:,3),k);
len = size(uhtot,1)/3;
uh = uhtot(1:len,:); Ntime = size(uh,2); uh=uh(:);
vh = uhtot(len+1:2*len,:); vh=vh(:);
wh = uhtot(2*len+1:end,:); wh=wh(:);
M=ones(d2,1)*(0:Ntime-1); M=len*M(:);
DOFM=bsxfun(@plus,repmat(DOF,[Ntime 1]),M);
uh=uh(DOFM); vh=vh(DOFM); wh=wh(DOFM);
uh=mat2cell(uh,d2*ones(1,Ntime),Nf); uh=cell2mat(uh');
vh=mat2cell(vh,d2*ones(1,Ntime),Nf); vh=cell2mat(vh');
wh=mat2cell(wh,d2*ones(1,Ntime),Nf); wh=cell2mat(wh');
U=P*uh; U=reshape(U,[d2*Nf,Ntime]);
V=P*vh; V=reshape(V,[d2*Nf,Ntime]);
W=P*wh; W=reshape(W,[d2*Nf,Ntime]);

% Taking the values of uh which contributed by the vertices and adding the
% mesh coordinates
Uh=bsxfun(@plus,U,X(:));
Vh=bsxfun(@plus,V,Y(:));
Wh=bsxfun(@plus,W,Z(:));

% Setting a fix coloring
disp('Uniform coloring is set.');
C = ones(Nf,Ntime);
maxc = max(max(C)) + eps; %adding a small number to avoid minc=maxc case
minc = min(min(C));

% Determining the movie options
iwantmovie=false;
rot=0;
angle=[-37.5,30];
facecolor='flat';
axismode=true;
linestyle='-';
framerate=30;
colorscale=parula;
```

```
if nargin≥4
    movie=varargin{1};
    iwantmovie=true;
    if ¬isempty(movie)
        rot=movie{1};
        if length(movie)>1
            angle=movie{2};
            if strcmp(angle,'default')==1
                angle=[−37.5, 30];
            end
            if length(movie)>2
                facecolor=movie{3};
                if strcmp(facecolor,'default')==1
                    facecolor='flat';
                end
                if ischar(facecolor) && ¬isempty(strfind(facecolor,'scale'))%#ok
                    colorscale=strrep(facecolor,'scale','');
                    facecolor='flat';
                end
                if length(movie)>3
                    axismode=movie{4};
                    if length(movie)>4
                        linestyle=movie{5};
                        if length(movie)>5
                            framerate=movie{6};
                        end
                    end
                end
            end
        end
    end
end
if iwantsnapshots
    iwantmovie=false;
end

%creating the movie in the designated folder
if iwantmovie
    close all;
    DateAndTime=datetime('now');
    TimeStamp=datestr(datenum(DateAndTime),'yy−mm−dd−HH−MM');
    if exist([pwd '/movies'],'file') ≠ 7
        disp('We can not find the /movies folder in current path.');
        FileStr=strcat(TimeStamp,'_FastMovie_k=',...
            num2str(k),'_M=',num2str(Ntime));
        writerObj = VideoWriter(FileStr,'MPEG−4');
        writerObj.Quality = 100;
        writerObj.FrameRate = framerate;
        open(writerObj);
        disp('Therefore, your videos is being saved in:');
        display(pwd);
    else
        FileStr=strcat('movies/',TimeStamp,'_FastMovie_k=',...
            num2str(k),'_M=',num2str(Ntime));
        writerObj = VideoWriter(FileStr,'MPEG−4');
        writerObj.Quality = 100;
        writerObj.FrameRate = framerate;
        open(writerObj);
        disp('Your videos is being saved in:');
        disp([pwd '/movies']);
    end
end

% Animation part
currentcolor=C(:,1);
plotHandle = @(u,v,w,Col) trisurf(TRI', u, v, w, 0.1+0*u,...
```

```
    'LineStyle',linestyle,...
    'FaceLighting','flat',...
    'FaceColor',facecolor,...
    'AmbientStrength',0.3,...
    'DiffuseStrength', 0.8,...
    'SpecularStrength', 0.9,...
    'SpecularExponent', 10);
% Creating a full size figure
figure('units','normalized','outerposition',[0 0 1 1]);
shading interp
colormap(colorscale);
figH=plotHandle(Uh(:,1), Vh(:,1), Wh(:,1), currentcolor);
axis equal;
if ¬iwantsnapshots
    title(['time step = ' num2str(1) '/' num2str(Ntime)]);
end
plotmaxx=max(max(Uh)); plotminx=min(min(Uh));
plotmaxy=max(max(Vh)); plotminy=min(min(Vh));
plotmaxz=max(max(Wh)); plotminz=min(min(Wh));
xlim([plotminx,plotmaxx])
ylim([plotminy,plotmaxy])
zlim([plotminz,plotmaxz])
caxis([minc,maxc]);
current_angle = [(1-1)/(Ntime)*rot*360,0] + angle;
%if no rotation is given, angle of view is fixed
%if rotation = 1, then it rotates as the movie goes on
view(current_angle);
if axismode==false
    axis off;
else
    ax = gca;
    ax.XLabel.String = 'x';
    ax.XLabel.FontSize = 12;
    ax.YLabel.String = 'y';
    ax.YLabel.FontSize = 12;
    ax.ZLabel.String = 'z';
    ax.ZLabel.FontSize = 12;
    ax.ZLabel.Rotation = 0;
end
camlight('headlight');

if iwantmovie
    Myvideo.F(1) =  getframe(gcf);
    writeVideo(writerObj,Myvideo.F(1));
elseif iwantsnapshots
    if exist([pwd '/movies'],'file') ≠ 7
        figurename=['snapshot_' num2str(1) '.png'];
    else
        figurename=['movies/snapshot_' num2str(1) '.png'];
    end
    saveas(gcf,figurename);
else
    pause(0.1);
end
for j=2:Ntime
    if ¬ishghandle(figH)
        break
    end
    shading interp
    colormap(colorscale);
    currentcolor=C(:,j);
    figH=plotHandle(Uh(:,j), Vh(:,j), Wh(:,j), currentcolor);
    axis equal;
    xlim([plotminx,plotmaxx])
    ylim([plotminy,plotmaxy])
    zlim([plotminz,plotmaxz])
```

```
    caxis([minc,maxc]);
    if ¬iwantsnapshots
        title(['time step = ' num2str(j) '/' num2str(Ntime)]);
    end
    current_angle = [(j-1)/(Ntime)*rot*360,0] + angle;
    %if no rotation is given, angle of view is fixed
    %if rotation = 1, then it rotates as the movie goes on
    view(current_angle);
    if axismode==false
        axis off;
    else
        ax = gca;
        ax.XLabel.String = 'x';
        ax.XLabel.FontSize = 12;
        ax.YLabel.String = 'y';
        ax.YLabel.FontSize = 12;
        ax.ZLabel.String = 'z';
        ax.ZLabel.FontSize = 12;
        ax.ZLabel.Rotation = 0;
    end
    camlight('headlight');
    if iwantmovie
        Myvideo.F(j) =  getframe(gcf);
        writeVideo(writerObj,Myvideo.F(j));
    elseif iwantsnapshots
        if exist([pwd '/movies'],'file') ≠ 7
            figurename=['snapshot_' num2str(j) '.png'];
        else
            figurename=['movies/snapshot_' num2str(j) '.png'];
        end
        saveas(gcf,figurename);
    else
        pause(0.1);
    end
end
if iwantmovie
    close(writerObj)
    close
elseif iwantsnapshots
    clc;close all;
    if exist([pwd '/movies'],'file') ≠ 7
        disp('Your snapshots are being saved in:');
        disp(pwd);
    else
        disp('Your snapshots are being saved in:');
        disp([pwd '/movies']);
    end
end


end
```

6 Needs checking

6.1 Piezo elasticity

Suppose now that we want an elasticity problem inside a solid with piezoelectric properties. We build upon the function `FEMelasticity.m` by redefining σ as

$$\sigma = \mu(\nabla \mathbf{u} + \nabla \mathbf{u}^\top) + \lambda \nabla \cdot \mathbf{u} I_3 + \mathbf{e} \nabla \psi,$$

where ψ is a piezoelectric potential and $\mathbf{e}$ is the third order piezoelectric tensor $\mathbf{e} = (e_{kij})_{i,j,k=1}^3$ with the symmetries $e_{kij} = e_{kji}$. We need to include the equation

$$\nabla \cdot (\mathbf{e}^\top 1/2(\nabla \mathbf{u} + \nabla \mathbf{u}^\top) - \kappa \nabla \psi) = 0,$$

where $\kappa = (\kappa_{ij})_{i,j=1}^3$ is the second order dielectric tensor with $\kappa_{ij} = \kappa_{ji}$. Note that this code does not enforce this divergence free condition, but rather given a right-hand side gives the coefficients for $\mathbf{u}^h$ and ψ^h. In this way the code can be used in a more general, if not strictly physical, setting.

Implementation of stiffness matrices. The main difference between the function `FEMpiezoElasticity3D.m` and `FEMelasticity3D.m` is the structure of the matrix. We still have the presence of the discretized versions of the bilinear forms $a(\mathbf{u}, \mathbf{v}^\alpha)_{\alpha \in \{x,y,z\}}$, but now must include discretized versions of the following integrals. Before we write the integrals, to save space we will define

$$\mathbf{e}\nabla\psi = \begin{bmatrix} \sum_k e_{k11}\psi_{x_k} & \sum_k e_{k12}\psi_{x_k} & \sum_k e_{k13}\psi_{x_k} \\ \sum_k e_{k12}\psi_{x_k} & \sum_k e_{k22}\psi_{x_k} & \sum_k e_{k23}\psi_{x_k} \\ \sum_k e_{k13}\psi_{x_k} & \sum_k e_{k23}\psi_{x_k} & \sum_k e_{k33}\psi_{x_k} \end{bmatrix},$$

where $(x_1, x_2, x_3) = (x, y, z)$.

$$\begin{aligned} b(\mathbf{e}\nabla\psi, \mathbf{v}^x) &= \int_\Omega \mathbf{e}\nabla\psi : \begin{bmatrix} v_x & v_y & v_z \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \, \mathrm{d}\mathbf{x} \\ &= \int_\Omega \left(\sum_k e_{k11}\psi_{x_k} \right) v_x + \int_\Omega \left(\sum_k e_{k12}\psi_{x_k} \right) v_y + \int_\Omega \left(\sum_k e_{k13}\psi_{x_k} \right) v_z \end{aligned}$$

$$\begin{aligned} b(\mathbf{e}\nabla\psi, \mathbf{v}^y) &= \int_\Omega \mathbf{e}\nabla\psi : \begin{bmatrix} 0 & 0 & 0 \\ v_x & v_y & v_z \\ 0 & 0 & 0 \end{bmatrix} \, \mathrm{d}\mathbf{x} \\ &= \int_\Omega \left(\sum_k e_{k12}\psi_{x_k} \right) v_x + \int_\Omega \left(\sum_k e_{k22}\psi_{x_k} \right) v_y + \int_\Omega \left(\sum_k e_{k23}\psi_{x_k} \right) v_z \end{aligned}$$

$$\begin{aligned} b(\mathbf{e}\nabla\psi, \mathbf{v}^z) &= \int_\Omega \mathbf{e}\nabla\psi : \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ v_x & v_y & v_z \end{bmatrix} \, \mathrm{d}\mathbf{x} \\ &= \int_\Omega \left(\sum_k e_{k13}\psi_{x_k} \right) v_x + \int_\Omega \left(\sum_k e_{k23}\psi_{x_k} \right) v_y + \int_\Omega \left(\sum_k e_{k33}\psi_{x_k} \right) v_z. \end{aligned}$$

We also need the transpose of this form acting on the displacement vector $\mathbf{u}$, $b^\top(\mathbf{e}^\top\mathbf{u}, \nabla\varphi)$, which is defined as

$$\begin{aligned} &\int_\Omega \begin{bmatrix} e_{111}u_x^x + e_{112}(u_y^x + u_x^y) + e_{113}(u_z^x + u_z^z) + e_{122}u_y^y + e_{123}(u_z^y + u_y^z) + e_{133}u_z^z \\ e_{211}u_x^x + e_{212}(u_y^x + u_x^y) + e_{213}(u_z^x + u_z^z) + e_{222}u_y^y + e_{223}(u_z^y + u_y^z) + e_{233}u_z^z \\ e_{311}u_x^x + e_{312}(u_y^x + u_x^y) + e_{313}(u_z^x + u_z^z) + e_{322}u_y^y + e_{323}(u_z^y + u_y^z) + e_{333}u_z^z \end{bmatrix} \cdot \begin{bmatrix} \varphi_x \\ \varphi_y \\ \varphi_z \end{bmatrix} \, \mathrm{d}\mathbf{x} \\ = &\int_\Omega \sum_{k,j} e_{k1j}u_{x_j}^x \varphi_{x_k} + \int_\Omega \sum_{k,j} e_{k2j}u_{x_j}^y \varphi_{x_k} + \int_\Omega \sum_{k,j} e_{k3j}u_{x_j}^z \varphi_{x_k}, \end{aligned}$$

where we are again using mixed notation $\mathbf{x} = (x, y, z)^\top = (x_1, x_2, x_3)^\top$, and recalling that $e_{kij} = e_{kji}$. The final bilinear form that we need is also the simplest

$$c(\kappa\nabla\psi, \nabla\varphi) = \int_\Omega \left(\begin{bmatrix} \kappa_{11} & \kappa_{12} & \kappa_{13} \\ \kappa_{21} & \kappa_{22} & \kappa_{23} \\ \kappa_{31} & \kappa_{32} & \kappa_{33} \end{bmatrix} \begin{bmatrix} \psi_x \\ \psi_y \\ \psi_z \end{bmatrix} \right) \cdot \begin{bmatrix} \varphi_x \\ \varphi_y \\ \varphi_z \end{bmatrix} \, \mathrm{d}\mathbf{x}$$

which is explained in Section the stiffness matrix of FEM3Ddocumentation.pdf. With all of these defined, we get the following form of the matrix

$$\begin{bmatrix} a(\mathbf{u},\mathbf{v}^x) & b(\nabla\psi,\mathbf{v}^x) \\ a(\mathbf{u},\mathbf{v}^y) & b(\nabla\psi,\mathbf{v}^y) \\ a(\mathbf{u},\mathbf{v}^z) & b(\nabla\psi,\mathbf{v}^z) \\ b^\top(\mathbf{e}^\top\mathbf{u},\nabla\varphi) & c(\kappa\nabla\psi,\nabla\varphi) \end{bmatrix},$$

which is a 4×4 block matrix of stiffness matrices. The challenge with implementation here is that for the $b(\cdot,\cdot)$ forms, we cannot use the function `stiffnessMatrices3D`, because the tensor of functions that we pass in is not symmetric. The reason for this is that we code the piezoelectric tensor as a 3×6 cell array of functions `e{i,j}` where $1\le i\le 3$ and $1\le j\le 6$, where this `i` corresponds to the index k above and the six indices `j` corresponds to the ij indices 11, 12, 13, 22, 23, and 33 respectively, and for each $b(\cdot,\cdot)$ we need 9 different entries instead of just 6. To compute the discretized $b(\cdot,\cdot)$ forms, we use the function `stiffnessMatricesNonSymmetric3D` with the appropriate entries of `e{i,j}`.

Details of the input data. The function `FEMpiezoElasticity3D.m` uses the following data

- `f` is a 1×4 cell array of vectorized functions representing any source terms in the right hand side.
- `uD` is a 1×4 cell array containing vectorized functions which are used by the function to enforce Dirichlet Boundary conditions.
- The cell array `sigma` is exactly as in `FEMelasticity3D.m`.
- `eta` is a 1×3 cell array which contains the vectorized functions needed to implement the Neumann boundary condition
$$\nabla\cdot(\mathbf{e}^\top\varepsilon(\mathbf{u})-\kappa\nabla\psi)\cdot\boldsymbol{\nu}=\eta,\quad \text{on } \Gamma_N.$$

A note on the boundary condition. We would also like to note that for this function the implementation of Dirichlet and Neumann boundary conditions must occur on the same parts of the boundary for both the elastic and piezoelectric quantities. That is for a given section of Γ, if we implement Dirichlet boundary conditions for σ, we must also implement Dirichlet boundary conditions for $\mathbf{e}^\top\varepsilon(\mathbf{u})-\kappa\nabla\psi$. Perhaps this will be remedied in the future.

```
function [u1h,u2h,u3h,psi] = FEMpiezoelasticity3D(mu,lam,rho,e,kap,f,uD,...
                                                  sigmaeta,T,k,varargin)
%
% [u1h,u2h,u3h,psi] = FEMpiezoElasticity3D(mu,lam,rho,e,kap,...
%                       {fx,fy,fz,fpsi},{ux,uy,uz,psiD},...
%                       {sigxx,sigxy,sigxz,sigyy,sigyz,sigzz,etax,etay,...
%                        etaz},T,k)
% [u1h,u2h,u3h,psi] = FEMpiezoElasticity3D(mu,lam,rho,e,kap,...
%                       {fx,fy,fz,fpsi},{ux,uy,uz,psiD},...
%                       {sigxx,sigxy,sigxz,sigyy,sigyz,sigzz,etax,etay,...
%                        etaz},T,k,s)
% [u1h,u2h,u3h,psi] = FEMpiezoElasticity3D(mu,lam,rho,e,kap,...
%                       {fx,fy,fz,fpsi},{ux,uy,uz,psiD},...
%                       {sigxx,sigxy,sigxz,sigyy,sigyz,sigzz,etax,etay,...
%                        etaz},T,k,s,rhsVec)
%
% Input:
% mu,lam           : Vectorized function handles of three variables
% rho              : vectorized function of three variables (density)
% e                : 3 x 6 cell array of vectorized function handles
%                    for piezoelectric tensor
% kap              : 1 X 6 cell array of vectorized function handles
%                    for dielectric tensor
% fx, fy, fz, fpsi : Vectorized functions of three variables (load vector)
% ux, uy, uz, psiD : Vectorized functions of three variables (Dirichlet BC)
```

```
% sigxx,...,etax,..: Nine vectorized functions of three variables, six for
%                     the stress and three for the function used for
%                     Neumann BC in the last equation
% T                 : Data structure. Basic FEM triangulation
% k                 : scalar. Polynomial degree
% s                 : Complex wave number (s=-1i k)
% rhsVec            : 3*dimVh-vector of tests for RHS
%
% Output:
% u1h,u2h,u3h       : Ndof x 1 Vectors. Vh coefficients of the displacement
%                     components.
% psi               : Ndof x 1 vector of piezoelectric coefficients
%
% Last modified July 13, 2016.

T = edgesAndFaces(T);
T = enhanceGrid3D(T);

fh =loadVector3D(f,T,k);
fx = fh(:,1);
fy = fh(:,2);
fz = fh(:,3);
fpsi = fh(:,4);

traction=neumannBC3D({sigmaeta{1:6}},T,k);
tx=traction(:,1);
ty=traction(:,2);
tz=traction(:,3);

ned = neumannBC3D({sigmaeta{7:9}},T,k);

if nargin==12
    rhsVec = varargin{2};
else
    rhsVec = zeros(size([fx;fy;fz;fpsi]));
end

Sm = stiffnessMatrices3D(mu,mu,mu,mu,mu,mu,T,k);
Sl = stiffnessMatrices3D(lam,lam,lam,lam,lam,lam,T,k);

E14 = stiffnessMatricesNonSymmetric3D(e{1,1},e{2,1},e{3,1},e{1,2},...
                                e{2,2},e{3,2},e{1,3},e{2,3},e{3,3},T,k);
E14 = E14{1,1} + E14{1,2} + E14{1,3} + E14{2,1} + E14{2,2} + E14{2,3}...
    + E14{3,1} + E14{3,2} + E14{3,3};
E24 = stiffnessMatricesNonSymmetric3D(e{1,2},e{2,2},e{3,2},e{1,4},...
                                e{2,4},e{3,4},e{1,5},e{2,5},e{3,5},T,k);
E24 = E24{1,1} + E24{1,2} + E24{1,3} + E24{2,1} + E24{2,2} + E24{2,3}...
    + E24{3,1} + E24{3,2} + E24{3,3};
E34 = stiffnessMatricesNonSymmetric3D(e{1,3},e{2,3},e{3,3},e{1,5},...
                                e{2,5},e{3,5},e{1,6},e{2,6},e{3,6},T,k);
E34 = E34{1,1} + E34{1,2} + E34{1,3} + E34{2,1} + E34{2,2} + E34{2,3}...
    + E34{3,1} + E34{3,2} + E34{3,3};

Kap = stiffnessMatrices3D(kap{1},kap{2},kap{3},kap{4},kap{5},kap{6}, T,k);
Kap = Kap{1,1} + Kap{1,2} + Kap{1,2}' + Kap{1,3} + Kap{1,3}' + Kap{2,2}...
    + Kap{2,3} + Kap{2,3}' + Kap{3,3};

% MAT = [(C \varepsilon(w), \varepsilon(w) (e \grad \phi, \grad \phi);...
%       -(e^T \varepsilon(w), \grad \phi)  (\kappa \grad \phi, \grad \phi)]

MAT=[2*Sm{1,1}+Sl{1,1}+Sm{2,2}+Sm{3,3},Sl{1,2}+Sm{1,2}',...
     Sl{1,3}+Sm{1,3}',                  E14;...
     Sm{1,2}+Sl{1,2}',                  2*Sm{2,2}+Sm{1,1}+Sl{2,2}+Sm{3,3}...
     Sl{2,3}+Sm{2,3}',                  E24;...
     Sl{1,3}'+Sm{1,3},                  Sl{2,3}'+Sm{2,3},...
     2*Sm{3,3}+Sm{1,1}+Sm{2,2}+Sl{3,3},E34;...
```

```
    -E14',                          -E24',...
    -E34',                          Kap];

if nargin > 10
    s = varargin{1};
    Mh = massMatrix3D(rho,T,k);
    O=sparse(size(Mh,1),size(Mh,2));
    MAT=MAT+s^2*[Mh O O O;...
                 O Mh O O;...
                 O O Mh O;...
                 O O O  O];
end

[uh,~,free] = dirichletBC3D(uD,T,k);
uh=uh(:);
RHS = [fx+tx; fy+ty;fz+tz; fpsi - ned]+rhsVec-MAT*uh;

Ndof = size(fx,1);
free = [free(:); Ndof+free(:);2*Ndof+free(:);3*Ndof + free(:)];
uh(free) = MAT(free,free)\RHS(free);
u1h = reshape(uh,[Ndof,4]);
u2h = u1h(:,2);
u3h = u1h(:,3);
psi = u1h(:,4);
u1h(:,2:4) = [];

end
```

7 Appendix

7.1 Mittag-Leffler function

sec:7.1

We use the function `mlf.m` designed by Igor Podlubny and Martin Kacenak. It approximates the Mittag-Leffler function

$$E_{\alpha,\beta}(z) = \sum_{k=0}^{\infty} \frac{z^k}{\Gamma(\alpha k + \beta)}$$

with an accuracy of 10^{-p}.

References

HaSa2016

[1] M. Hassell and F.-J. Sayas. *Convolution quadrature for wave simulations*, volume 9 of *SEMA SIMAI Springer Ser.* Springer, [Cham], 2016.