pyOptFEM Documentation

Release 0.0.7

F. Cuvelier

November 11, 2013

pyOptFEM is a Python module which aims at measuring and comparing the performance of three programming techniques for **assembling finite element matrices in 2D and 3D**. For each of the matrices studied, three assembly versions are available: `base`, `OptV1` and `OptV2`

Three matrices are currently implemented (more details are given in *FEM2D module* and *FEM3D module*) :

- **Mass matrix** :

$$\mathbb{M}_{i,j} = \int_{\Omega_h} \varphi_j(q)\ \varphi_i(q) dq,\ \ \forall (i,j) \in \{1, ..., \mathrm{n_q}\}^2.$$

- **Stiffness matrix** :

$$\mathbb{S}_{i,j} = \int_{\Omega_h} \nabla\varphi_j(q).\nabla\varphi_i(q) dq,\ \ \forall (i,j) \in \{1, ..., \mathrm{n_q}\}^2.$$

- **Elastic stiffness matrix** :

$$\mathbb{K}_{m,l} = \int_{\Omega_h} \underline{\boldsymbol{\epsilon}}^t(\psi_l(q))\underline{\boldsymbol{\sigma}}(\psi_m(q)) dq,\ \ \forall (m,l) \in \{1, ..., d\mathrm{n_q}\}^2.$$

  where d is the space dimension.

Contents:

CHAPTER

ONE

PRESENTATION

Let Ω_h be a triangular (2d) or tetrahedral (3d) mesh of $\Omega \subset \mathbb{R}^d$ corresponding to the following data structure:

name	type	dimension	description
$\mathrm{n_q}$	integer	1	number of vertices
$\mathrm{n_{me}}$	integer	1	number of elements
q	double	$d \times \mathrm{n_q}$	array of vertices coordinates. $\mathrm{q}(\nu, j)$ is the ν-th coordinate of the j-th vertex, $\nu \in \{1, \ldots, d\}$, $j \in \{1, \ldots, \mathrm{n_q}\}$. The j-th vertex will be also denoted by $\mathrm{q^j}$
me	integer	$(d+1) \times \mathrm{n_{me}}$	connectivity array. $\mathrm{me}(\beta, k)$ is the storage index of the β-th vertex of the k-th element, in the array q, for $\beta \in \{1, \ldots, (d+1)\}$ and $k \in \{1, \ldots, \mathrm{n_{me}}\}$
volumes	double	$1 \times \mathrm{n_{me}}$	array of volumes in 3d or areas in 2d. $\mathrm{volumes}(k)$ is the k-th element volume (3d) or element area (2d), $k \in \{1, \ldots, \mathrm{n_{me}}\}$

- **Scalar finite elements :**

  Let $H_h^1(\Omega_h) = \{v \in \mathcal{C}^0(\overline{\Omega_h}),\ \ v_{|T} \in \mathcal{P}_1(T),\ \forall T \in \mathcal{T}_h\}$, be the finite dimensional space spanned by the $\mathcal{P}_1$-Lagrange (scalar) basis functions $\{\varphi_i\}_{i \in \{1,\ldots,\mathrm{n_q}\}}$ where $\mathcal{P}_1(T)$ denotes the space of all polynomials defined over T of total degree less than or equal to 1. The functions φ_i satisfy

  $$\varphi_i(\mathrm{q^j}) = \delta_{\mathrm{i,j}},\ \ \forall (\mathrm{i,j}) \in \{1, \ldots, \mathrm{n_q}\}^2$$

  Then we have $\varphi_i \equiv 0$ on T_k, $\forall k \in \{1, \ldots, \mathrm{n_{me}}\}$ such that $\mathrm{q}^i \notin T_k$.

  A $\mathcal{P}_1$-Lagrange (scalar) finite element matrix is of the generic form

  $$\mathbb{D}_{i,j} = \int_{\Omega_h} \mathcal{D}(\varphi_j, \varphi_i) d\Omega_h,\ \ \forall (i,j) \in \{1, \ldots, \mathrm{n_q}\}^2$$

  where $\mathcal{D}$ is the bilinear differential operator (order 1) defined for all $u, v \in H_h^1(\Omega_h)$ by

  $$\mathcal{D}(u,v) = \langle \mathbb{A} \nabla u, \nabla v \rangle - (u \langle \boldsymbol{b}, \nabla v \rangle - v \langle \nabla u, \boldsymbol{c} \rangle) + duv$$

  and $\mathbb{A} \in (L^\infty(\Omega_h))^{d \times d}$, $\boldsymbol{b} \in (L^\infty(\Omega_h))^d$, $\boldsymbol{c} \in (L^\infty(\Omega_h))^d$ and $d \in L^\infty(\Omega_h)$ are given functions on Ω_h.

  We can notice that $\mathbb{D}$ is a **sparse** matrix due to support properties of φ_i functions.

  Let $\mathrm{n_{dfe}} = d + 1$. The element matrix $\mathbb{D}_{\alpha,\beta}(T_k)$, associated to $\mathbb{D}$, is the $\mathrm{n_{dfe}} \times \mathrm{n_{dfe}}$ matrix defined by

  $$\mathbb{D}_{\alpha,\beta}(T_k) = \int_{T_k} \mathcal{D}(\varphi_\beta^k, \varphi_\alpha^k)(\mathrm{q}) d\mathrm{q},\ \ \forall (\alpha, \beta) \in \{1, \ldots, \mathrm{n_{dfe}}\}^2$$

where $\varphi_\alpha^k = \varphi_{\mathrm{me}(\alpha,k)}$ is the α-th local basis function associated to the k-th element.

For examples,

– the **Mass** matrix is defined by

$$\mathbb{M}_{i,j} = \int_{\Omega_h} \varphi_j \varphi_i d\Omega_h$$

The corresponding bilinear differential operator $\mathcal{D}$ is completely defined with $\mathbb{A} = 0$, $\boldsymbol{b} = \boldsymbol{c} = 0$ and $d = 1$.

– the **Stiffness** matrix is defined by

$$\mathbb{S}_{i,j} = \int_{\Omega_h} \langle \nabla \varphi_j, \nabla \varphi_i \rangle \, d\Omega_h$$

The corresponding bilinear differential operator $\mathcal{D}$ is completely defined with $\mathbb{A} = \mathbb{I}$, $\boldsymbol{b} = \boldsymbol{c} = 0$ and $d = 0$.

- **Vector finite elements :**

The dimension of the space $(H_h^1(\Omega_h))^d$ is $\mathrm{n_{df}} = d\,\mathrm{n_q}$.

– in dimension $d = 2$, we have two natural basis :

the global *alternate* basis $\mathcal{B}_a$ defined by

$$\mathcal{B}_a = \{\boldsymbol{\psi}_1, \ldots, \boldsymbol{\psi}_{2\mathrm{n_q}}\} = \left\{ \begin{pmatrix} \varphi_1 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ \varphi_1 \end{pmatrix}, \begin{pmatrix} \varphi_2 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ \varphi_2 \end{pmatrix}, \ldots, \begin{pmatrix} \varphi_{\mathrm{n_q}} \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ \varphi_{\mathrm{n_q}} \end{pmatrix} \right\}$$

and the global *block* basis $\mathcal{B}_b$ defined by

$$\mathcal{B}_b = \{\boldsymbol{\phi}_1, \ldots, \boldsymbol{\phi}_{2\mathrm{n_q}}\} = \left\{ \begin{pmatrix} \varphi_1 \\ 0 \end{pmatrix}, \begin{pmatrix} \varphi_2 \\ 0 \end{pmatrix}, \ldots, \begin{pmatrix} \varphi_{\mathrm{n_q}} \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ \varphi_1 \end{pmatrix}, \begin{pmatrix} 0 \\ \varphi_2 \end{pmatrix}, \ldots, \begin{pmatrix} 0 \\ \varphi_{\mathrm{n_q}} \end{pmatrix} \right\}.$$

– in dimension $d = 3$, we have two natural basis :

the global *alternate* basis $\mathcal{B}_a$ defined by

$$\mathcal{B}_a = \{\boldsymbol{\psi}_1, \ldots, \boldsymbol{\psi}_{3\mathrm{n_q}}\} = \left\{ \begin{pmatrix} \varphi_1 \\ 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ \varphi_1 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 0 \\ \varphi_1 \end{pmatrix}, \ldots, \begin{pmatrix} \varphi_{\mathrm{n_q}} \\ 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ \varphi_{\mathrm{n_q}} \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 0 \\ \varphi_{\mathrm{n_q}} \end{pmatrix} \right\}$$

and the global *block* basis $\mathcal{B}_b$ defined by

$$\mathcal{B}_b = \{\boldsymbol{\phi}_1, \ldots, \boldsymbol{\phi}_{3\mathrm{n_q}}\} = \left\{ \begin{pmatrix} \varphi_1 \\ 0 \\ 0 \end{pmatrix}, \ldots, \begin{pmatrix} \varphi_{\mathrm{n_q}} \\ 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ \varphi_1 \\ 0 \end{pmatrix}, \ldots, \begin{pmatrix} 0 \\ \varphi_{\mathrm{n_q}} \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 0 \\ \varphi_1 \end{pmatrix}, \ldots, \begin{pmatrix} 0 \\ 0 \\ \varphi_{\mathrm{n_q}} \end{pmatrix} \right\}.$$

A $\mathcal{P}_1$-Lagrange (vector) finite element matrix in $\mathcal{B}_a$ basis is of the generic form

$$\mathbb{H}_{l,m} = \int_{\Omega_h} \mathcal{H}(\boldsymbol{\psi}_m, \boldsymbol{\psi}_l) d\Omega_h, \ \ \forall (l,m) \in \{1, \ldots, d\mathrm{n_q}\}^2$$

where $\mathcal{H}$ is the bilinear differential operator (order 1) defined by

$$\mathcal{H}(\boldsymbol{u}, \boldsymbol{v}) = \sum_{\alpha,\beta=1}^{d} \mathcal{D}^{\alpha,\beta}(u_\alpha, v_\beta)$$

and $\mathcal{D}^{\alpha,\beta}$ is a bilinear differential operator of scalar type.

For example, in dimension $d = 2$, the **Elastic Stiffness** matrix is defined by

$$\mathbb{K}_{m,l} = \int_{\Omega_h} \underline{\boldsymbol{\epsilon}}^t(\psi_l(q)) \mathbb{C} \underline{\boldsymbol{\epsilon}}(\psi_m(q)), \ \ \forall (m,l) \in \{1, ..., 2\mathrm{n_q}\}^2.$$

where $\underline{\boldsymbol{\epsilon}} = (\epsilon_{xx}, \epsilon_{yy}, \epsilon_{xy})^t$ is the strain tensor respectively and $\mathbb{C}$ is the Hooke matrix

$$\mathbb{C} = \begin{pmatrix} \lambda + 2\mu & \lambda & 0 \\ \lambda & \lambda + 2\mu & 0 \\ 0 & 0 & \mu \end{pmatrix}$$

Then the bilinear differential operator associated to this matrix is given by

$$\begin{aligned} \mathcal{H}(\boldsymbol{u}, \boldsymbol{v}) \ &= \ \left\langle \begin{pmatrix} \lambda + 2\mu & 0 \\ 0 & \mu \end{pmatrix} \nabla u_1, \nabla v_1 \right\rangle + \left\langle \begin{pmatrix} 0 & \mu \\ \lambda & 0 \end{pmatrix} \nabla u_2, \nabla v_1 \right\rangle \\ &+ \ \left\langle \begin{pmatrix} 0 & \lambda \\ \mu & 0 \end{pmatrix} \nabla u_1, \nabla v_2 \right\rangle + \left\langle \begin{pmatrix} \mu & 0 \\ 0 & \lambda + 2\mu \end{pmatrix} \nabla u_2, \nabla v_2 \right\rangle \end{aligned}$$

We present now three algorithms (`base`, `OptV1` and `OptV2` versions) for assembling this kind of matrix.

Note: These algorithms can be successfully implemented in various interpreted languages under some assumptions. For all versions, it must have a sparse matrix implementation. For `OptV1` and `OptV2` versions, we also need a particular sparse matrix constructor (see *Sparse matrix requirement*). And, finally, `OptV2` also required that the interpreted languages support classical **vectorization** operations. Here is the current list of interpreted languages for which we have successfully implemented these three algorithms :

- Python with *Numpy* and *Scipy* modules,
- Matlab,
- Octave,
- Scilab

1.1 Classical assembly algorithm (`base` version)

Due to support properties of P_1-Lagrange basis functions, we have the classical algorithm :

Note: We recall the classical matrix assembly in dimension d with $\mathrm{n_{dfe}} = d + 1$.

1: $\mathbb{M} \leftarrow 0$ ▷ Sparse matrix $\mathrm{n_q} \times \mathrm{n_q}$
2: **for** $k \leftarrow 1$ **to** $\mathrm{n_{me}}$ **do** ▷ Loop over mesh elements
3: $\quad \mathbb{E} \leftarrow \texttt{ElemMat}(\mathrm{areas}(k), \ldots)$ ▷ Compute element matrix $n_e \times n_e$
4: $\quad$ **for** $\alpha \leftarrow 1$ **to** n_e **do**
5: $\quad\quad i \leftarrow \mathrm{me}(\alpha, k)$
6: $\quad\quad$ **for** $\beta \leftarrow 1$ **to** $d + 1$ **do**
7: $\quad\quad\quad j \leftarrow \mathrm{me}(\beta, k)$
8: $\quad\quad\quad \mathbb{M}_{i,j} \leftarrow \mathbb{M}_{i,j} + \mathbb{E}_{\alpha,\beta}$
9: $\quad\quad$ **end for**
10: $\quad$ **end for**
11: **end for**

Figure 1.1: Classical matrix assembly in 2d or 3d

In fact, for each element we add its element matrix to the global sparse matrix (lines 4 to 10 of the previous algorithm). This operation is illustrated in the following figure in 2d scalar fields case :

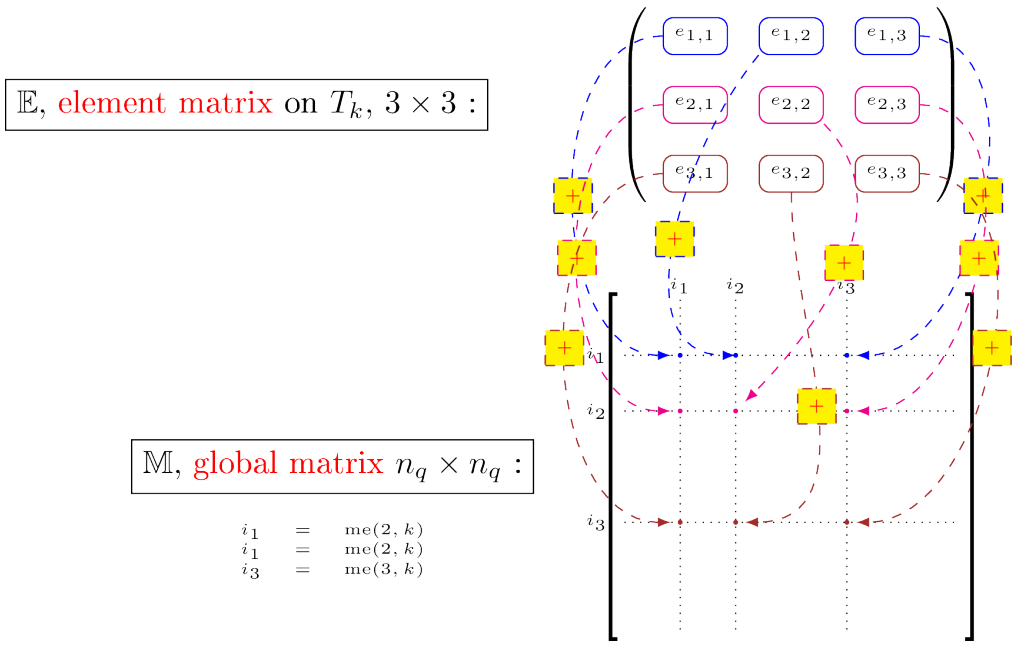


Figure 1.2: Adding of an element matrix to global matrix in 2d scalar fields case

The repetition of elements insertion in **sparse** matrix is very expensive.

1.2 Sparse matrix requirement

The interpreted language must contain a function to generate a sparse matrix `M` from three 1d arrays of same length `Ig`, `Jg` and `Kg` such that `M(Ig(k),Jg(k))=Kg(k)` . Furthermore, the elements of `Kg` having the same indices in `Ig` and `Jg` must be summed.

We give for several interpreted languages the corresponding function :

- Python (*scipy.sparse* module) : `M=sparse.<format>_matrix(Kg,(Ig,Jg),shape=(m,n)` where `<format>` is the sparse matrix format chosen in `csc` , `csr` , `lil` ,...
- Matlab : `M=sparse(Ig,Jg,Kg,m,n)` , only `csc` format.
- Octave : `M=sparse(Ig,Jg,Kg,m,n)` , only `csc` format.
- Scilab : `M=sparse([Ig,Jg],Kg,[m,n])` , only `row-by-row` format.

Obviously, this kind of function exists in compiled languages. For example, in C language, one can use the *SuiteSparse* from T. Davis and with Nvidia GPU, one can use *Thrust* library.

1.3 Optimized classical assembly algorithm (`OptV1` version)

The idea is to create three global 1d-arrays $\boldsymbol{I}_g$, $\boldsymbol{J}_g$ and $\boldsymbol{K}_g$ allowing the storage of the element matrices as well as the position of their elements in the global matrix. The length of each array is $\mathrm{n}_{\mathrm{dfe}}^2\mathrm{n}_{\mathrm{me}}$. (i.e. $9\mathrm{n}_{\mathrm{me}}$ for $d = 2$ and $16\mathrm{n}_{\mathrm{me}}$ for $d = 3$). Once these arrays are created, the matrix assembly is obtained with one of the previous commands.

To create these three arrays, we first define three local 1d-arrays $\boldsymbol{K}_k^e$, $\boldsymbol{I}_k^e$ and $\boldsymbol{J}_k^e$ of $\mathrm{n}_{\mathrm{dfe}}^2$ elements obtained from a generic element matrix $\mathbb{E}(T_k)$ of dimension $\mathrm{n}_{\mathrm{dfe}}$:

$$\begin{array}{lcl} \boldsymbol{K}_k^e & : & \text{elements of the matrix } \mathbb{E}(T_k) \text{ stored column-wise,} \\ \boldsymbol{I}_k^e & : & \text{global row indices associated to the elements stored in } \boldsymbol{K}_k^e, \\ \boldsymbol{J}_k^e & : & \text{global column indices associated to the elements stored in } \boldsymbol{K}_k^e. \end{array}$$

From these arrays, it is then possible to build the three global arrays $\boldsymbol{I}_g$, $\boldsymbol{J}_g$ and $\boldsymbol{K}_g$, of size $\mathrm{n}_{\mathrm{dfe}}^2\mathrm{n}_{\mathrm{me}} \times 1$ defined, $\forall k \in \{1,\ldots,\mathrm{n}_{\mathrm{me}}\}$, $\forall l \in \left\{1,\ldots,\mathrm{n}_{\mathrm{dfe}}^2\right\}$, by

$$\begin{array}{rcl} \boldsymbol{K}(\mathrm{n}_{\mathrm{dfe}}^2(k-1)+l) & = & \boldsymbol{K}_k^e(l), \\ \boldsymbol{I}(\mathrm{n}_{\mathrm{dfe}}^2(k-1)+l) & = & \boldsymbol{I}_k^e(l) = \mathcal{I}^k((l-1) \bmod m + 1), \\ \boldsymbol{J}(\mathrm{n}_{\mathrm{dfe}}^2(k-1)+l) & = & \boldsymbol{J}_k^e(l) = \mathcal{I}^k(E((l-1)/m) + 1). \end{array}$$

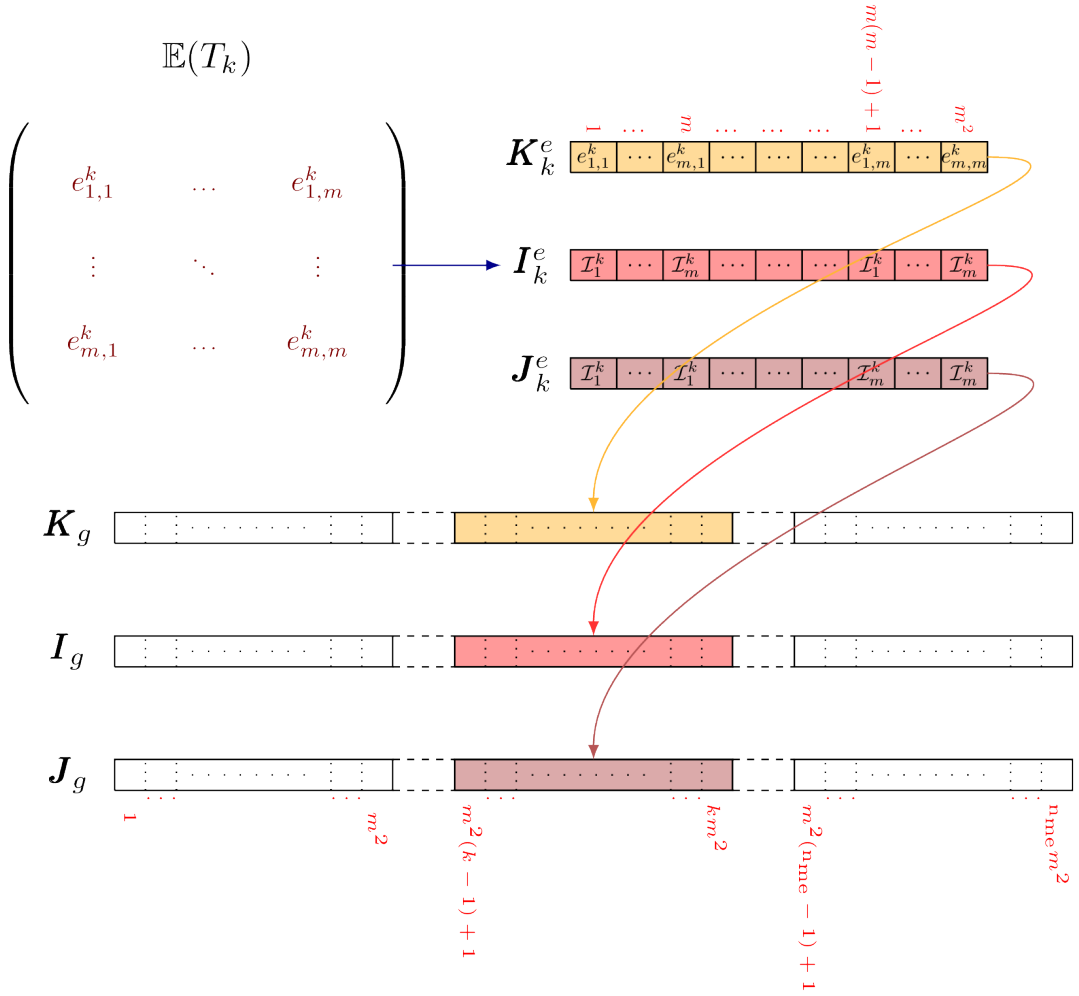
So, for each triangle T_k, we have

Then, a simple algorithm can build these three arrays using a loop over each triangle.

Note: We give the complete `OptV1` algorithm

1.4 New Optimized assembly algorithm (`OptV2` version)

We present the optimized version 2 algorithm where no loop is used. We define three 2d-arrays that allow to store all the element matrices as well as their positions in the global matrix. We denote by $\mathbb{K}_g$, $\mathbb{I}_g$ and $\mathbb{J}_g$ these $\mathrm{n}_{\mathrm{dfe}}^2 - by - \mathrm{n}_{\mathrm{me}}$



Algorithm 1 Optimized matrix assembly - version 1

$dim \leftarrow {\mathrm{n_{dfe}}}^2 \mathrm{n_{me}}$
$\boldsymbol{I}_g \leftarrow \mathbb{O}_{dim \times 1}$ ▷ 1d array of size dim
$\boldsymbol{J}_g \leftarrow \mathbb{O}_{dim \times 1}$ ▷ 1d array of size dim
$\boldsymbol{K}_g \leftarrow \mathbb{O}_{dim \times 1}$ ▷ 1d array of size dim
$ii \leftarrow [1 : \mathrm{n_{dfe}}]^t * \mathbb{1}_{1 \times \mathrm{n_{dfe}}}; ii \leftarrow ii(:);$
$jj \leftarrow \mathbb{1}_{\mathrm{n_{dfe}} \times 1} * [1 : ndfe]; jj \leftarrow jj(:)$
$kk \leftarrow [1 : {\mathrm{n_{dfe}}}^2]$
for $k \leftarrow 1$ **to** $\mathrm{n_{me}}$ **do**
  $\mathbb{E} \leftarrow$ `ElemMat`$(\mathrm{areas}(k), \ldots)$
  $\boldsymbol{I}_g(kk) \leftarrow \mathcal{I}^k(ii)$
  $\boldsymbol{J}_g(kk) \leftarrow \mathcal{I}^k(jj)$
  $\boldsymbol{K}_g(kk) \leftarrow \mathbb{E}(:)$
  $kk \leftarrow kk + {\mathrm{n_{dfe}}}^2$
end for
$\mathbb{M} \leftarrow$`sparse`$(\boldsymbol{I}_g, \boldsymbol{J}_g, \boldsymbol{K}_g, \mathrm{n_q}, \mathrm{n_q})$

arrays, defined $\forall k \in \{1, \dots, \mathrm{n_{me}}\}$, $\forall il \in \{1, \dots, \mathrm{n}_{\mathrm{dfe}}^2\}$ by

$$\mathbb{K}_g(il, k) = \boldsymbol{K}_k^e(il), \qquad \mathbb{I}_g(il, k) = \boldsymbol{I}_k^e(il), \qquad \mathbb{J}_g(il, k) = \boldsymbol{J}_k^e(il). \tag{1.1}$$

The three local arrays $\boldsymbol{K}_k^e$, $\boldsymbol{I}_k^e$ and $\boldsymbol{J}_k^e$ are thus stored in the k^{th} column of the global arrays $\mathbb{K}_g$, $\mathbb{I}_g$ and $\mathbb{J}_g$ respectively.

A natural way to build these three arrays consists in using a loop through the triangles T_k in which we insert the local arrays column-wise

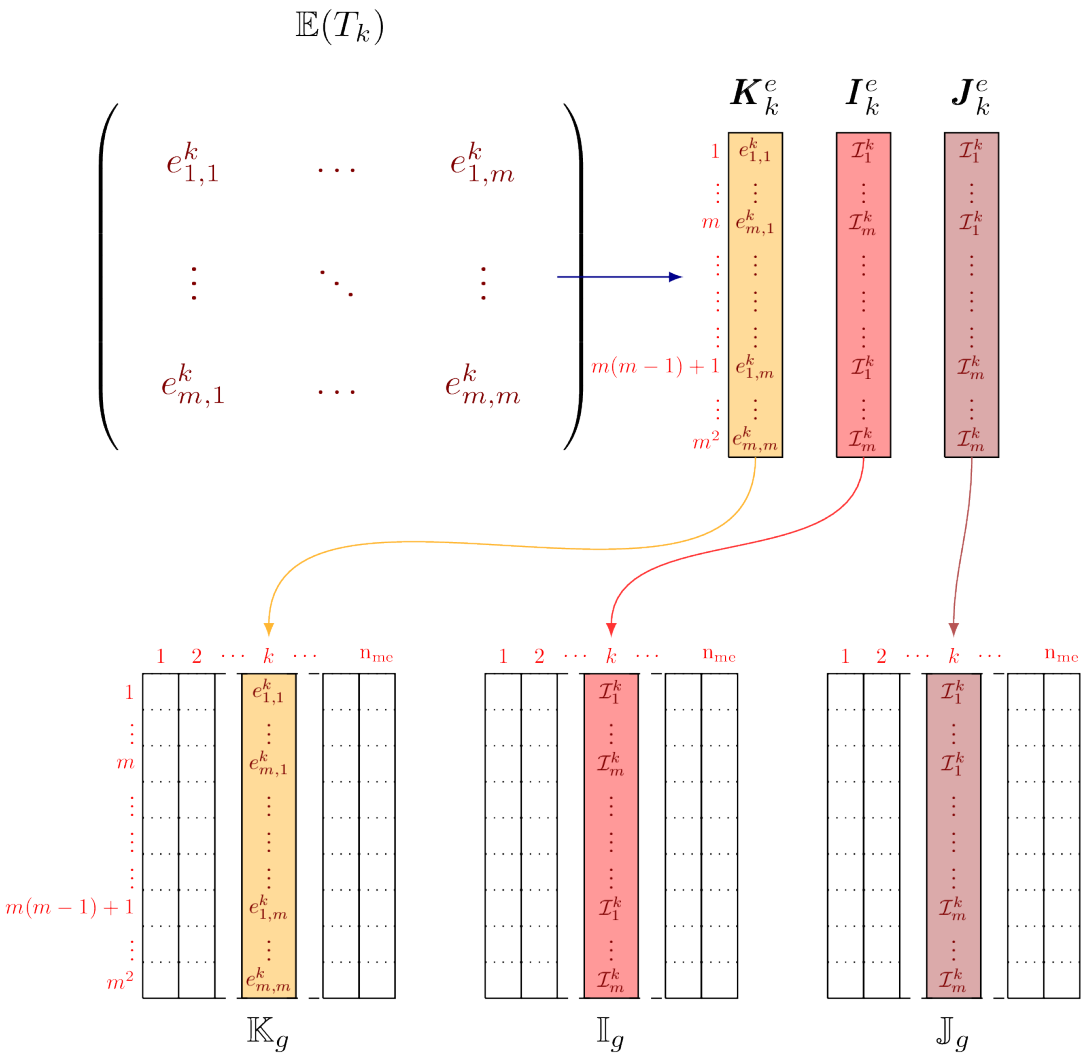


Figure 1.3: Construction of $\mathbb{K}_g$, $\mathbb{I}_g$ and $\mathbb{J}_g$ with loop

The natural construction of these three arrays is done column-wise. So, for each array there are $\mathrm{n_{me}}$ columns to compute, which depends on the mesh size. To vectorize, we must fill these arrays by row-wise operations and then for each array there are $\mathrm{n}_{\mathrm{dfe}}^2$ rows to compute. We recall that $\mathrm{n_{dfe}}$ does not depend on the mesh size. These rows insertions are represented in Figure 1.4 . We can also remark that, $\forall(\alpha, \beta) \in \{1, \dots, \mathrm{n_{dfe}}\}$, with $m = \mathrm{n_{dfe}}$,

$$\mathbb{K}_g(m(\beta-1)+\alpha, k) = e_{\alpha,\beta}^k, \quad \mathbb{I}_g(m(\beta-1)+\alpha, k) = \mathcal{I}_\alpha^k, \quad \mathbb{J}_g(m(\beta-1)+\alpha, k) = \mathcal{I}_\beta^k.$$

where, in scalar fields case, $\mathcal{I}_{\alpha}^{k} = \mathrm{me}(\alpha, k)$, $\forall \alpha \in \{1, \ldots, d+1\}$ and in vector fields case, $\mathcal{I}_{da-b}^{k} = d\,\mathrm{me}(a, k) - b$, $\forall a \in \{1, \ldots, d+1\}$, $\forall b \in \{0, \ldots, d-1\}$.

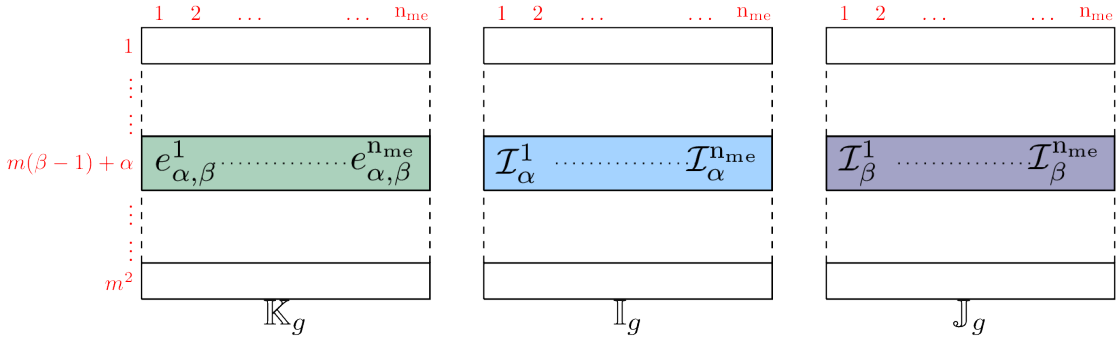


Figure 1.4: Row-wise operations on global arrays $\mathbb{I}_g$, $\mathbb{J}_g$ and $\mathbb{K}_g$

As we can see in Figures 1.5 and 1.6, it is quite easy to vectorize $\mathbb{I}_g$ and $\mathbb{J}_g$ computations in scalar fields case by filling these arrays lines by lines :

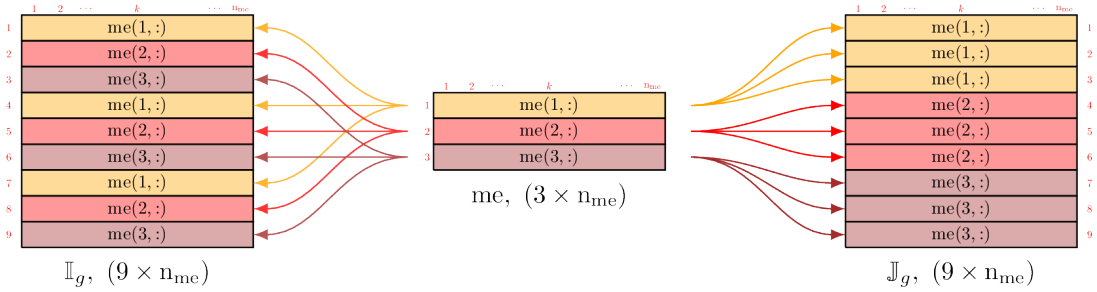


Figure 1.5: Construction of $\mathbb{I}_g$ and $\mathbb{J}_g$: `OptV2` version for 2d *scalar* matrix

Using vectorization tools, we can compute $\mathbb{I}_g$ and $\mathbb{J}_g$ in one line. The vectorized algorithms in 2d and 3d scalar fields are represented by Figure 1.7.

In vector fields case, we construct the tabular $\mathbb{T}$ such that $\mathbb{T}(\alpha, k) = \mathcal{I}_{\alpha}^{k}$, $\forall \alpha \in \{1, \ldots, \mathrm{n_{dfe}}\}$. Then we can vectorize $\mathbb{I}_g$ and $\mathbb{J}_g$ computations in vector fields case. We represent in Figure 1.8 these operations in 2d.

Now, it remains to vectorize the computation of $\mathbb{K}_g$ array which contains all the element matrices associated to $\mathbb{D}$ or $\mathbb{H}$: it should be done by **row-wise** vector operations.

We now describe a vectorized construction of $\mathbb{K}_g$ array associated to a generic bilinear form a. For the mesh element T_k, the element matrix is then given by

$$\mathbb{E}(T_k) = \begin{pmatrix} a(\varphi_{i_1^k}, \varphi_{i_1^k}) & a(\varphi_{i_1^k}, \varphi_{i_2^k}) & a(\varphi_{i_1^k}, \varphi_{i_3^k}) \\ a(\varphi_{i_2^k}, \varphi_{i_1^k}) & a(\varphi_{i_2^k}, \varphi_{i_2^k}) & a(\varphi_{i_2^k}, \varphi_{i_3^k}) \\ a(\varphi_{i_3^k}, \varphi_{i_1^k}) & a(\varphi_{i_3^k}, \varphi_{i_2^k}) & a(\varphi_{i_3^k}, \varphi_{i_3^k}) \end{pmatrix}$$

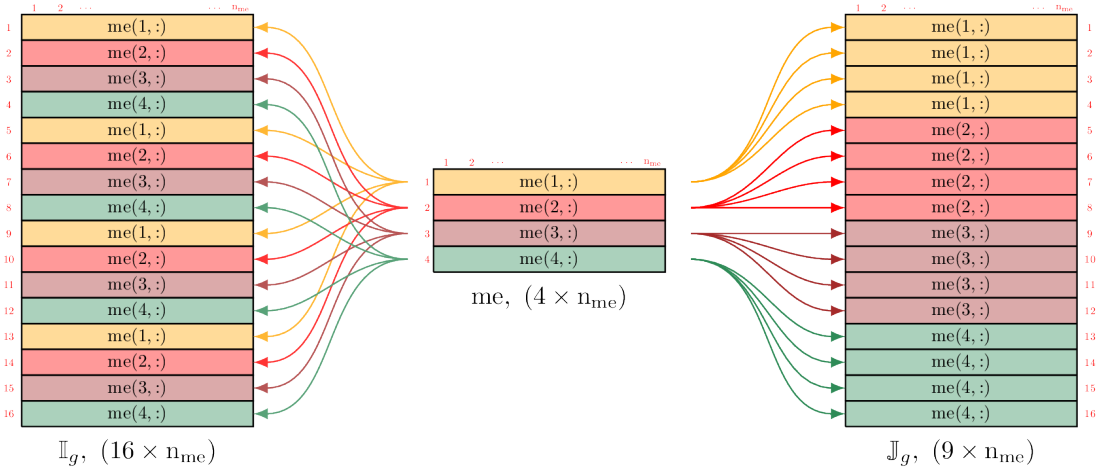


Figure 1.6: Construction of $\mathbb{I}_g$ and $\mathbb{J}_g$: `OptV2` version for 3d *scalar* matrix

Algorithm 1 $\mathbb{I}_g$ and $\mathbb{J}_g$: 2d scalar fields case

Function: $[\mathbb{I}_g, \mathbb{J}_g] \leftarrow$BUILDIGJG2D(me)
$ii \leftarrow [1, 2, 3, 1, 2, 3, 1, 2, 3]$
$jj \leftarrow [1, 1, 1, 2, 2, 2, 3, 3, 3]$
$\mathbb{I}_g \leftarrow \text{me}(ii, :)$ ▷ $9 \times \text{n}_{\text{me}}$ array
$\mathbb{J}_g \leftarrow \text{me}(jj, :)$ ▷ $9 \times \text{n}_{\text{me}}$ array
end Function:

Algorithm 2 $\mathbb{I}_g$ and $\mathbb{J}_g$: 3d scalar fields case

Function: $[\mathbb{I}_g, \mathbb{J}_g] \leftarrow$BUILDIGJG3D(me)
$ii \leftarrow [1:4]^t * \mathbb{1}_{1\times 4}; ii \leftarrow ii(:);$
$jj \leftarrow \mathbb{1}_{4\times 1} * [1:4]; jj \leftarrow jj(:)$
$\mathbb{I}_g \leftarrow \text{me}(ii, :)$ ▷ $16 \times \text{n}_{\text{me}}$ array
$\mathbb{J}_g \leftarrow \text{me}(jj, :)$ ▷ $16 \times \text{n}_{\text{me}}$ array
end Function:

Figure 1.7: Vectorized algorithms for $\mathbb{I}_g$ and $\mathbb{J}_g$ computations

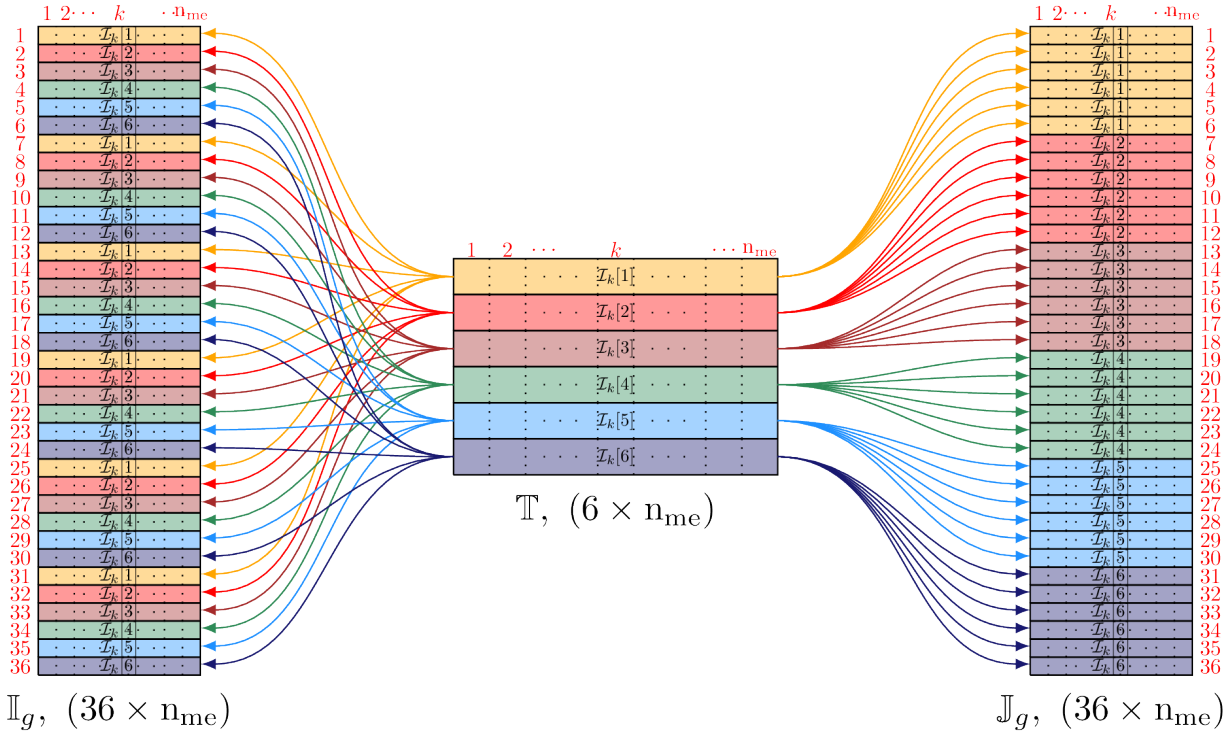


Figure 1.8: Construction of $\mathbb{I}_g$ and $\mathbb{J}_g$: `OptV2` version in 2d vector fields case

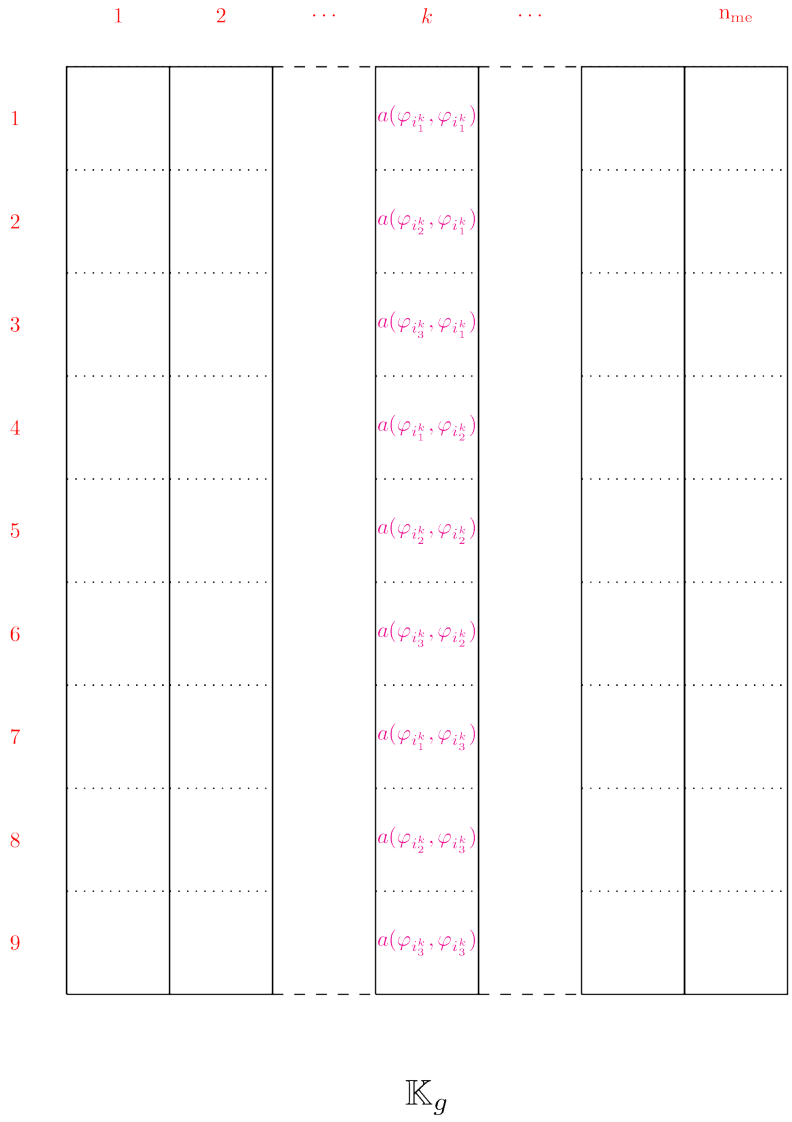


Figure 1.9: Construction of $\mathbb{K}_g$ for **Mass** matrix

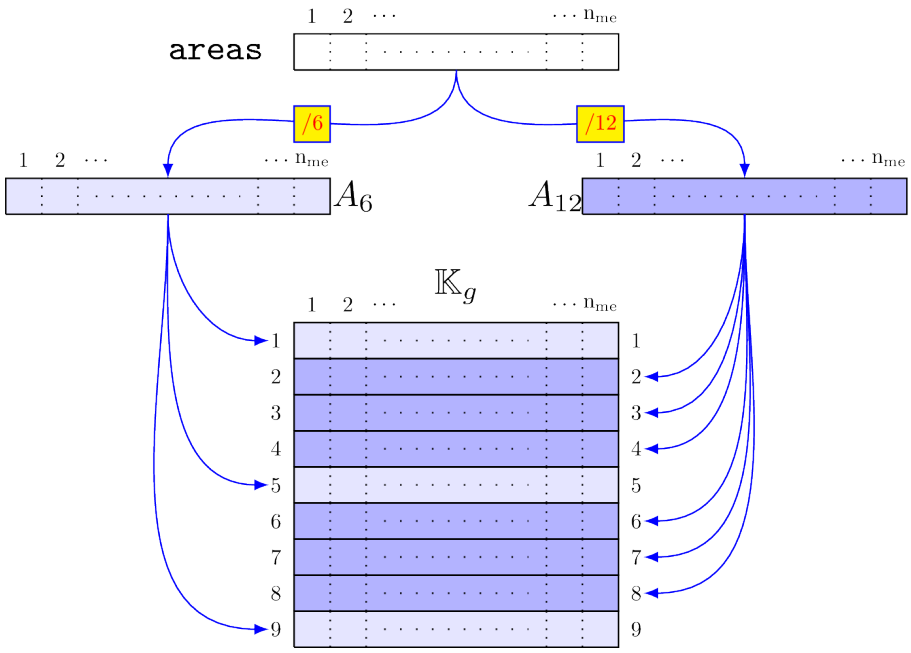


Figure 1.10: Construction of $\mathbb{K}_g$ for **Mass** matrix

CHAPTER

TWO

2D BENCHMARKS

Contents

2.1 Benchmark usage

pyOptFEM.FEM2D.assemblyBench.**assemblyBench**([*assembly=<string>*, *version=<string>*, *LN=<array>*, *meshname=<string>*, *meshdir=<string>*, *nbruns=<int>*, *la=<float>*, *mu=<float>*, *Num=<int>*, *tag=<string>*, ...])

Benchmark code for P_1-Lagrange finite element matrices defined in FEM2D.

Parameters

- **assembly** – Name of an assembly routine. The string should be :
  - *'MassAssembling2DP1'*,
  - *'StiffAssembling2DP1'* (default)
  - *'StiffElasAssembling2DP1'*
- **versions** – List of versions. Must be a list of any size whose elements may be *{'base','OptV1','OptV2'}* (default : `['OptV2','OptV1','base']`)
- **meshname** – Name of the mesh files. By default it is an empty string *''*. It means it uses SquareMesh function to generate meshes.
- **meshdir** – Directory location of FreeFEM++ mesh files. Used if meshname is not empty.
- **LN** – array of integer values. <LN> contains the values of N. Used to generate meshes data via Th=SquareMesh(N) function if meshname is empty or via *getMesh* class to read FreeFEM++ meshes Th=getMesh(meshdir+'/'+meshname+str(N)+'.msh')

  LN default value is `range(50, 90, 10)`
- **nbruns** – Number of runs on each mesh (default : *1*)
- **save** – For saving benchmark results in a file (see parameters <output>, <outdir> and <tag>)

- **plot** – For plotting computation times (default : *True*)
- **output** – Name used to set results file name (default : *'bench2D'*)
- **outdir** – Directory location of saved results file name (default : *'./results'*).
- **tag** – Specify a tag string added to filename
- **la** – the first Lame coefficient in Hooke's law, denoted by λ. Used by functions for assembling the elastic stiffness matrix. (default : *2*)
- **mu** – the second Lame coefficient in Hooke's law, denoted by μ. Used by functions for assembling the elastic stiffness matrix. (default : *0.3*)
- **Num** – Numbering choice. Used by functions for assembling the elastic stiffness matrix. (default : *0*)

2.2 Mass Matrix

- Benchmark of **MassAssembling2DP1** with `base`, `OptV1` and `OptV2` versions (see *Mass Matrix*)

Table 2.1: MassAssembling2DP1 benchmark summary

N	nq	ndof	OptV2	OptV1	base
50	2601	2601	0.0042(s)	0.0816(s)	3.2783(s)
			x1.00	x19.50	x783.85
60	3721	3721	0.0058(s)	0.1162(s)	4.7136(s)
			x1.00	x20.02	x812.43
70	5041	5041	0.0078(s)	0.1584(s)	6.3931(s)
			x1.00	x20.32	x819.97
80	6561	6561	0.0102(s)	0.2075(s)	8.4154(s)
			x1.00	x20.33	x824.26
90	8281	8281	0.0133(s)	0.2622(s)	10.6460(s)
			x1.00	x19.66	x798.28
100	10201	10201	0.0162(s)	0.3228(s)	13.1028(s)
			x1.00	x19.89	x807.38
110	12321	12321	0.0197(s)	0.3912(s)	15.9244(s)
			x1.00	x19.88	x809.23
120	14641	14641	0.0232(s)	0.4685(s)	18.9165(s)
			x1.00	x20.21	x816.09

This tabular was built with the following code :

```
>>> from pyOptFEM.FEM2D import assemblyBench
>>> assemblyBench(assembly='MassAssembling2DP1',LN=range(50,130,10))
SquareMesh(50):
 -> MassAssembling2DP1OptV2(nq=2601, nme=5000)
  run ( 1/ 1) : cputime=0.004182(s) - matrix 2601-by-2601
...
SquareMesh(120):
 -> MassAssembling2DP1base(nq=14641, nme=28800)
  run ( 1/ 1) : cputime=18.916504(s) - matrix 14641-by-14641
...
```

We also obtain

- Benchmark of **MassAssembling2DP1** with `OptV2` and `OptV1` versions

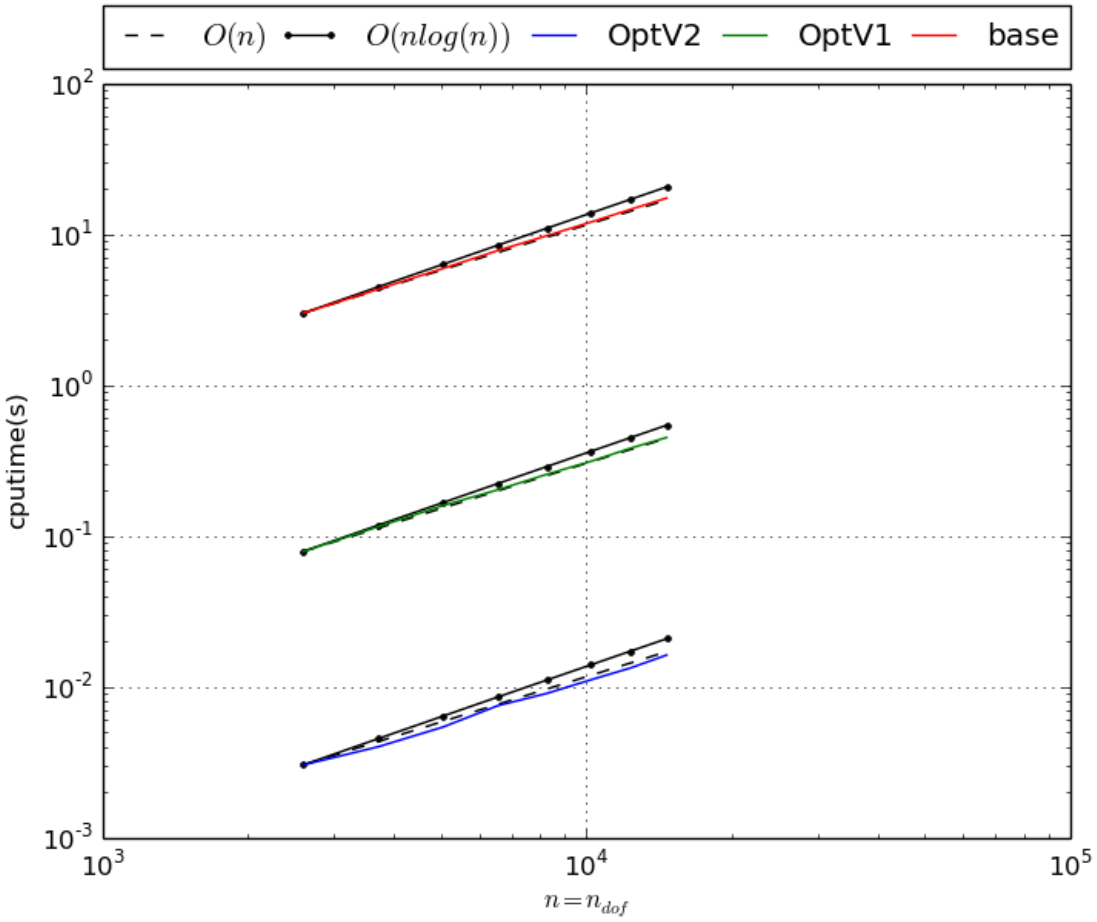


Figure 2.1: MassAssembling2DP1 benchmark

Table 2.2: MassAssembling2DP1 benchmark summary

N	nq	ndof	OptV2	OptV1
100	10201	10201	0.0163(s)	0.3196(s)
			x1.00	x19.61
200	40401	40401	0.0648(s)	1.3025(s)
			x1.00	x20.09
300	90601	90601	0.1583(s)	2.9198(s)
			x1.00	x18.45
400	160801	160801	0.2859(s)	5.1809(s)
			x1.00	x18.12
500	251001	251001	0.4591(s)	8.1059(s)
			x1.00	x17.65
600	361201	361201	0.6595(s)	11.6834(s)
			x1.00	x17.72
700	491401	491401	0.9056(s)	15.8209(s)
			x1.00	x17.47
800	641601	641601	1.1796(s)	20.8883(s)
			x1.00	x17.71
900	811801	811801	1.4933(s)	26.0264(s)
			x1.00	x17.43
1000	1002001	1002001	1.8503(s)	32.5076(s)
			x1.00	x17.57
1100	1212201	1212201	2.2262(s)	39.3237(s)
			x1.00	x17.66
1200	1442401	1442401	2.6616(s)	46.6362(s)
			x1.00	x17.52

This tabular was built with the following code :

```
>>> from pyOptFEM.FEM2D import assemblyBench
>>> assemblyBench(assembly='MassAssembling2DP1',versions=['OptV2','OptV1'],LN=range(100,1300,100
SquareMesh(100):
 -> MassAssembling2DP1OptV2(nq=10201, nme=20000)
  run ( 1/ 1) : cputime=0.016294(s) - matrix 10201-by-10201
...
SquareMesh(1200):
 -> MassAssembling2DP1OptV1(nq=1442401, nme=2880000)
  run ( 1/ 1) : cputime=46.636235(s) - matrix 1442401-by-1442401
...
```

We also obtain

2.3 Stiffness Matrix

- Benchmark of **StiffAssembling2DP1** with `base`, `OptV1` and `OptV2` versions

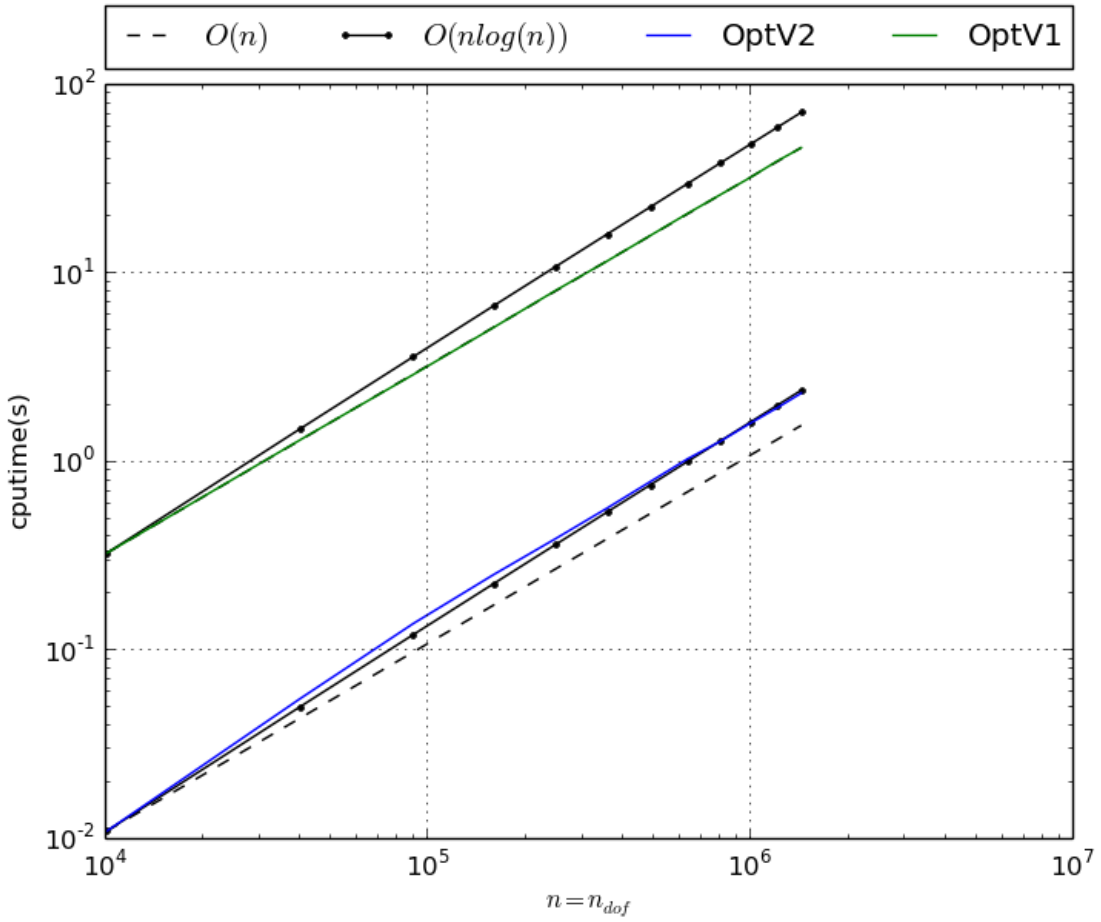


Figure 2.2: MassAssembling2DP1 benchmark

Table 2.3: StiffAssembling2DP1 benchmark summary

N	nq	ndof	OptV2	OptV1	base
50	2601	2601	0.0055(s)	0.2501(s)	3.5504(s)
			x1.00	x45.35	x643.82
60	3721	3721	0.0076(s)	0.3593(s)	5.1301(s)
			x1.00	x47.38	x676.64
70	5041	5041	0.0101(s)	0.4897(s)	6.9854(s)
			x1.00	x48.33	x689.37
80	6561	6561	0.0138(s)	0.6362(s)	9.0807(s)
			x1.00	x46.11	x658.09
90	8281	8281	0.0178(s)	0.8119(s)	11.5354(s)
			x1.00	x45.57	x647.46
100	10201	10201	0.0223(s)	1.0005(s)	14.2190(s)
			x1.00	x44.77	x636.22
110	12321	12321	0.0266(s)	1.2124(s)	17.2704(s)
			x1.00	x45.61	x649.74
120	14641	14641	0.0325(s)	1.4503(s)	20.4368(s)
			x1.00	x44.69	x629.75

This tabular was built with the following code :

```
>>> from pyOptFEM.FEM2D import assemblyBench
>>> assemblyBench(assembly='StiffAssembling2DP1',LN=range(50,130,10))
SquareMesh(50):
 -> StiffAssembling2DP1OptV2(nq=2601, nme=5000)
  run ( 1/ 1) : cputime=0.005515(s) - matrix 2601-by-2601
...
SquareMesh(120):
 -> StiffAssembling2DP1base(nq=14641, nme=28800)
  run ( 1/ 1) : cputime=20.436779(s) - matrix 14641-by-14641
...
```

We also obtain

- Benchmark of **StiffAssembling2DP1** with `OptV2` and `OptV1` versions

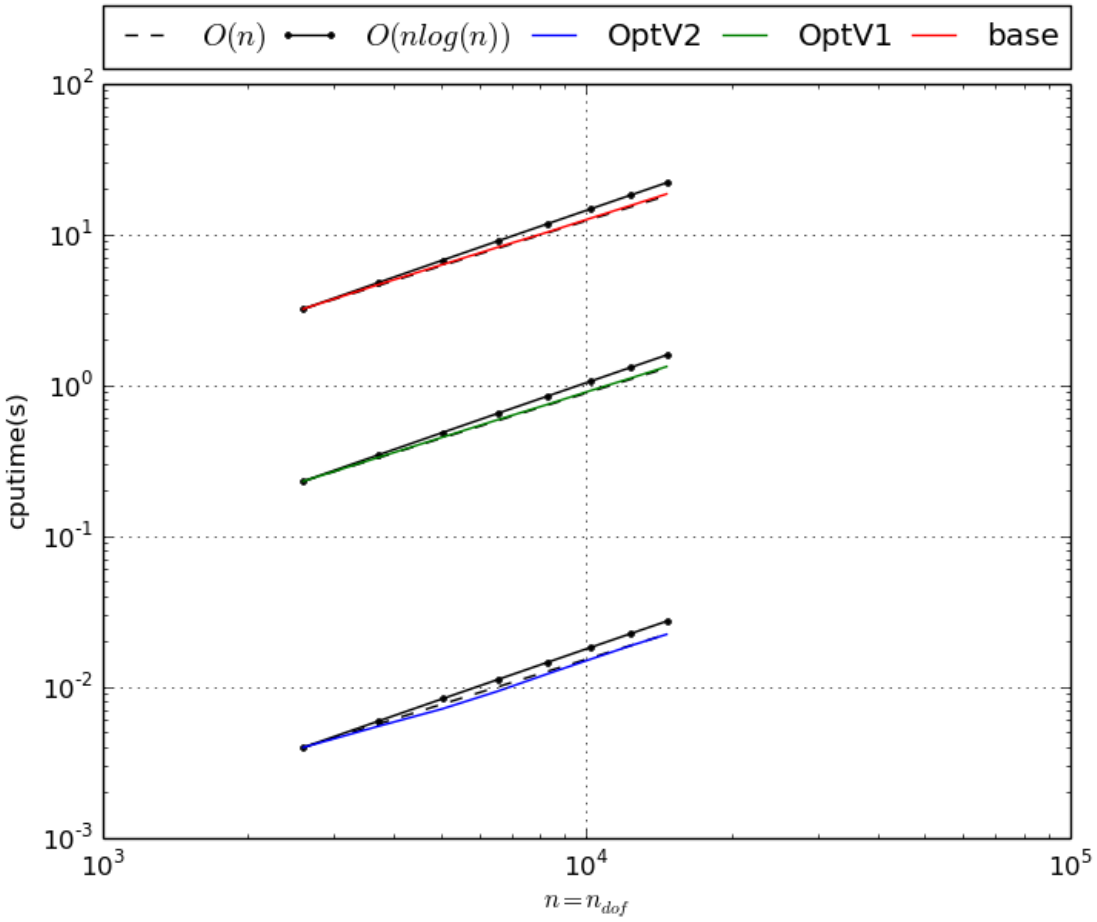


Figure 2.3: StiffAssembling2DP1 benchmark

Table 2.4: StiffAssembling2DP1 benchmark summary

N	nq	ndof	OptV2	OptV1
100	10201	10201	0.0225(s)	1.0104(s)
			x1.00	x44.93
200	40401	40401	0.0945(s)	4.0847(s)
			x1.00	x43.22
300	90601	90601	0.2379(s)	9.1818(s)
			x1.00	x38.59
400	160801	160801	0.4459(s)	16.3333(s)
			x1.00	x36.63
500	251001	251001	0.7169(s)	25.6513(s)
			x1.00	x35.78
600	361201	361201	1.0316(s)	36.8382(s)
			x1.00	x35.71
700	491401	491401	1.4316(s)	49.9507(s)
			x1.00	x34.89
800	641601	641601	1.8965(s)	65.8354(s)
			x1.00	x34.71
900	811801	811801	2.4164(s)	82.7545(s)
			x1.00	x34.25
1000	1002001	1002001	2.9450(s)	102.3931(s)
			x1.00	x34.77
1100	1212201	1212201	3.4908(s)	123.9802(s)
			x1.00	x35.52
1200	1442401	1442401	4.3144(s)	147.2963(s)
			x1.00	x34.14

This tabular was built with the following code :

```
>>> from pyOptFEM.FEM2D import assemblyBench
>>> assemblyBench(assembly='StiffAssembling2DP1',versions=['OptV2','OptV1'],LN=range(100,1300,10
SquareMesh(100):
 -> StiffAssembling2DP1OptV2(nq=10201, nme=20000)
  run ( 1/ 1) : cputime=0.022488(s) - matrix 10201-by-10201
...
SquareMesh(1200):
 -> StiffAssembling2DP1OptV1(nq=1442401, nme=2880000)
  run ( 1/ 1) : cputime=147.296321(s) - matrix 1442401-by-1442401
...
```

We also obtain

2.4 Elastic Stiffness Matrix

- Benchmark of **StiffElasAssembling2DP1** with `base`, `OptV1` and `OptV2` versions

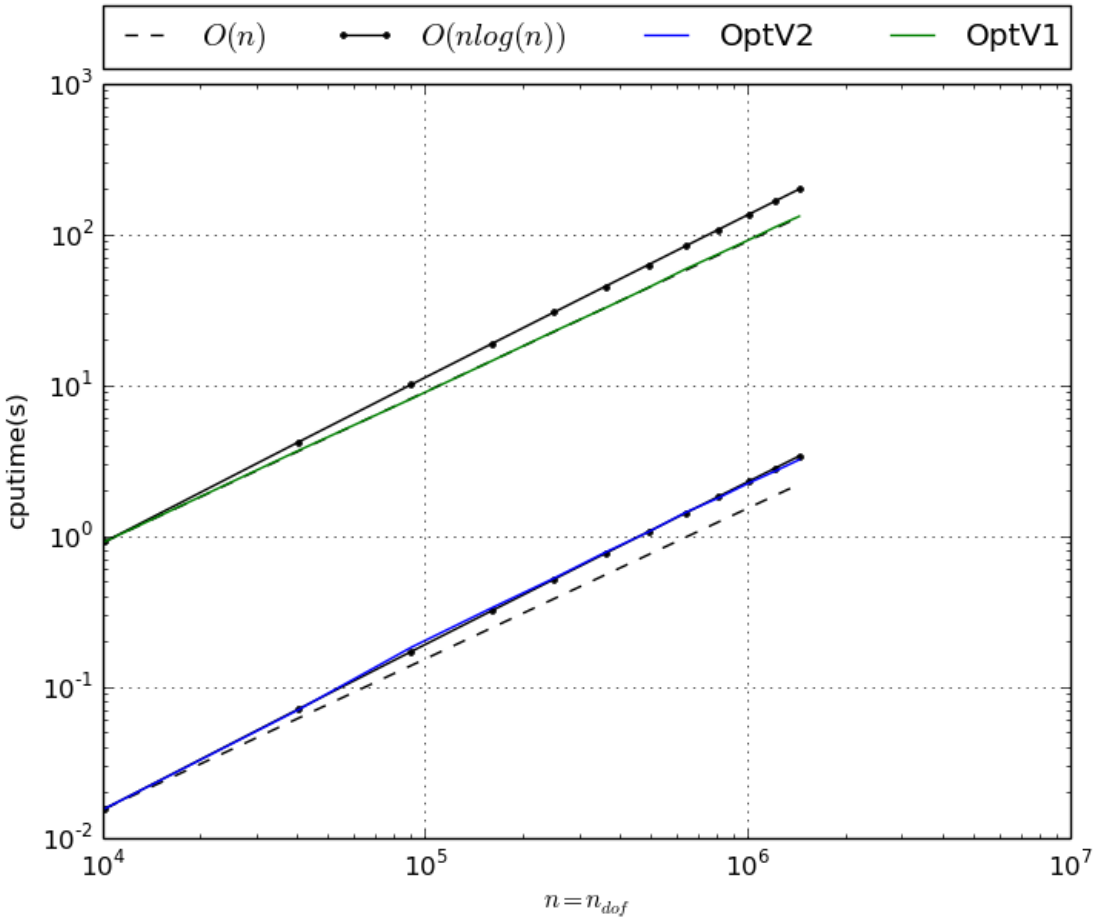


Figure 2.4: StiffAssembling2DP1 benchmark

Table 2.5: StiffElasAssembling2DP1 benchmark summary

N	nq	ndof	OptV2	OptV1	base
30	961	1922	0.0107(s)	0.2809(s)	1.8485(s)
			x1.00	x26.29	x173.04
35	1296	2592	0.0134(s)	0.3807(s)	2.5109(s)
			x1.00	x28.37	x187.11
40	1681	3362	0.0172(s)	0.4986(s)	3.2952(s)
			x1.00	x28.92	x191.13
45	2116	4232	0.0216(s)	0.6306(s)	4.1589(s)
			x1.00	x29.26	x192.97
50	2601	5202	0.0269(s)	0.7800(s)	5.1323(s)
			x1.00	x29.04	x191.07
55	3136	6272	0.0322(s)	0.9453(s)	6.2180(s)
			x1.00	x29.39	x193.35
60	3721	7442	0.0386(s)	1.1282(s)	7.4995(s)
			x1.00	x29.21	x194.14

This tabular was built with the following code :

```
>>> from pyOptFEM.FEM2D import assemblyBench
>>> assemblyBench(assembly='StiffElasAssembling2DP1',LN=range(30,65,5))
SquareMesh(30):
 -> StiffElasAssembling2DP1OptV2(nq=961, nme=1800)
  run ( 1/ 1) : cputime=0.010683(s) - matrix 1922-by-1922
...
SquareMesh(30):
 -> StiffElasAssembling2DP1OptV2(nq=961, nme=1800)
  run ( 1/ 1) : cputime=0.010683(s) - matrix 1922-by-1922
...
```

We also obtain

- Benchmark of **StiffElasAssembling2DP1** with `OptV2` and `OptV1` versions

Table 2.6: StiffElasAssembling2DP1 benchmark summary

N	nq	ndof	OptV2	OptV1
100	10201	20402	0.1110(s)	3.1286(s)
			x1.00	x28.18
200	40401	80802	0.5036(s)	12.5861(s)
			x1.00	x24.99
300	90601	181202	1.2058(s)	28.2232(s)
			x1.00	x23.41
400	160801	321602	2.1346(s)	50.2407(s)
			x1.00	x23.54
500	251001	502002	3.5360(s)	78.4291(s)
			x1.00	x22.18
600	361201	722402	5.3290(s)	113.4194(s)
			x1.00	x21.28
700	491401	982802	7.4691(s)	153.6793(s)
			x1.00	x20.58
800	641601	1283202	9.8943(s)	201.4319(s)
			x1.00	x20.36

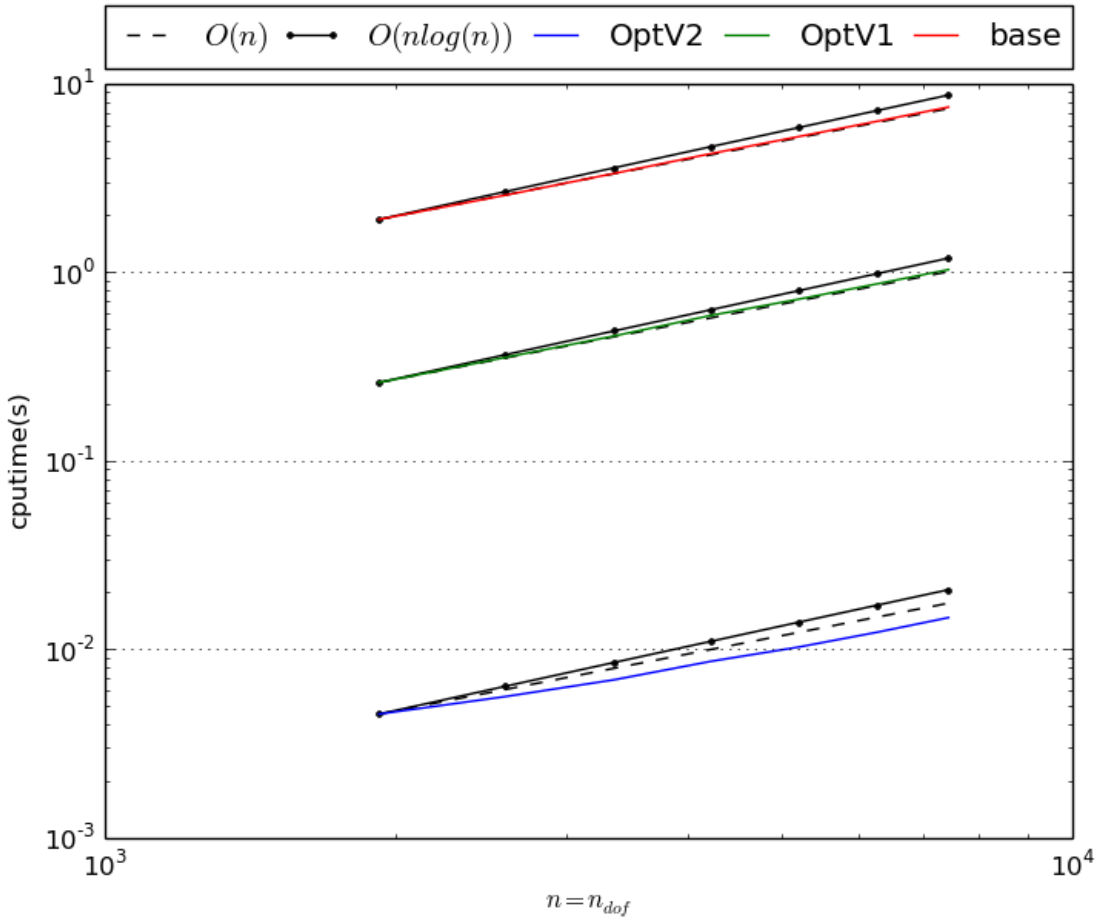


Figure 2.5: StiffElasAssembling2DP1 benchmark

This tabular was built with the following code :

```
>>> from pyOptFEM.FEM2D import assemblyBench
>>> assemblyBench(assembly='StiffElasAssembling2DP1',versions=['OptV2','OptV1'],LN=range(100,900
SquareMesh(100):
 -> StiffElasAssembling2DP1OptV2(nq=10201, nme=20000)
  run ( 1/ 1) : cputime=0.111037(s) - matrix 20402-by-20402
...
SquareMesh(800):
 -> StiffElasAssembling2DP1OptV1(nq=641601, nme=1280000)
  run ( 1/ 1) : cputime=201.431866(s) - matrix 1283202-by-1283202
...
```

We also obtain

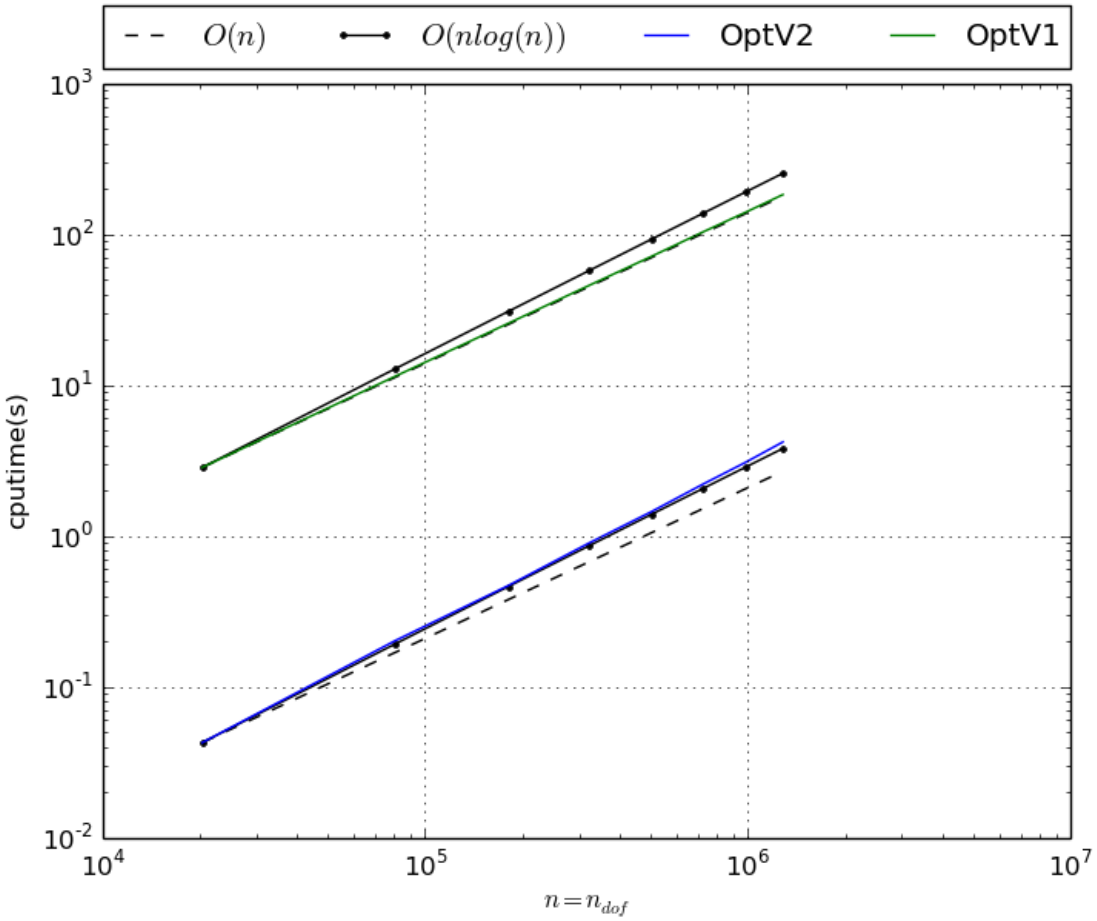


Figure 2.6: StiffElasAssembling2DP1 benchmark

CHAPTER

THREE

3D BENCHMARKS

Contents

3.1 Benchmark usage

pyOptFEM.FEM3D.assemblyBench.**assemblyBench**([*assembly=<string>*, *version=<string>*, *LN=<array>*, *meshname=<string>*, *meshdir=<string>*, *nbruns=<int>*, *la=<float>*, *mu=<float>*, *Num=<int>*, *tag=<string>*, ...])

Benchmark code for P_1-Lagrange finite element matrices defined in FEM3D.

Parameters

- **assembly** – Name of an assembly routine. The string should be :
  - *'MassAssembling3DP1'*,
  - *'StiffAssembling3DP1'* (default)
  - *'StiffElasAssembling3DP1'*
- **versions** – List of versions. Must be a list of any size whose elements may be *{'base','OptV1','OptV2'}* (default : ['OptV2','OptV1','base'])
- **meshname** – Name of the *medit* mesh files. By default it is an empty string *''*. It means it uses CubeMesh(N) function to generate meshes.
- **meshdir** – Directory location of *medit* mesh files. Used if meshname is not empty.
- **LN** – array of integer values. <LN> contains the values of N. Used to generate meshes data via Th=CubeMesh(N) function if meshname is empty or via *getMesh* class to read FreeFEM++ meshes Th=getMesh(meshdir+'/'+meshname+str(N)+'.mesh')

  LN default value is range(5, 14, 2)
- **nbruns** – Number of runs on each mesh (default : *1*)
- **save** – For saving benchmark results in a file (see parameters <output>, <outdir> and <tag>). (default : *False*)

- **plot** – For plotting computation times (default : *True*)
- **output** – Name used to set results file name (default : *'bench2D'*)
- **outdir** – Directory location of saved results file name (default : *'./results'*).
- **tag** – Specify a tag string added to filename
- **la** – the first Lame coefficient in Hooke's law, denoted by λ. Used by functions for assembling the elastic stiffness matrix. (default : *2*)
- **mu** – the second Lame coefficient in Hooke's law, denoted by μ. Used by functions for assembling the elastic stiffness matrix. (default : *0.3*)
- **Num** – Numbering choice. Used by functions for assembling the elastic stiffness matrix. (default : *0*)

3.2 Mass Matrix

- Benchmark of **MassAssembling3DP1** with `base`, `OptV1` and `OptV2` versions (see *Mass Matrix*)

Table 3.1: MassAssembling23P1 benchmark summary

N	nq	ndof	OptV2	OptV1	base
5	216	216	0.0015(s)	0.0424(s)	0.8962(s)
			x1.00	x28.29	x597.49
10	1331	1331	0.0075(s)	0.3390(s)	7.1521(s)
			x1.00	x44.96	x948.64
15	4096	4096	0.0268(s)	1.1413(s)	24.2836(s)
			x1.00	x42.55	x905.35
20	9261	9261	0.0677(s)	2.7240(s)	56.9878(s)
			x1.00	x40.26	x842.21
25	17576	17576	0.1457(s)	5.2998(s)	111.0287(s)
			x1.00	x36.38	x762.11
30	29791	29791	0.2493(s)	9.2205(s)	192.1030(s)
			x1.00	x36.99	x770.64
35	46656	46656	0.4121(s)	14.6558(s)	304.5941(s)
			x1.00	x35.57	x739.17

This tabular was built with the following code :

```
>>> from pyOptFEM.FEM3D import assemblyBench
>>> assemblyBench(assembly='MassAssembling3DP1',LN=range(5,40,5))
CubeMesh(5):
 -> MassAssembling3DP1OptV2(nq=216, nme=750)
  run ( 1/ 1) : cputime=0.001254(s) - matrix 216-by-216
...
CubeMesh(35):
 -> MassAssembling3DP1base(nq=46656, nme=257250)
  run ( 1/ 1) : cputime=304.594078(s) - matrix 46656-by-46656
```

We also obtain

- Benchmark of **MassAssembling3DP1** with `OptV2` and `OptV1` versions

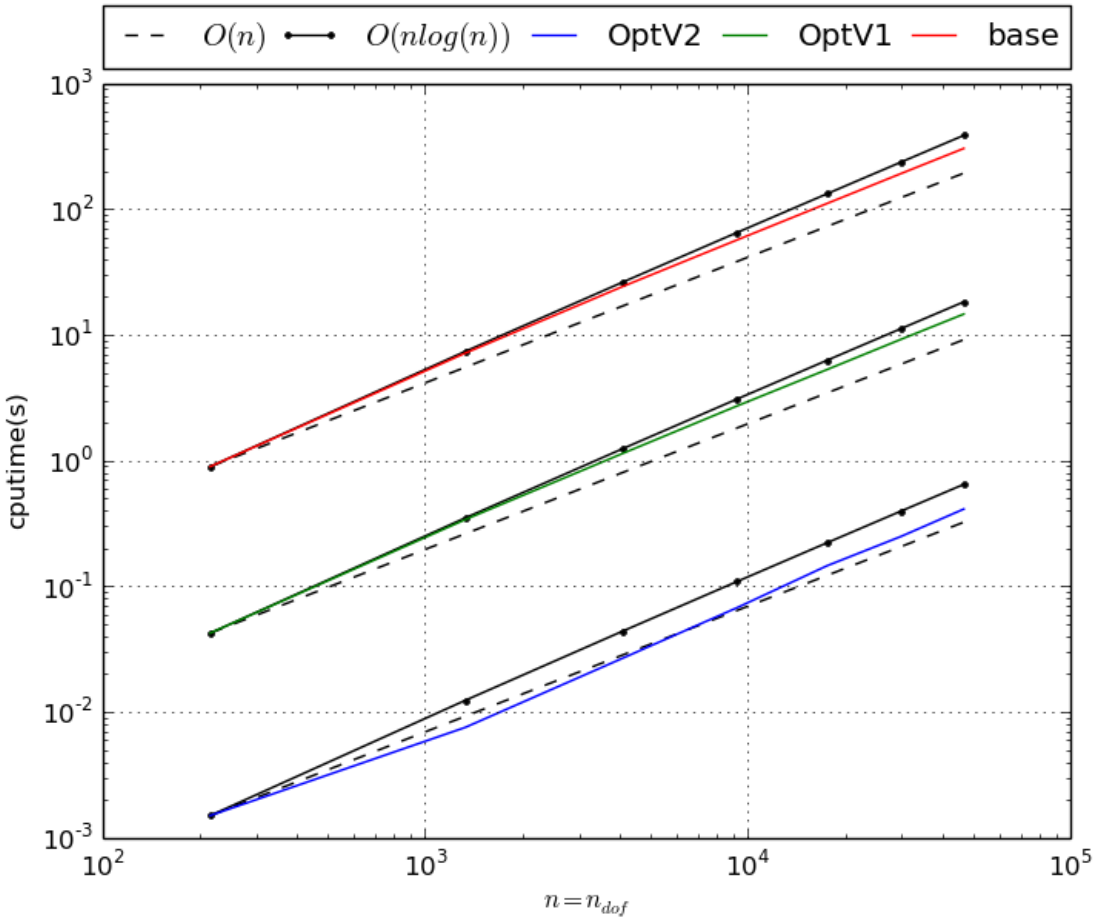


Figure 3.1: MassAssembling3DP1 benchmark

Table 3.2: MassAssembling2DP1 benchmark summary

N	nq	ndof	OptV2	OptV1
10	1331	1331	0.0065(s)	0.3656(s)
			x1.00	x56.05
20	9261	9261	0.0578(s)	2.6968(s)
			x1.00	x46.70
30	29791	29791	0.2195(s)	9.1710(s)
			x1.00	x41.77
40	68921	68921	0.6109(s)	21.7386(s)
			x1.00	x35.59
50	132651	132651	1.1558(s)	42.1679(s)
			x1.00	x36.49
60	226981	226981	2.1182(s)	73.9066(s)
			x1.00	x34.89
70	357911	357911	3.1952(s)	116.2151(s)
			x1.00	x36.37
80	531441	531441	5.1899(s)	174.4814(s)
			x1.00	x33.62
90	753571	753571	7.0460(s)	247.8245(s)
			x1.00	x35.17
100	1030301	1030301	9.8218(s)	340.9609(s)
			x1.00	x34.71

This tabular was built with the following code :

```
>>> from pyOptFEM.FEM3D import assemblyBench
>>> assemblyBench(assembly='MassAssembling3DP1',versions=['OptV2','OptV1'],LN=range(10,110,10))
CubeMesh(10):
 -> MassAssembling3DP1OptV2(nq=1331, nme=6000)
  run ( 1/ 1) : cputime=0.006523(s) - matrix 1331-by-1331
...
CubeMesh(100):
 -> MassAssembling3DP1OptV1(nq=1030301, nme=6000000)
  run ( 1/ 1) : cputime=340.960874(s) - matrix 1030301-by-1030301
```

We also obtain

3.3 Stiffness Matrix

- Benchmark of **StiffAssembling3DP1** with `base`, `OptV1` and `OptV2` versions

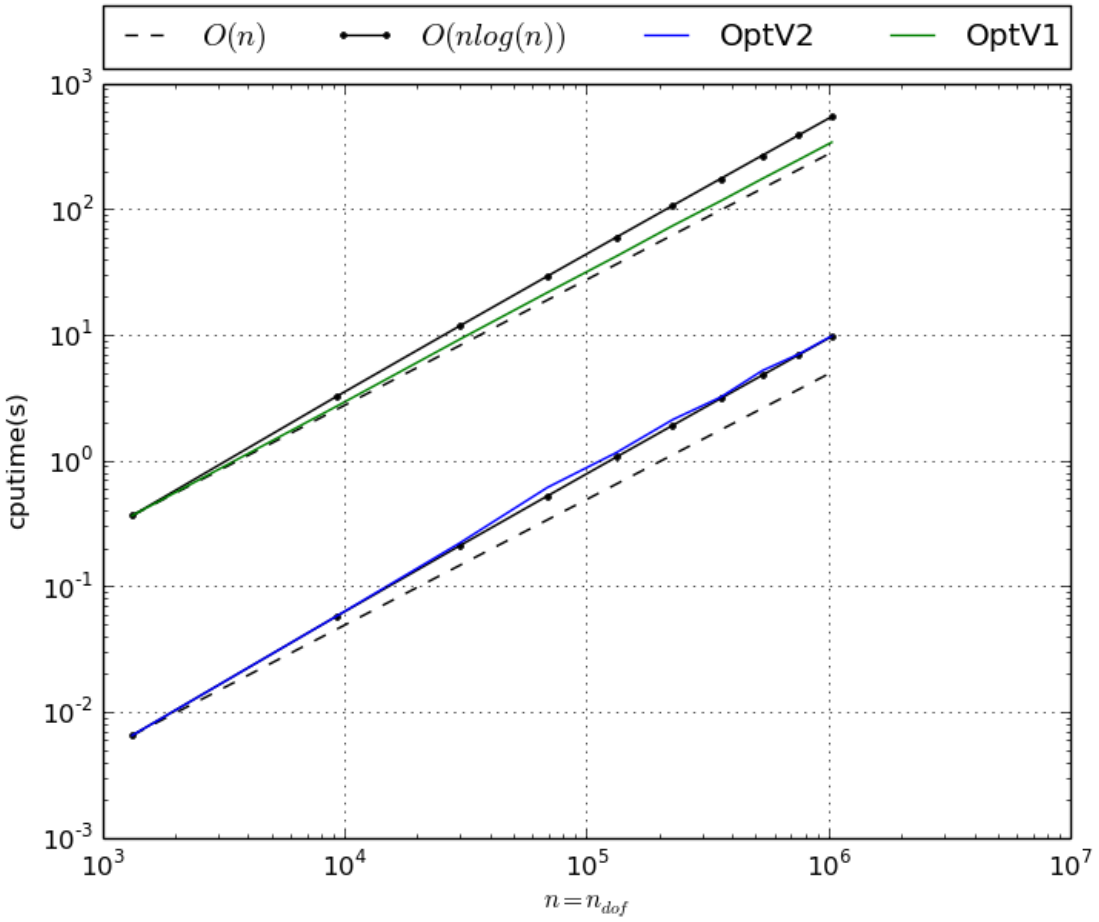


Figure 3.2: MassAssembling3DP1 benchmark

Table 3.3: StiffAssembling3DP1 benchmark summary

N	nq	ndof	OptV2	OptV1	base
5	216	216	0.0024(s)	0.0546(s)	0.9065(s)
			x1.00	x23.22	x385.72
10	1331	1331	0.0072(s)	0.4396(s)	7.2347(s)
			x1.00	x60.68	x998.63
15	4096	4096	0.0258(s)	1.4985(s)	24.5593(s)
			x1.00	x58.08	x951.81
20	9261	9261	0.0652(s)	3.4915(s)	57.9647(s)
			x1.00	x53.55	x888.96
25	17576	17576	0.1315(s)	6.7851(s)	112.9478(s)
			x1.00	x51.61	x859.14
30	29791	29791	0.2479(s)	11.8616(s)	195.8290(s)
			x1.00	x47.85	x790.05
35	46656	46656	0.4126(s)	18.3069(s)	310.0601(s)
			x1.00	x44.37	x751.46

This tabular was built with the following code :

```
>>> from pyOptFEM.FEM3D import assemblyBench
>>> assemblyBench(assembly='StiffAssembling3DP1',LN=range(5,40,5))
CubeMesh(5):
 -> StiffAssembling3DP1OptV2(nq=216, nme=750)
  run ( 1/ 1) : cputime=0.002350(s) - matrix 216-by-216
...
CubeMesh(35):
 -> StiffAssembling3DP1base(nq=46656, nme=257250)
  run ( 1/ 1) : cputime=310.060121(s) - matrix 46656-by-46656
```

We also obtain

- Benchmark of **StiffAssembling3DP1** with `OptV2` and `OptV1` versions

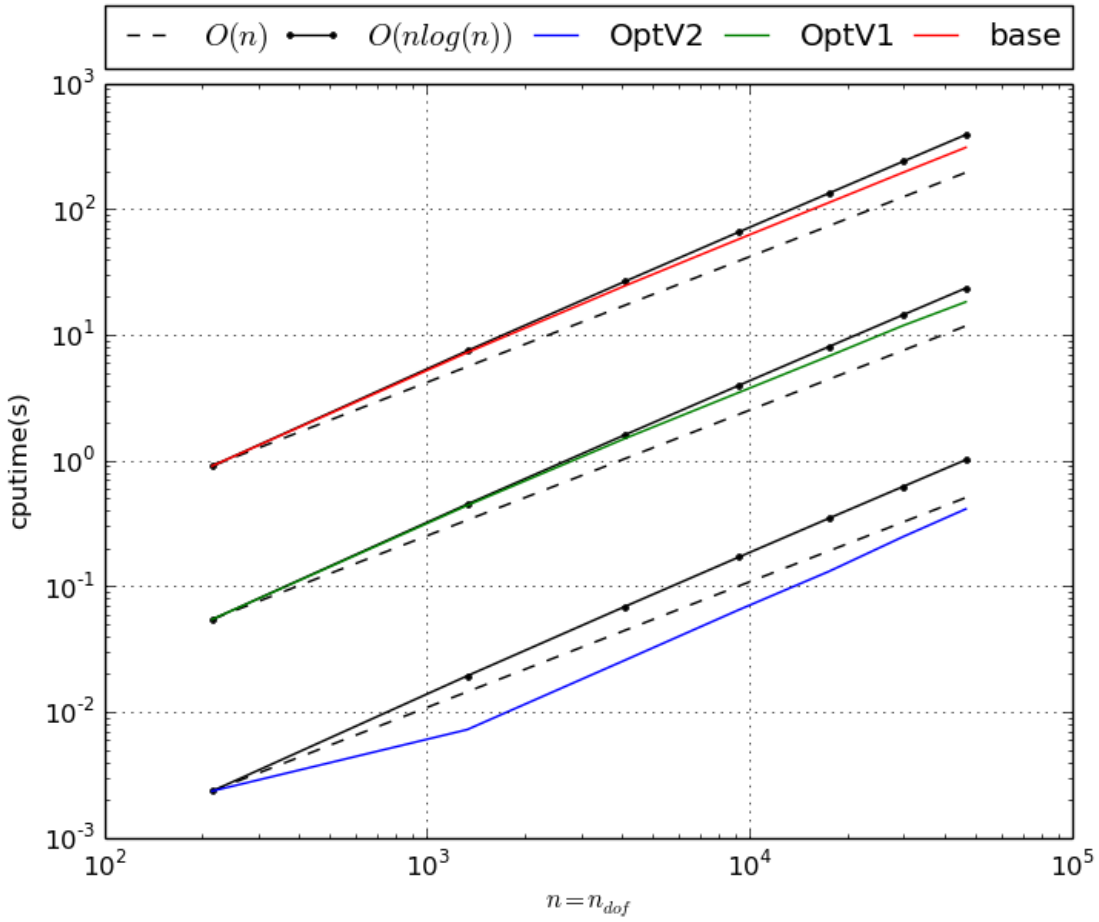


Figure 3.3: StiffAssembling3DP1 benchmark

Table 3.4: StiffAssembling3DP1 benchmark summary

N	nq	ndof	OptV2	OptV1
10	1331	1331	0.0071(s)	0.4329(s)
			x1.00	x60.68
20	9261	9261	0.0652(s)	3.4323(s)
			x1.00	x52.64
30	29791	29791	0.2525(s)	11.6349(s)
			x1.00	x46.07
40	68921	68921	0.7178(s)	27.6186(s)
			x1.00	x38.48
50	132651	132651	1.5883(s)	53.9955(s)
			x1.00	x34.00
60	226981	226981	2.8236(s)	93.1197(s)
			x1.00	x32.98
70	357911	357911	4.6881(s)	147.6608(s)
			x1.00	x31.50
80	531441	531441	7.4632(s)	220.7232(s)
			x1.00	x29.57
90	753571	753571	10.6093(s)	314.8404(s)
			x1.00	x29.68
100	1030301	1030301	14.7064(s)	427.0878(s)
			x1.00	x29.04

This tabular was built with the following code :

```
>>> from pyOptFEM.FEM3D import assemblyBench
>>> assemblyBench(assembly='StiffAssembling3DP1',versions=['OptV2','OptV1'],LN=range(10,110,10))
CubeMesh(10):
 -> StiffAssembling3DP1OptV2(nq=1331, nme=6000)
  run ( 1/ 1) : cputime=0.007133(s) - matrix 1331-by-1331
...
CubeMesh(100):
 -> StiffAssembling3DP1OptV1(nq=1030301, nme=6000000)
run ( 1/ 1) : cputime=427.087812(s) - matrix 1030301-by-1030301
```

We also obtain

3.4 Elastic Stiffness Matrix

- Benchmark of **StiffAssembling3DP1** with `base`, `OptV1` and `OptV2` versions

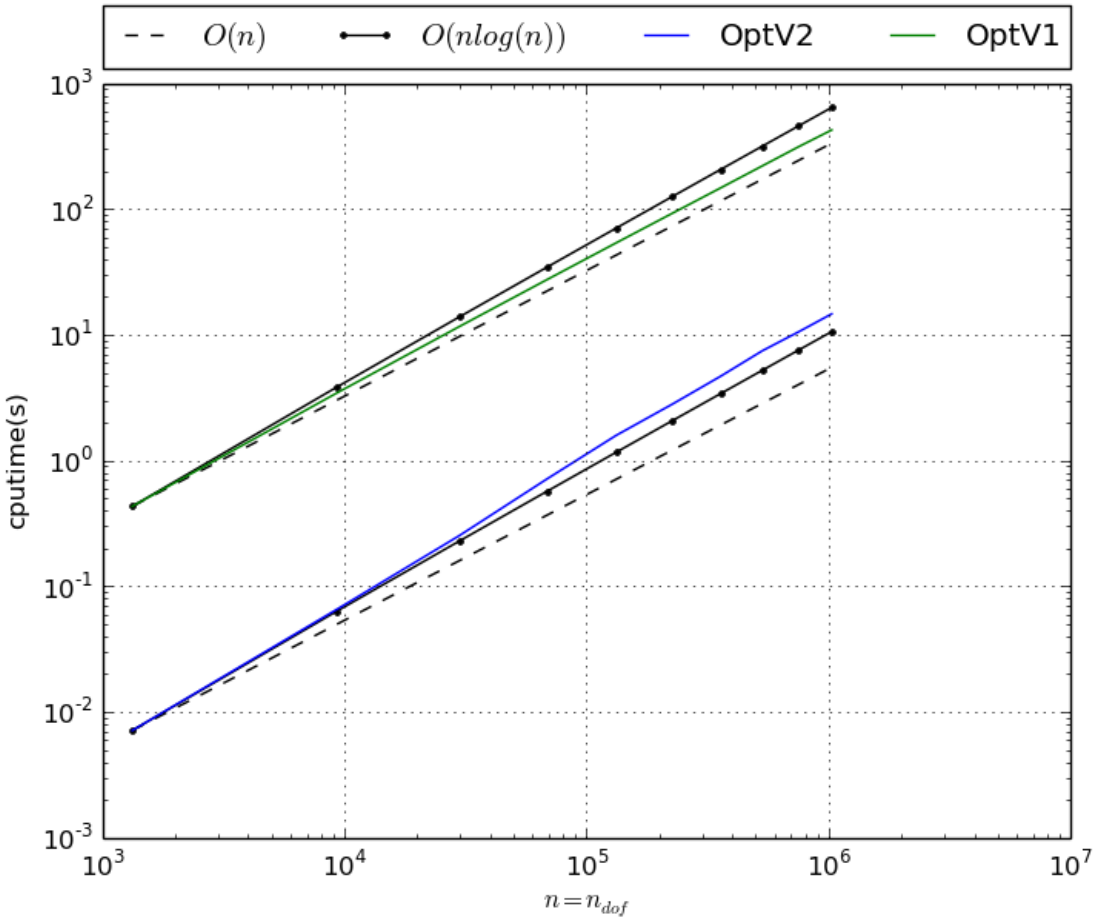


Figure 3.4: StiffAssembling3DP1 benchmark

Table 3.5: StiffAssembling3DP1 benchmark summary

N	nq	ndof	OptV2	OptV1	base
3	64	192	0.0056(s)	0.0626(s)	0.6896(s)
			x1.00	x11.23	x123.60
5	216	648	0.0085(s)	0.2877(s)	3.1838(s)
			x1.00	x33.87	x374.88
7	512	1536	0.0183(s)	0.7933(s)	8.7350(s)
			x1.00	x43.43	x478.25
10	1331	3993	0.0493(s)	2.3134(s)	25.5321(s)
			x1.00	x46.91	x517.74
13	2744	8232	0.1061(s)	5.0799(s)	55.8011(s)
			x1.00	x47.89	x526.04
17	5832	17496	0.2426(s)	11.3663(s)	126.2934(s)
			x1.00	x46.86	x520.66
20	9261	27783	0.4317(s)	18.6554(s)	205.4239(s)
			x1.00	x43.22	x475.90

This tabular was built with the following code :

```
>>> from pyOptFEM.FEM3D import assemblyBench
>>> from numpy import *
>>> assemblyBench(assembly='StiffElasAssembling3DP1',LN=array([3,5,7,10,13,17,20]))
CubeMesh(3):
 -> StiffElasAssembling3DP1OptV2(nq=64, nme=162)
  run ( 1/ 1) : cputime=0.005579(s) - matrix 192-by-192
...
CubeMesh(20):
 -> StiffElasAssembling3DP1base(nq=9261, nme=48000)
  run ( 1/ 1) : cputime=205.423872(s) - matrix 27783-by-27783
```

We also obtain

- Benchmark of **StiffAssembling3DP1** with `OptV1` and `OptV2` versions

Table 3.6: StiffAssembling3DP1 benchmark summary

N	nq	ndof	OptV2	OptV1
10	1331	3993	0.0497(s)	2.2711(s)
			x1.00	x45.73
20	9261	27783	0.4325(s)	18.1990(s)
			x1.00	x42.07
30	29791	89373	1.7316(s)	61.6716(s)
			x1.00	x35.62
40	68921	206763	4.5098(s)	146.4793(s)
			x1.00	x32.48
50	132651	397953	9.3904(s)	284.8776(s)
			x1.00	x30.34
60	226981	680943	17.2215(s)	494.1435(s)
			x1.00	x28.69
70	357911	1073733	28.1707(s)	785.3533(s)
			x1.00	x27.88

This tabular was built with the following code :

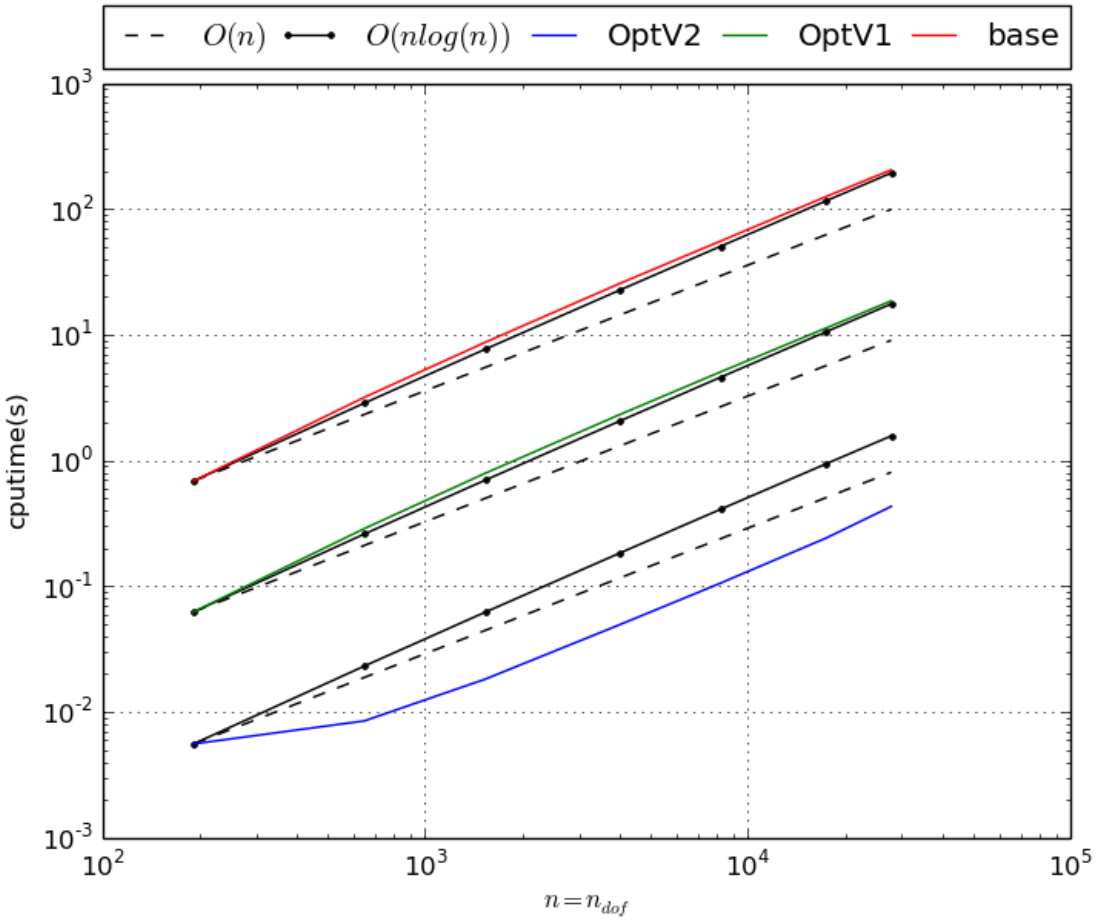


Figure 3.5: StiffElasAssembling3DP1 benchmark

```
>>> from pyOptFEM.FEM3D import assemblyBench
>>> assemblyBench(assembly='StiffElasAssembling3DP1',versions=['OptV2','OptV1'],LN=range(10,80,1
CubeMesh(10):
 -> StiffElasAssembling3DP1OptV2(nq=1331, nme=6000)
  run ( 1/ 1) : cputime=0.049666(s) - matrix 3993-by-3993
...
CubeMesh(70):
 -> StiffElasAssembling3DP1OptV1(nq=357911, nme=2058000)
  run ( 1/ 1) : cputime=785.353305(s) - matrix 1073733-by-1073733
```

We also obtain

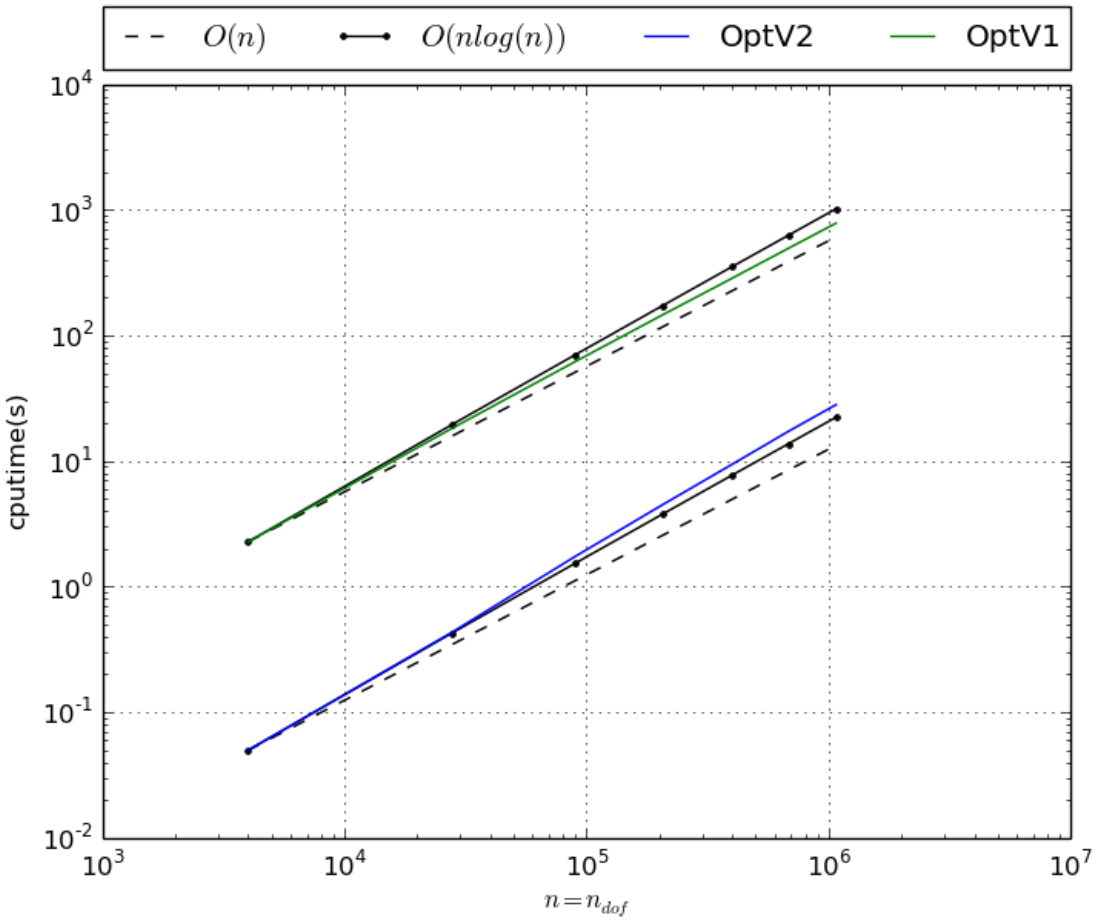


Figure 3.6: StiffElasAssembling3DP1 benchmark

CHAPTER

FOUR

FEM2D MODULE

Author Francois Cuvelier <cuvelier@math.univ-paris13.fr>

Date 15/09/2013

Contains functions to build some finite element matrices using P_1-Lagrange finite elements on a 2D mesh. Each assembly matrix is computed by three different versions called `base`, `OptV1` and `OptV2` (see here)

Contents

4.1 Assembly matrices (`base`, `OptV1` and `OptV2` versions)

Let $\mathcal{T}_h$ be a triangular mesh of Ω. We denote by $\Omega_h = \bigcup_{T_k \in \mathcal{T}_h} T_k$ a triangulation of Ω with the following data structure:

name	type	dimension	description	Python
$\mathrm{n_q}$	integer	1	number of vertices	`nq`
$\mathrm{n_{me}}$	integer	1	number of elements	`nme`
q	double	$2 \times \mathrm{n_q}$	array of vertices coordinates. $\mathrm{q}(\nu, j)$ is the ν-th coordinate of the j-th vertex, $\nu \in \{1, 2\}$, $j \in \{1, \ldots, \mathrm{n_q}\}$. The j-th vertex will be also denoted by $\mathrm{q^j}$	`q` (transposed) `q[j-1]` = q^j
me	integer	$3 \times \mathrm{n_{me}}$	connectivity array. $\mathrm{me}(\beta, k)$ is the storage index of the β-th vertex of the k-th element, in the array q, for $\beta \in \{1, 2, 3\}$ and $k \in \{1, \ldots, \mathrm{n_{me}}\}$	`me` (transposed)
areas	double	$1 \times \mathrm{n_{me}}$	array of areas. $\mathrm{areas}(k)$ is the k-th triangle area, $k \in \{1, \ldots, \mathrm{n_{me}}\}$	`areas`

The P_1-Lagrange basis functions associated to Ω_h are denoted by φ_i for all $i \in \{1, \ldots, \mathrm{n_q}\}$ and defined by

$$\varphi_i(\mathrm{q^j}) = \delta_{\mathrm{i,j}}, \quad \forall (\mathrm{i}, \mathrm{j}) \in \{1, \ldots, \mathrm{n_q}\}^2$$

We also define the global *alternate* basis $\mathcal{B}_a$ by

$$\mathcal{B}_a = \{\boldsymbol{\psi}_1, \ldots, \boldsymbol{\psi}_{2\mathrm{n_q}}\} = \left\{ \begin{pmatrix} \varphi_1 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ \varphi_1 \end{pmatrix}, \begin{pmatrix} \varphi_2 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ \varphi_2 \end{pmatrix}, \ldots, \begin{pmatrix} \varphi_{\mathrm{n_q}} \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ \varphi_{\mathrm{n_q}} \end{pmatrix} \right\}$$

and the global *block* basis $\mathcal{B}_b$ by

$$\mathcal{B}_b = \{\boldsymbol{\phi}_1, \ldots, \boldsymbol{\phi}_{2\mathrm{n_q}}\} = \left\{ \begin{pmatrix} \varphi_1 \\ 0 \end{pmatrix}, \begin{pmatrix} \varphi_2 \\ 0 \end{pmatrix}, \ldots, \begin{pmatrix} \varphi_{\mathrm{n_q}} \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ \varphi_1 \end{pmatrix}, \begin{pmatrix} 0 \\ \varphi_2 \end{pmatrix}, \ldots, \begin{pmatrix} 0 \\ \varphi_{\mathrm{n_q}} \end{pmatrix} \right\}.$$

4.1.1 Mass Matrix

Assembly of the Mass Matrix by P_1-Lagrange finite elements using `base`, `OptV1` and `OptV2` versions respectively (see report). The Mass Matrix $\mathbb{M}$ is given by

$$\mathbb{M}_{i,j} = \int_{\Omega_h} \varphi_j(q)\, \varphi_i(q) dq, \quad \forall (i, j) \in \{1, ..., \mathrm{n_q}\}^2.$$

Note: generic syntax:

```
M = MassAssembling2DP1<version>(nq,nme,me,areas)
```

- `nq`: total number of nodes of the mesh, also denoted by $\mathrm{n_q}$,
- `nme`: total number of triangles, also denoted by $\mathrm{n_{me}}$,

- me: Connectivity array, (nme,3) array,
- areas: Array of areas, (nme,) array,
- **returns** a *Scipy* CSC sparse matrix of size $\mathrm{n_q} \times \mathrm{n_q}$

where <version> is base, OptV1 or OptV2

Benchmarks of theses functions are presented in *Mass Matrix*. We give a simple usage :

```
>>> from pyOptFEM.FEM2D import *
>>> Th=SquareMesh(5)
>>> Mbase =  MassAssembling2DP1base(Th.nq,Th.nme,Th.me,Th.areas)
>>> MOptV1= MassAssembling2DP1OptV1(Th.nq,Th.nme,Th.me,Th.areas)
>>> print(" NormInf(Mbase-MOptV1)=%e " % NormInf(Mbase-MOptV1))
NormInf(Mbase-MOptV1)=6.938894e-18
>>> MOptV2= MassAssembling2DP1OptV2(Th.nq,Th.nme,Th.me,Th.areas)
>>> print(" NormInf(Mbase-MOptV2)=%e " % NormInf(Mbase-MOptV2))
NormInf(Mbase-MOptV2)=6.938894e-18
```

We can show sparsity of the Mass matrix :

```
>>> from pyOptFEM.FEM2D import *
>>> Th=SquareMesh(20)
>>> M=MassAssembling2DP1base(Th.nq,Th.nme,Th.me,Th.areas)
>>> showSparsity(M)
```

Note: source code

pyOptFEM.FEM2D.assembly.**MassAssembling2DP1base**(*nq*, *nme*, *me*, *areas*)
Assembly of the Mass Matrix by P_1-Lagrange finite elements using base version (see report).

pyOptFEM.FEM2D.assembly.**MassAssembling2DP1OptV1**(*nq*, *nme*, *me*, *areas*)
Assembly of the Mass Matrix by P_1-Lagrange finite elements using OptV1 version (see report).

pyOptFEM.FEM2D.assembly.**MassAssembling2DP1OptV2**(*nq*, *nme*, *me*, *areas*)
Assembly of the Mass Matrix by P_1-Lagrange finite elements using OptV2 version (see report).

4.1.2 Stiffness Matrix

Assembly of the Stiffness Matrix by P_1-Lagrange finite elements using base, OptV1 and OptV2 versions respectively (see report). The Stiffness Matrix $\mathbb{S}$ is given by

$$\mathbb{S}_{i,j} = \int_{\Omega_h} \nabla\varphi_j(q).\nabla\varphi_i(q)dq, \quad \forall (i,j) \in \{1,...,\mathrm{n_q}\}^2.$$

Note: generic syntax:

```
M = StiffAssembling2DP1<version>(nq,nme,q,me,areas)
```

- nq: total number of nodes of the mesh, also denoted by $\mathrm{n_q}$,
- nme: total number of triangles, also denoted by $\mathrm{n_{me}}$,
- q: Array of vertices coordinates, *(nq,2)* array

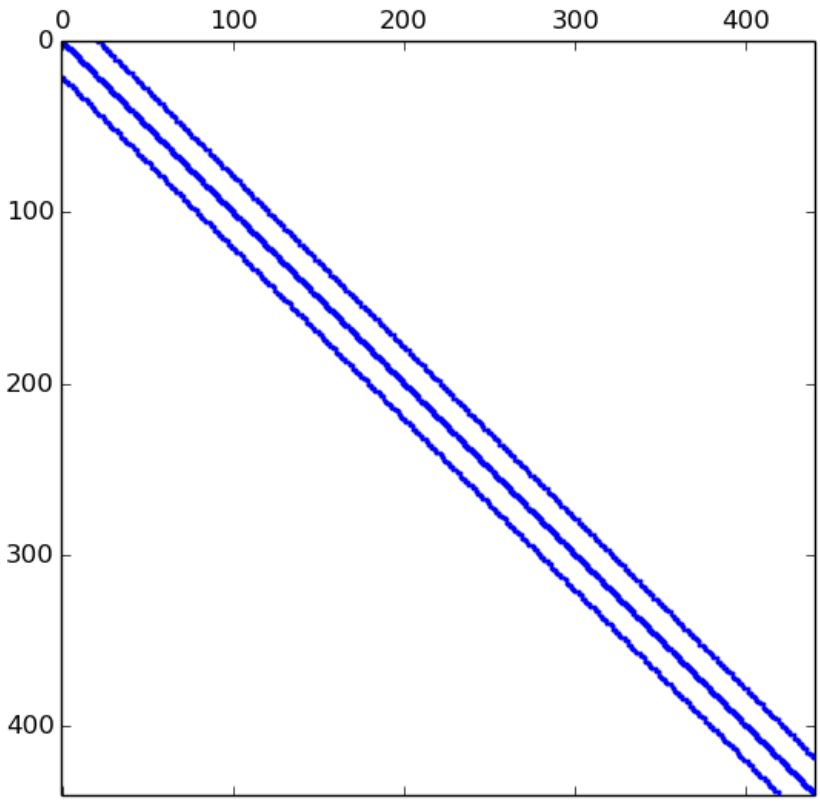


Figure 4.1: Sparsity of Mass Matrix generated with command *showSparsity(M)*

- `me`: Connectivity array, `(nme,3)` array,
- `areas`: Array of areas, `(nme,)` array,
- **returns** a *Scipy* CSC sparse matrix of size $\mathrm{n_q} \times \mathrm{n_q}$

where `<version>` is `base`, `OptV1` or `OptV2`

Benchmarks of theses functions are presented in *Stiffness Matrix*. We give a simple usage :

```
>>> pyOptFEM.FEM2D import *
>>> Th=SquareMesh(5)
>>> Sbase =  StiffAssembling2DP1base(Th.nq,Th.nme,Th.q,Th.me,Th.areas)
>>> SOptV1= StiffAssembling2DP1OptV1(Th.nq,Th.nme,Th.q,Th.me,Th.areas)
>>> print(" NormInf(Sbase-SOptV1)=%e " % NormInf(Sbase-SOptV1))
NormInf(Sbase-SOptV1)=0.000000e+00
>>> SOptV2= StiffAssembling2DP1OptV2(Th.nq,Th.nme,Th.q,Th.me,Th.areas)
>>> print(" NormInf(Sbase-SOptV2)=%e " % NormInf(Sbase-SOptV2))
NormInf(Sbase-SOptV1)=4.440892e-16
```

Note: source code

`pyOptFEM.FEM2D.assembly.`**`StiffAssembling2DP1base`**(*nq*, *nme*, *q*, *me*, *areas*)
Assembly of the Stiffness Matrix by P_1-Lagrange finite elements using `base` version (see report).

`pyOptFEM.FEM2D.assembly.`**`StiffAssembling2DP1OptV1`**(*nq*, *nme*, *q*, *me*, *areas*)
Assembly of the Stiffness Matrix by P_1-Lagrange finite elements using `OptV1` version (see report).

`pyOptFEM.FEM2D.assembly.`**`StiffAssembling2DP1OptV2`**(*nq*, *nme*, *q*, *me*, *areas*)
Assembly of the Stiffness Matrix by P_1-Lagrange finite elements using `OptV2` version (see report).

4.1.3 Elastic Stiffness Matrix

Assembly of the Elastic Stiffness Matrix by P_1-Lagrange finite elements using `base`, `OptV1` and `OptV2` versions respectively (see report). The Elastic Stiffness Matrix $\mathbb{K}$ is given by

$$\mathbb{K}_{m,l} = \int_{\Omega_h} \underline{\boldsymbol{\epsilon}}^t(\psi_l(q))\underline{\boldsymbol{\sigma}}(\psi_m(q)), \quad \forall (m,l) \in \{1,...,2\mathrm{n_q}\}^2.$$

where $\underline{\boldsymbol{\sigma}} = (\sigma_{xx}, \sigma_{yy}, \sigma_{xy})^t$ and $\underline{\boldsymbol{\epsilon}} = (\epsilon_{xx}, \epsilon_{yy}, \epsilon_{xy})^t$ are the elastic stress and strain tensors respectively.

Note: generic syntax:

```
M = StiffElasAssembling2DP1<version>(nq,nme,q,me,areas,la,mu,Num)
```

- `nq`: total number of nodes of the mesh, also denoted by $\mathrm{n_q}$,
- `nme`: total number of triangles, also denoted by $\mathrm{n_{me}}$,
- **`q`: array of vertices coordinates,**
  - `(nq,2)` array for `base` and `OptV1`,
  - `(2,nq)` array for `OptV2` version,
- **`me`: Connectivity array,**
  - `(nme,3)` array for `base` and `OptV1`,

  - `(3,nme)` array for `OptV2` version,
- `areas`: `(nme,)` array of areas,
- `la`: the first Lame coefficient in Hooke's law, denoted by λ,
- `mu`: the second Lame coefficient in Hooke's law, denoted by μ,
- **Num:**
  - `0`: global alternate numbering with local alternate numbering (classical method),
  - `1`: global block numbering with local alternate numbering,
  - `2`: global alternate numbering with local block numbering,
  - `3`: global block numbering with local block numbering.
- **returns** a *Scipy* CSC sparse matrix of size $2n_q \times 2n_q$

where `<version>` is `base`, `OptV1` or `OptV2`

Benchmarks of theses functions are presented in *Elastic Stiffness Matrix*. We give a simple usage :

```
>>> from pyOptFEM.FEM2D import *
>>> Th=SquareMesh(5)
>>> Kbase =  StiffElasAssembling2DP1base(Th.nq,Th.nme,Th.q,Th.me,Th.areas,2,0.5,0)
>>> KOptV1= StiffElasAssembling2DP1OptV1(Th.nq,Th.nme,Th.q,Th.me,Th.areas,2,0.5,0)
>>> print(" NormInf(Kbase-KOptV1)=%e " % NormInf(Kbase-KOptV1))
NormInf(Kbase-KOptV1)=8.881784e-16
>>> KOptV2= StiffElasAssembling2DP1OptV2(Th.nq,Th.nme,Th.q,Th.me,Th.areas,2,0.5,0)
>>> print(" NormInf(Kbase-KOptV2)=%e " % NormInf(Kbase-KOptV2))
NormInf(Kbase-KOptV2)=1.776357e-15
```

We now illustrate the consequences of the choice of the global basis on matrix sparsity

- global *alternate* basis $\mathcal{B}_a$ (`Num=0` or `Num=2`)

  ```
  >>> from pyOptFEM.FEM2D import *
  >>> Th=SquareMesh(15)
  >>> K0=StiffElasAssembling2DP1OptV1(Th.nq,Th.nme,Th.q,Th.me,Th.areas,2,0.5,0)
  >>> showSparsity(K0)

  >>> K3=StiffElasAssembling2DP1OptV1(Th.nq,Th.nme,Th.q,Th.me,Th.areas,2,0.5,3)
  >>> showSparsity(K3)
  ```

- global *block* basis $\mathcal{B}_a$ (`Num=1` or `Num=3`)

Note: source code

`pyOptFEM.FEM2D.assembly.`**`StiffElasAssembling2DP1base`**(*nq*, *nme*, *q*, *me*, *areas*, *la*, *mu*, *Num*)

Assembly of the Elasticity Stiffness Matrix by P_1-Lagrange finite elements using OptV2 version (see report).

`pyOptFEM.FEM2D.assembly.`**`StiffElasAssembling2DP1OptV1`**(*nq*, *nme*, *q*, *me*, *areas*, *la*, *mu*, *Num*)

Assembly of the Elasticity Stiffness Matrix by P_1-Lagrange finite elements using OptV1 version (see report).

`pyOptFEM.FEM2D.assembly.`**`StiffElasAssembling2DP1OptV2`**(*nq*, *nme*, *q*, *me*, *areas*, *la*, *mu*, *Num*)

Assembly of the Elasticity Stiffness Matrix by P_1-Lagrange finite elements using OptV2 version (see report).

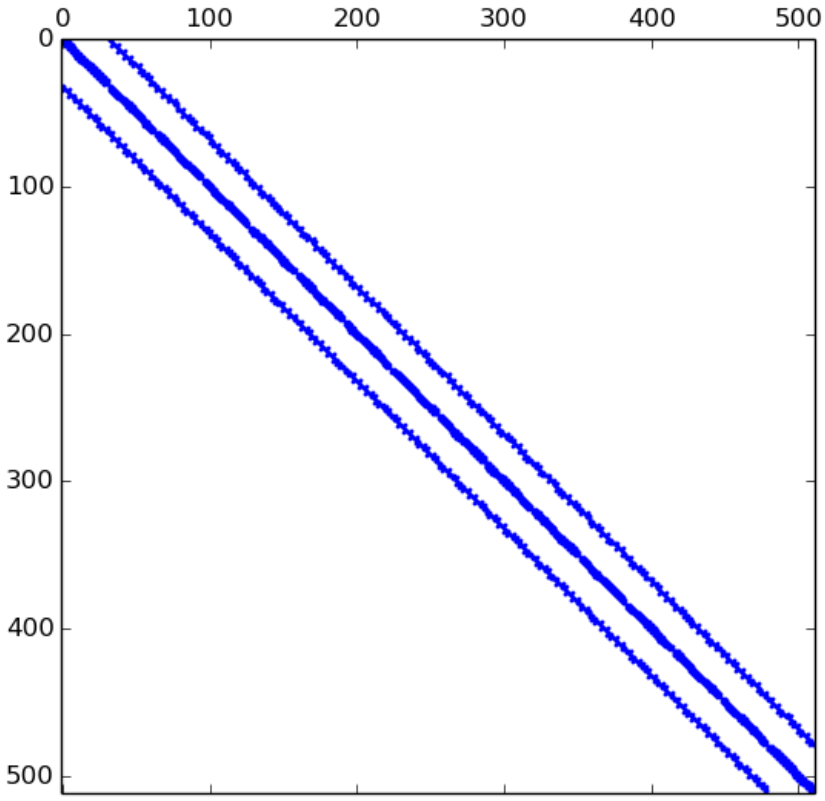


Figure 4.2: Sparsity of the Elastic Stiffness Matrix generated with global alternate numbering (Num=0 or 2)

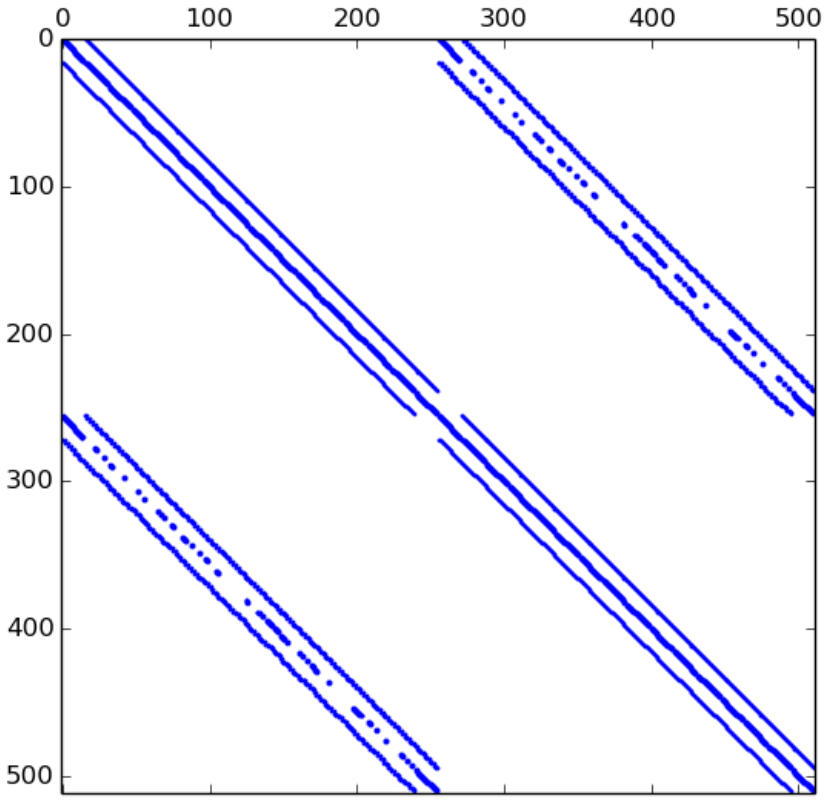


Figure 4.3: Sparsity of the Elastic Stiffness Matrix generated with global block numbering (Num=1 or 3)

4.2 Element matrices (used by `base` and `OptV1` versions)

Let T be a triangle, of area $|T|$ and with $\boldsymbol{q}^1$, $\boldsymbol{q}^2$ and $\boldsymbol{q}^3$ its three vertices. We denote by $\tilde{\varphi}_1$, $\tilde{\varphi}_2$ and $\tilde{\varphi}_3$ the P_1-Lagrange local basis functions such that $\tilde{\varphi}_i(\boldsymbol{q}^j) = \delta_{i,j},\ \forall (i,j) \in \{1,2,3\}^2$.

We also define the local *alternate* basis $\tilde{\mathcal{B}}_a$ by

$$\tilde{\mathcal{B}}_a = \{\tilde{\boldsymbol{\psi}}_1, \ldots, \tilde{\boldsymbol{\psi}}_6\} = \left\{ \begin{pmatrix} \tilde{\varphi}_1 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ \tilde{\varphi}_1 \end{pmatrix}, \begin{pmatrix} \tilde{\varphi}_2 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ \tilde{\varphi}_2 \end{pmatrix}, \begin{pmatrix} \tilde{\varphi}_3 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ \tilde{\varphi}_3 \end{pmatrix} \right\}$$

and the local *block* basis $\tilde{\mathcal{B}}_b$ by

$$\tilde{\mathcal{B}}_b = \{\tilde{\boldsymbol{\phi}}_1, \ldots, \tilde{\boldsymbol{\phi}}_6\} = \left\{ \begin{pmatrix} \tilde{\varphi}_1 \\ 0 \end{pmatrix}, \begin{pmatrix} \tilde{\varphi}_2 \\ 0 \end{pmatrix}, \begin{pmatrix} \tilde{\varphi}_3 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ \tilde{\varphi}_1 \end{pmatrix}, \begin{pmatrix} 0 \\ \tilde{\varphi}_2 \end{pmatrix}, \begin{pmatrix} 0 \\ \tilde{\varphi}_3 \end{pmatrix} \right\}.$$

The **elasticity tensor**, $\mathbb{H}$, obtained from Hooke's law with an isotropic material, defined with the Lamé parameters λ and μ is given by

$$\mathbb{H} = \begin{pmatrix} \lambda + 2\mu & \lambda & 0 \\ \lambda & \lambda + 2\mu & 0 \\ 0 & 0 & \mu \end{pmatrix}$$

and, for a function $\boldsymbol{u} = (u_1, u_2)$ the strain tensors is given by

$$\underline{\boldsymbol{\epsilon}}(\boldsymbol{u}) = \left(\frac{\partial u_1}{\partial x}, \quad \frac{\partial u_2}{\partial y}, \frac{\partial u_2}{\partial x} + \frac{\partial u_1}{\partial y} \right)^t$$

4.2.1 Element Mass Matrix

The element Mass matrix $\mathbb{M}^e(T)$ for the triangle T is defined by

$$\mathbb{M}^e_{i,j}(T) = \int_T \tilde{\varphi}_i(\boldsymbol{q}) \tilde{\varphi}_j(\boldsymbol{q})\, d\boldsymbol{q}, \ \ \forall (i,j) \in \{1,2,3\}^2$$

We obtain :

$$\mathbb{M}^e(T) = \frac{|T|}{12} \begin{pmatrix} 2 & 1 & 1 \\ 1 & 2 & 1 \\ 1 & 1 & 2 \end{pmatrix} \tag{4.1}$$

Note:

`pyOptFEM.FEM2D.elemMatrix.`**`ElemMassMat2DP1`**`(`*area*`)`

Computes the element mass matrix $\mathbb{M}^e(T)$ for a given triangle T of area $|T|$

Parameters **area** (*float*) – area of the triangle.

Returns 3×3 *numpy* array of floats.

4.2.2 Element Stiffness Matrix

The element stiffness matrix, $\mathbb{S}^e(T)$, for the T is defined by

$$\mathbb{S}^e_{i,j}(T) = \int_T < \nabla\tilde{\varphi}_i(\boldsymbol{q}), \nabla\tilde{\varphi}_j(\boldsymbol{q}) > \; d\boldsymbol{q}, \;\; (i,j) \in \{1,2,3\}^2$$

We have :

$$\mathbb{S}^e(T) = \frac{1}{4|T|}\begin{pmatrix} <\boldsymbol{u},\boldsymbol{u}> & <\boldsymbol{u},\boldsymbol{v}> & <\boldsymbol{u},\boldsymbol{w}> \\ <\boldsymbol{v},\boldsymbol{u}> & <\boldsymbol{v},\boldsymbol{v}> & <\boldsymbol{v},\boldsymbol{w}> \\ <\boldsymbol{w},\boldsymbol{u}> & <\boldsymbol{w},\boldsymbol{v}> & <\boldsymbol{w},\boldsymbol{w}> \end{pmatrix}.$$

where $\boldsymbol{u} = \boldsymbol{q}^2 - \boldsymbol{q}^3$, $\boldsymbol{v} = \boldsymbol{q}^3 - \boldsymbol{q}^1$ and $\boldsymbol{w} = \boldsymbol{q}^1 - \boldsymbol{q}^2$.

Note:

`pyOptFEM.FEM2D.elemMatrix.`**`ElemStiffMat2DP1`**(*q1*, *q2*, *q3*, *area*)

Computes the element stiffness matrix $\mathbb{S}^e(T)$ for a given triangle T

Parameters

- **q1,q2,q3** (2×1 *numpy* array) – the three vertices of the triangle,
- **area** (*float*) – area of the triangle.

Returns

Type 3×3 *numpy* array of floats.

4.2.3 Element Elastic Stiffness Matrix

Let $\boldsymbol{u} = \boldsymbol{q}^2 - \boldsymbol{q}^3$, $\boldsymbol{v} = \boldsymbol{q}^3 - \boldsymbol{q}^1$ and $\boldsymbol{w} = \boldsymbol{q}^1 - \boldsymbol{q}^2$.

- The element elastic stiffness matrix, $\mathbb{K}^e(T)$, for a given triangle T in the local *alternate* basis $\tilde{\mathcal{B}}_a$ is defined by

$$\mathbb{K}^e_{i,j}(T) = \int_T \underline{\boldsymbol{\epsilon}}(\tilde{\boldsymbol{\psi}}_j)^t(\boldsymbol{q})\mathbb{H}\underline{\boldsymbol{\epsilon}}(\tilde{\boldsymbol{\psi}}_i)(\boldsymbol{q})\, d\boldsymbol{q}, \quad \forall (i,j) \in \{1,\ldots,6\}^2.$$

We also have

$$\mathbb{K}^e(T) = \frac{1}{4|T|}\mathbb{B}^t\mathbb{H}\mathbb{B}$$

where $\mathbb{H}$ is the elasticity tensor and

$$\mathbb{B} = \begin{pmatrix} u_2 & 0 & v_2 & 0 & w_2 & 0 \\ 0 & -u_1 & 0 & -v_1 & 0 & -w_1 \\ -u_1 & u_2 & -v_1 & v_2 & -w_1 & w_2 \end{pmatrix}$$

Note:

pyOptFEM.FEM2D.elemMatrix.**ElemStiffElasMat2DP1Ba**(*ql*, *area*, *H*)

Returns the element elastic stiffness matrix $\mathbb{K}^e(T)$ for a given triangle T in the local *alternate* basis $\mathcal{B}_a$

Parameters

- **ql** (3×2 *numpy* array) – contains the three vertices of the triangle : `ql[0]`, `ql[1]` and `ql[2]`,
- **area** (*float*) – area of the triangle ,
- **H** (3×3 *numpy* array) – Elasticity tensor, $\mathbb{H}$.

Returns $\mathbb{K}^e(T)$ in $\mathcal{B}_a$ basis.

Type 6×6 *numpy* array of floats.

- The element elastic stiffness matrix, $\mathbb{K}^e(T)$, for a given triangle T in the local *block* basis $\tilde{\mathcal{B}}_b$ is defined by

$$\mathbb{K}^e_{i,j}(T) = \int_T \underline{\boldsymbol{\epsilon}}(\tilde{\boldsymbol{\phi}}_j)^t(\boldsymbol{q})\mathbb{H}\underline{\boldsymbol{\epsilon}}(\tilde{\boldsymbol{\phi}}_i)(\boldsymbol{q})\, d\boldsymbol{q}, \quad \forall (i,j) \in \{1,\dots,6\}^2.$$

We also have

$$\mathbb{K}^e(T) = \frac{1}{4|T|}\mathbb{B}^t\mathbb{H}\mathbb{B}$$

where $\mathbb{H}$ is the elasticity tensor and

$$\mathbb{B} = \begin{pmatrix} u_2 & v_2 & w_2 & 0 & 0 & 0 \\ 0 & 0 & 0 & -u_1 & -v_1 & -w_1 \\ -u_1 & -v_1 & -w_1 & u_2 & v_2 & w_2 \end{pmatrix}$$

Note:

pyOptFEM.FEM2D.elemMatrix.**ElemStiffElasMat2DP1Bb**(*ql*, *area*, *H*)

Returns the element elastic stiffness matrix $\mathbb{K}^e(T)$ for a given triangle T in the local *block* basis $\mathcal{B}_b$

Parameters

- **ql** (3×2 *numpy* array) – contains the three vertices of the triangle : `ql[0]`, `ql[1]` and `ql[2]`,
- **area** (*float*) – area of the triangle,
- **H** (3×3 *numpy* array) – Elasticity tensor, $\mathbb{H}$.

Returns $\mathbb{K}^e(T)$ in $\mathcal{B}_b$ basis.

Type 6×6 *numpy* array of floats

4.3 Vectorized tools (used by `OptV2` version)

4.3.1 Vectorized computation of basis functions gradients

By construction, the gradients of basis functions are constants on each element T_k. So, we denote, $\forall \alpha \in \{1, \ldots, 3\}$, by $\boldsymbol{G}^\alpha$ the $2 \times \mathrm{n_{me}}$ array defined, $\forall k \in \{1, \ldots, \mathrm{n_{me}}\}$, by

$$\boldsymbol{G}^\alpha(:, k) = \nabla \varphi_{\mathrm{me}(\alpha,k)}(\mathrm{q}), \ \forall \mathrm{q} \in T_k.$$

On a triangle T_k, we define $\boldsymbol{D}^{12} = \mathrm{q}^{\mathrm{me}(1,k)} - \mathrm{q}^{\mathrm{me}(2,k)}$, $\boldsymbol{D}^{13} = \mathrm{q}^{\mathrm{me}(1,k)} - \mathrm{q}^{\mathrm{me}(3,k)}$ and $\boldsymbol{D}^{23} = \mathrm{q}^{\mathrm{me}(2,k)} - \mathrm{q}^{\mathrm{me}(3,k)}$. Then, we have

$$\nabla \varphi_1^k(\mathrm{q}) = \frac{1}{2|T_k|} \begin{pmatrix} \boldsymbol{D}_y^{23} \\ -\boldsymbol{D}_x^{23} \end{pmatrix}, \ \nabla \varphi_2^k(\mathrm{q}) = \frac{1}{2|T_k|} \begin{pmatrix} -\boldsymbol{D}_y^{13} \\ \boldsymbol{D}_x^{13} \end{pmatrix}, \ \nabla \varphi_3^k(\mathrm{q}) = \frac{1}{2|T_k|} \begin{pmatrix} \boldsymbol{D}_y^{12} \\ -\boldsymbol{D}_x^{12} \end{pmatrix}.$$

With these formulas, we obtain the vectorized algorithm given in Algorithm 4.4.

Algorithm 4.4

Input :

q : array of vertices coordinates $(2 \times \mathrm{n_q})$
me : connectivity array $(3 \times \mathrm{n_{me}})$
areas : array of mesh elements areas $(1 \times \mathrm{n_{me}})$

Output :

$\boldsymbol{G}^1, \boldsymbol{G}^2, \boldsymbol{G}^3$: gradients arrays $(2 \times \mathrm{n_{me}})$
$\boldsymbol{G}^\alpha(:, k) = \nabla \varphi_\alpha^k(\mathrm{q}), \ \forall \alpha \in \{1, \ldots, 3\}$

Function: $[\boldsymbol{G}^1, \boldsymbol{G}^2, \boldsymbol{G}^3] \leftarrow$ GradientVec2D$(\mathrm{q}, \mathrm{me}, \mathrm{areas})$

$\boldsymbol{D}^{12} \leftarrow \mathrm{q}(:, \mathrm{me}(1,:)) - \mathrm{q}(:, \mathrm{me}(2,:))$ ▷ $2 \times \mathrm{n_{me}}$ array
$\boldsymbol{D}^{13} \leftarrow \mathrm{q}(:, \mathrm{me}(1,:)) - \mathrm{q}(:, \mathrm{me}(3,:))$ ▷ $2 \times \mathrm{n_{me}}$ array
$\boldsymbol{D}^{23} \leftarrow \mathrm{q}(:, \mathrm{me}(2,:)) - \mathrm{q}(:, \mathrm{me}(3,:))$ ▷ $2 \times \mathrm{n_{me}}$ array
$\boldsymbol{G}^1 \leftarrow \begin{pmatrix} \boldsymbol{D}^{23}(2,:)./(2 * \mathrm{areas}) \\ -\boldsymbol{D}^{23}(1,:)./(2 * \mathrm{areas}) \end{pmatrix}$ ▷ $2 \times \mathrm{n_{me}}$ array
$\boldsymbol{G}^2 \leftarrow \begin{pmatrix} -\boldsymbol{D}^{13}(2,:)./(2 * \mathrm{areas}) \\ \boldsymbol{D}^{13}(1,:)./(2 * \mathrm{areas}) \end{pmatrix}$ ▷ $2 \times \mathrm{n_{me}}$ array
$\boldsymbol{G}^3 \leftarrow \begin{pmatrix} \boldsymbol{D}^{12}(2,:)./(2 * \mathrm{areas}) \\ -\boldsymbol{D}^{12}(1,:)./(2 * \mathrm{areas}) \end{pmatrix}$ ▷ $2 \times \mathrm{n_{me}}$ array

end Function:

Figure 4.4: Vectorized algorithm for computation of basis functions gradients in 2d

4.4 Vectorized element matrices (used by `OptV2` version)

4.4.1 Element Mass Matrix

We have

$$\mathbb{M}^e(T_k) = \frac{|T_k|}{12}\begin{pmatrix} 2 & 1 & 1 \\ 1 & 2 & 1 \\ 1 & 1 & 2 \end{pmatrix}$$

Then with $\mathbb{K}_g$ definition (see Section *New Optimized assembly algorithm (OptV2 version)*) , we obtain

$$\mathbb{K}_g(3(i-1)+j,k) = |T_k|\frac{1+\delta_{i,j}}{12} \quad 1 \leq i,j \leq 3,$$

We represent in figure 4.5 the corresponding row-wise operations.

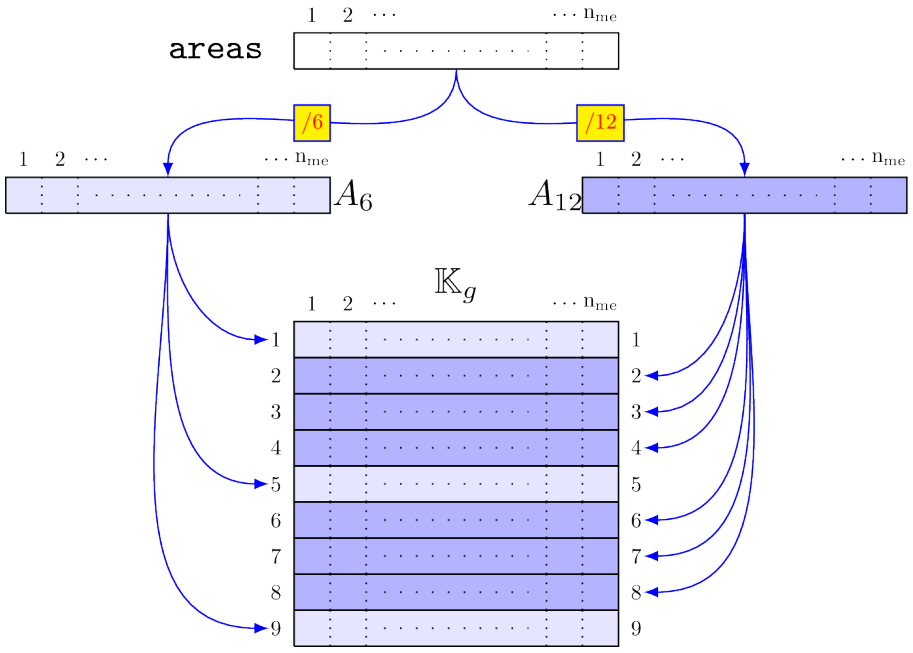


Figure 4.5: Construction of $\mathbb{K}_g$ associated to 2d **Mass** matrix in

So the vectorized algorithm for $\mathbb{K}_g$ computation is simple and given in Algorithm 4.6.

Algorithm 4.6

Note:

`pyOptFEM.FEM2D.elemMatrixVec.`**`ElemMassMat2DP1Vec`**(*areas*)

Computes all the element Mass matrices $\mathbb{M}^e(T_k)$ for $k \in \{0, \ldots, \mathrm{n_{me}} - 1\}$

Parameters **areas** ($\mathrm{n_{me}}$ *numpy* array of floats) – areas of all the mesh elements.

Algorithm 1 $\mathbb{K}_g$ associated to Mass matrix in 2d

Function: $\mathbb{K}_g \leftarrow$ ElemMassMat2DP1Vec$(\mathrm{n_{me}}, \mathrm{areas})$
$\mathbb{K}_g \leftarrow \mathbb{O}_{9\times \mathrm{n_{me}}}$ ▷ $9 \times \mathrm{n_{me}}$ array of zeros
$\mathbb{K}_g(1,:) \leftarrow \mathbb{K}_g(5,:) \leftarrow \mathbb{K}_g(9,:) \leftarrow \mathrm{areas}/12$
$\mathbb{K}_g(2,:) \leftarrow \mathbb{K}_g(3,:) \leftarrow \mathbb{K}_g(4,:) \leftarrow \mathbb{K}_g(6,:) \leftarrow \mathbb{K}_g(7,:) \leftarrow \mathbb{K}_g(8,:) \leftarrow \mathrm{areas}/6$
end Function:

Figure 4.6: Vectorized algorithm for $\mathbb{K}_g$ associated to 2d **Mass** matrix

Returns a one dimensional *numpy* array of size $9\mathrm{n_{me}}$

4.4.2 Element Stiffness Matrix

We have $\forall(\alpha,\beta) \in \{1,\ldots,3\}^2$

$$\mathbb{S}^e_{\alpha,\beta}(T_k) = |T_k| \left\langle \nabla \varphi^k_\beta, \nabla \varphi^k_\alpha \right\rangle .$$

Using vectorized algorithm function GradientVec2D given in Algorithm 4.4, we obtain the vectorized algorithm 4.7 for $\mathbb{K}_g$ computation for the **Stiffness** matrix in 2d.

Algorithm 4.7

Function: $\mathbb{K}_g \leftarrow$ ElemStiffMat2DP1Vec$(\mathrm{n_{me}}, \mathrm{q}, \mathrm{me}, \mathrm{areas})$
  $[\boldsymbol{G}^1, \boldsymbol{G}^2, \boldsymbol{G}^3] \leftarrow$ GradientVec2D$(\mathrm{me}, \mathrm{areas})$
  $\mathbb{K}_g \leftarrow \mathbb{O}_{9\times \mathrm{n_{me}}}$ ▷ $9 \times \mathrm{n_{me}}$ array of zeros
  $\mathbb{K}_g(1,:) \leftarrow \mathrm{areas}. * \mathrm{sum}(\boldsymbol{G}^1 . * \boldsymbol{G}^1, 1)$
  $\mathbb{K}_g(2,:) \leftarrow \mathbb{K}_g(2,:) \leftarrow \mathrm{areas}. * \mathrm{sum}(\boldsymbol{G}^1 . * \boldsymbol{G}^2, 1)$
  $\mathbb{K}_g(3,:) \leftarrow \mathbb{K}_g(7,:) \leftarrow \mathrm{areas}. * \mathrm{sum}(\boldsymbol{G}^1 . * \boldsymbol{G}^3, 1)$
  $\mathbb{K}_g(5,:) \leftarrow \mathrm{areas}. * \mathrm{sum}(\boldsymbol{G}^2 . * \boldsymbol{G}^2, 1)$
  $\mathbb{K}_g(6,:) \leftarrow \mathbb{K}_g(8,:) \leftarrow \mathrm{areas}. * \mathrm{sum}(\boldsymbol{G}^2 . * \boldsymbol{G}^3, 1)$
  $\mathbb{K}_g(9,:) \leftarrow \mathrm{areas}. * \mathrm{sum}(\boldsymbol{G}^3 . * \boldsymbol{G}^3, 1)$
end Function:

Figure 4.7: Vectorized algorithm for $\mathbb{K}_g$ associated to 2d **Stiffness** matrix

Note:

pyOptFEM.FEM2D.elemMatrixVec.**ElemStiffMat2DP1Vec**(*nme*, *q*, *me*, *areas*)
Computes all the element stiffness matrices $\mathbb{S}^e(T_k)$ for $k \in \{0,\ldots,\mathrm{n_{me}}-1\}$

Parameters

- **nme** (*int*) – number of mesh elements,
- **q** ($\mathrm{n_q} \times 2$ *numpy* array of floats) – mesh vertices,
- **me** ($\mathrm{n_{me}} \times 3$ *numpy* array of integers) – mesh connectivity,

- **areas** ($\mathrm{n_{me}}$ *numpy* array of floats) – areas of all the mesh elements.

Returns a one dimensional *numpy* array of size $9\mathrm{n_{me}}$

4.4.3 Element Elastic Stiffness Matrix

- We define on T_k the local *alternate* basis $\mathcal{B}_a^k$ by

$$\mathcal{B}_a^k = \{\boldsymbol{\psi}_1^k, \ldots, \boldsymbol{\psi}_6^k\} = \left\{ \begin{pmatrix} \varphi_1^k \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ \varphi_1^k \end{pmatrix}, \begin{pmatrix} \varphi_2^k \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ \varphi_2^k \end{pmatrix}, \begin{pmatrix} \varphi_3^k \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ \varphi_3^k \end{pmatrix} \right\}$$

where $\varphi_\alpha^k = \varphi_{\mathrm{me}(\alpha,k)}$. With notations of *Presentation*, we have $\forall(\alpha,\beta) \in \{1,\ldots,6\}^2$

$$\mathbb{K}^e_{\alpha,\beta}(T_k) = \int_{T_k} \underline{\boldsymbol{\epsilon}}^t(\boldsymbol{\psi}_\beta^k)\mathbb{C}\underline{\boldsymbol{\epsilon}}(\boldsymbol{\psi}_\alpha^k)d\mathrm{q} = \int_{T_k} \mathcal{H}(\boldsymbol{\psi}_\beta^k, \boldsymbol{\psi}_\alpha^k)(\mathrm{q})d\mathrm{q}$$

with, $\forall \boldsymbol{u} = (u_1, u_2) \in H^1(\Omega)^2$, $\forall \boldsymbol{v} = (v_1, v_2) \in H^1(\Omega)^2$,

$$\begin{array}{rcl} \mathcal{H}(\boldsymbol{u},\boldsymbol{v}) & = & \left\langle \begin{pmatrix} \gamma & 0 \\ 0 & \mu \end{pmatrix} \nabla u_1, \nabla v_1 \right\rangle + \left\langle \begin{pmatrix} 0 & \lambda \\ \mu & 0 \end{pmatrix} \nabla u_2, \nabla v_1 \right\rangle \\ & + & \left\langle \begin{pmatrix} 0 & \mu \\ \lambda & 0 \end{pmatrix} \nabla u_1, \nabla v_2 \right\rangle + \left\langle \begin{pmatrix} \mu & 0 \\ 0 & \gamma \end{pmatrix} \nabla u_2, \nabla v_2 \right\rangle \end{array} \tag{4.2}$$

For example, we can explicitly compute the first two terms in the first column of $\mathbb{K}^e(T_k)$ which are given by

$$\begin{array}{rcl} \mathbb{K}^e_{1,1}(T_k) & = & \int_{T_k} \mathcal{H}(\boldsymbol{\psi}_1^k, \boldsymbol{\psi}_1^k)(\mathrm{q})d\mathrm{q} \\ & = & \int_{T_k} \mathcal{H}\left(\begin{pmatrix} \varphi_1^k \\ 0 \end{pmatrix}, \begin{pmatrix} \varphi_1^k \\ 0 \end{pmatrix} \right)(\mathrm{q})d\mathrm{q} \\ & = & |T_k| \left\langle \begin{pmatrix} \gamma & 0 \\ 0 & \mu \end{pmatrix} \nabla \varphi_1^k, \nabla \varphi_1^k \right\rangle = |T_k| \left(\gamma \frac{\partial \varphi_1^k}{\partial x} \frac{\partial \varphi_1^k}{\partial x} + \mu \frac{\partial \varphi_1^k}{\partial y} \frac{\partial \varphi_1^k}{\partial y} \right). \end{array}$$

and

$$\begin{array}{rcl} \mathbb{K}^e_{2,1}(T_k) & = & \int_{T_k} \mathcal{H}(\boldsymbol{\psi}_1^k, \boldsymbol{\psi}_2^k)(\mathrm{q})d\mathrm{q} \\ & = & \int_{T_k} \mathcal{H}\left(\begin{pmatrix} \varphi_1^k \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ \varphi_1^k \end{pmatrix} \right)(\mathrm{q})d\mathrm{q} \\ & = & |T_k| \left\langle \begin{pmatrix} 0 & \mu \\ \lambda & 0 \end{pmatrix} \nabla \varphi_1^k, \nabla \varphi_1^k \right\rangle = |T_k|(\lambda+\mu) \frac{\partial \varphi_1^k}{\partial x} \frac{\partial \varphi_1^k}{\partial y}. \end{array}$$

Using vectorized algorithm function GradientVec2D given in Algorithm 4.4, we obtain the vectorized algorithm 4.7 for $\mathbb{K}_g$ computation for the **Elastic Stiffness** matrix in 2d.

Algorithm 4.8

Note:

`pyOptFEM.FEM2D.elemMatrixVec.`**`ElemStiffElasMatBaVec2DP1`**`(`*nme*, *q*, *me*, *areas*, *L*, *M*, ***kwargs*`)`

Computes all the element elastic stiffness matrices $\mathbb{K}^e(T_k)$ for $k \in \{0,\ldots,\mathrm{n_{me}}-1\}$ in local *alternate* basis.

Function: $\mathbb{K}_g \leftarrow$ ElemStiffElasMatBaVec2DP1$(\ \mathrm{n_{me}}, \mathrm{q}, \mathrm{me}, \mathrm{areas}, \lambda, \mu)$

$[\boldsymbol{G}^1, \boldsymbol{G}^2, \boldsymbol{G}^3] \leftarrow$ GradientVec2D$(\mathrm{q}, \mathrm{me}, \mathrm{areas})$

$\mathbb{K}_g \leftarrow \mathbb{O}_{36 \times \mathrm{n_{me}}}$

$\mathbb{K}_g(1,:) \leftarrow ((\lambda + 2\mu) * \boldsymbol{G}^1(1,:). * \boldsymbol{G}^1(1,:) + \mu \boldsymbol{G}^1(2,:).\boldsymbol{G}^1(2,:)). * \mathrm{areas}$

$\mathbb{K}_g(2,:) \leftarrow (\lambda + \mu) * \boldsymbol{G}^1(1,:). * \boldsymbol{G}^1(2,:). * \mathrm{areas}$

$\vdots$

$\mathbb{K}_g(30,:) \leftarrow (\lambda + \mu) * \boldsymbol{G}^3(1,:). * \boldsymbol{G}^3(2,:). * \mathrm{areas}$

$\mathbb{K}_g(36,:) \leftarrow ((\lambda + 2\mu) * \boldsymbol{G}^3(2,:). * \boldsymbol{G}^3(2,:) + \mu \boldsymbol{G}^3(1,:). * \boldsymbol{G}^3(1,:)). * \mathrm{areas}$

$\mathbb{K}_g([7, 13, 14, 19, 20, 21, 25, 26, 27, 28, 31, 32, 33, 34, 35], :)$
$\leftarrow \mathbb{K}_g([2, 3, 9, 4, 10, 16, 5, 11, 17, 23, 6, 12, 18, 24, 30], :)$

end Function:

Figure 4.8: Vectorized algorithm for $\mathbb{K}_g$ associated to 2d **Elastic Stiffness** matrix

Parameters

- **nme** (*int*) – number of mesh elements,
- **q** (`(2,nq)` *numpy* array of floats) – mesh vertices,
- **me** (`(3,nme)` *numpy* array of integers) – mesh connectivity,
- **areas** (`(nme,)` *numpy* array of floats) – areas of all the mesh elements.
- **L** (*float*) – the λ Lame parameter,
- **M** (*float*) – the μ Lame parameter.

Returns a `(36*nme,)` *numpy* array of floats.

- We define on T_k the local *block* basis $\mathcal{B}_b^k$ by

$$\mathcal{B}_b^k = \{\boldsymbol{\phi}_1^k, \ldots, \boldsymbol{\phi}_6^k\} = \left\{ \begin{pmatrix} \varphi_1^k \\ 0 \end{pmatrix}, \begin{pmatrix} \varphi_2^k \\ 0 \end{pmatrix}, \begin{pmatrix} \varphi_3^k \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ \varphi_1^k \end{pmatrix}, \begin{pmatrix} 0 \\ \varphi_2^k \end{pmatrix}, \begin{pmatrix} 0 \\ \varphi_3^k \end{pmatrix} \right\}$$

where $\varphi_\alpha^k = \varphi_{\mathrm{me}(\alpha,k)}$.

For example, using formula (4.2), we can explicitly compute the first two terms in the first column of $\mathbb{K}^e(T_k)$ which are given by

$$\begin{aligned} \mathbb{K}_{1,1}^e(T_k) &= \int_{T_k} \mathcal{H}(\boldsymbol{\phi}_1^k, \boldsymbol{\phi}_1^k)(\mathrm{q}) d\mathrm{q} \\ &= \int_{T_k} \mathcal{H}\left(\begin{pmatrix} \varphi_1^k \\ 0 \end{pmatrix}, \begin{pmatrix} \varphi_1^k \\ 0 \end{pmatrix} \right)(\mathrm{q}) d\mathrm{q} \\ &= |T_k| \left\langle \begin{pmatrix} \gamma & 0 \\ 0 & \mu \end{pmatrix} \nabla \varphi_1^k, \nabla \varphi_1^k \right\rangle = |T_k| \left(\gamma \frac{\partial \varphi_1^k}{\partial x} \frac{\partial \varphi_1^k}{\partial x} + \mu \frac{\partial \varphi_1^k}{\partial y} \frac{\partial \varphi_1^k}{\partial y} \right). \end{aligned}$$

and

$$\begin{aligned}\mathbb{K}^e_{2,1}(T_k) &= \int_{T_k} \mathcal{H}(\boldsymbol{\phi}_1^k, \boldsymbol{\phi}_2^k)(\mathrm{q})d\mathrm{q} \\ &= \int_{T_k} \mathcal{H}\left(\begin{pmatrix}\varphi_1^k \\ 0\end{pmatrix}, \begin{pmatrix}\varphi_2^k \\ 0\end{pmatrix}\right)(\mathrm{q})d\mathrm{q} \\ &= |T_k| \left\langle \begin{pmatrix}\gamma & 0 \\ 0 & \mu\end{pmatrix} \nabla\varphi_1^k, \nabla\varphi_2^k \right\rangle = |T_k| \left(\gamma \frac{\partial\varphi_1^k}{\partial x}\frac{\partial\varphi_2^k}{\partial x} + \mu \frac{\partial\varphi_1^k}{\partial y}\frac{\partial\varphi_2^k}{\partial y}\right).\end{aligned}$$

Using vectorized algorithm function GradientVec2D given in Algorithm 4.4, we obtain the vectorized algorithm 4.9 for $\mathbb{K}_g$ computation for the **Elastic Stiffness** matrix in 2d.

Algorithm 4.9

Function: $\mathbb{K}_g \leftarrow$ ElemStiffElasMatBbVec2DP1$(\mathrm{n_{me}}, \mathrm{q}, \mathrm{me}, \mathrm{areas}, \lambda, \mu)$

$[\boldsymbol{G}^1, \boldsymbol{G}^2, \boldsymbol{G}^3] \leftarrow$ GradientVec2D$(\mathrm{q}, \mathrm{me}, \mathrm{areas})$

$\mathbb{K}_g \leftarrow \mathbb{O}_{36\times \mathrm{n_{me}}}$

$\mathbb{K}_g(1,:) \leftarrow ((\lambda+2\mu) * \boldsymbol{G}^1(1,:). * \boldsymbol{G}^1(1,:) + \mu \boldsymbol{G}^1(2,:).\boldsymbol{G}^1(2,:)). * \mathrm{areas}$

$\mathbb{K}_g(2,:) \leftarrow ((\lambda+2\mu) * \boldsymbol{G}^1(1,:). * \boldsymbol{G}^2(1,:) + \mu \boldsymbol{G}^1(2,:).\boldsymbol{G}^2(2,:)). * \mathrm{areas}$

$\vdots$

$\mathbb{K}_g(30,:) \leftarrow ((\lambda+\mu) * \boldsymbol{G}^2(2,:). * \boldsymbol{G}^3(2,:) + \mu \boldsymbol{G}^1(1,:).\boldsymbol{G}^3(1,:)). * \mathrm{areas}$

$\mathbb{K}_g(36,:) \leftarrow ((\lambda+2\mu) * \boldsymbol{G}^3(2,:). * \boldsymbol{G}^3(2,:) + \mu \boldsymbol{G}^3(1,:). * \boldsymbol{G}^3(1,:)). * \mathrm{areas}$

$\mathbb{K}_g([7,13,14,19,20,21,25,26,27,28,31,32,33,34,35],:)$
$\leftarrow \mathbb{K}_g([2,3,9,4,10,16,5,11,17,23,6,12,18,24,30],:)$

end Function:

Figure 4.9: Vectorized algorithm for $\mathbb{K}_g$ associated to 2d **Elastic Stiffness** matrix

Note:

pyOptFEM.FEM2D.elemMatrixVec.**ElemStiffElasMatBbVec2DP1**(*nme*, *q*, *me*, *areas*, *L*, *M*, ***kwargs*)

Computes all the element elastic stiffness matrices $\mathbb{K}^e(T_k)$ for $k \in \{0, \ldots, \mathrm{n_{me}} - 1\}$ in local *block* basis.

Parameters

- **nme** (*int*) – number of mesh elements,
- **q** (`(2,nq)` *numpy* array of floats) – mesh vertices,
- **me** (`(3,nme)` *numpy* array of integers) – mesh connectivity,
- **areas** (`(nme,)` *numpy* array of floats) – areas of all the mesh elements.
- **L** (*float*) – the λ Lame parameter,
- **M** (*float*) – the μ Lame parameter.

Returns a `(36*nme,)` *numpy* array of floats.

4.5 Mesh

class pyOptFEM.FEM2D.mesh.**SquareMesh**(*N*, ***kwargs*)

Creates meshes of the unit square $[0,1] \times [0,1]$. Class attributes are :

- **nq**, total number of mesh vertices (points), also denoted $\mathrm{n_q}$.
- **nme**, total number of mesh elements (triangles in 2d),
- **version**, mesh structure version,
- **q**, *Numpy* array of vertices coordinates, dimension `(nq,2)` *(version 0)* or `(2,nq)` *(version 1)*.

  `q[j]` *(version 0)* or `q[:,j]` *(version 1)* are the two coordinates of the j-th vertex, $j \in \{0,..,nq-1\}$
- **me**, *Numpy* connectivity array, dimension `(nme,3)` *(version 0)* or `(3,nme)` *(version 1)*.

  `me[k]` *(version 0)* or `me[:,k]` *(version 1)* are the storage index of the three vertices of the k-th triangle in the array `q` of vertices coordinates, $k \in \{0,...,nme-1\}$.
- **areas**, Array of mesh elements areas, `(nme,)` *Numpy* array.

  `areas[k]` is the area of k-th triangle, `k in range(0,nme)`

  Parameters **N** – number of points on each side of the square

optional parameter : `version=0` or `version=1`

```
>>> from pyOptFEM.FEM2D import *
>>> Th=SquareMesh(3)
>>> Th.nme,Th.nq
(18, 16)
>>> Th.q
array([[ 0.        ,  0.        ],
       [ 0.33333333,  0.        ],
       [ 0.66666667,  0.        ],
       [ 1.        ,  0.        ],
       [ 0.        ,  0.33333333],
       [ 0.33333333,  0.33333333],
       [ 0.66666667,  0.33333333],
       [ 1.        ,  0.33333333],
       [ 0.        ,  0.66666667],
       [ 0.33333333,  0.66666667],
       [ 0.66666667,  0.66666667],
       [ 1.        ,  0.66666667],
       [ 0.        ,  1.        ],
       [ 0.33333333,  1.        ],
       [ 0.66666667,  1.        ],
       [ 1.        ,  1.        ]])
>>> PlotMesh(Th)
```

class pyOptFEM.FEM2D.mesh.**getMesh**(*meshfile*, ***kwargs*)

Reads a *FreeFEM++* mesh from file `meshfile`. Class attributes are :

- **nq**, total number of mesh vertices (points), also denoted $\mathrm{n_q}$.
- **nme**, total number of mesh elements (triangles in 2d),
- **version**, mesh structure version,
- **q**, *Numpy* array of vertices coordinates, dimension `(nq,2)` *(version 0)* or `(2,nq)` *(version 1)*.

  `q[j]` *(version 0)* or `q[:,j]` *(version 1)* are the two coordinates of the j-th vertex, $j \in \{0,..,nq-1\}$

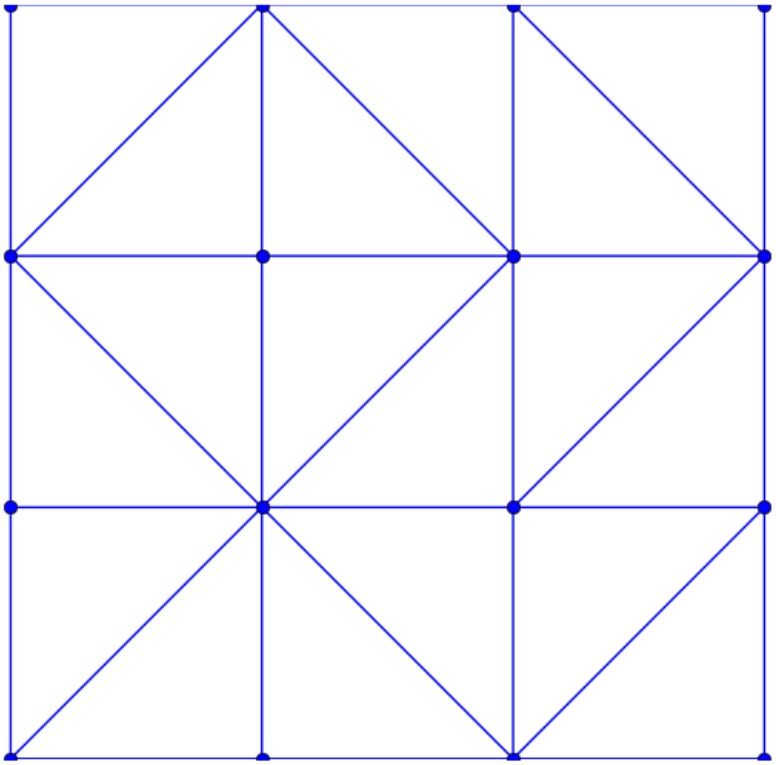


Figure 4.10: SquareMesh(3) visualisation

- **me**, *Numpy* connectivity array, dimension `(nme,3)` *(version 0)* or `(3,nme)` *(version 1)*.

  `me[k]` *(version 0)* or `me[:,k]` *(version 1)* are the storage index of the three vertices of the k-th triangle in the array `q` of vertices coordinates, $k \in \{0, ..., nme - 1\}$.

- **areas**, Array of mesh elements areas, `(nme,)` *Numpy* array.

  `areas[k]` is the area of k-th triangle, `k in range(0,nme)`

  Parameters **N** – number of points on each side of the square

optional parameter : `version=0` or `version=1`

```
>>> from pyOptFEM.FEM2D import *
>>> Th=getMesh('mesh/disk4-1-5.msh')
>>> PlotMesh(Th)
```

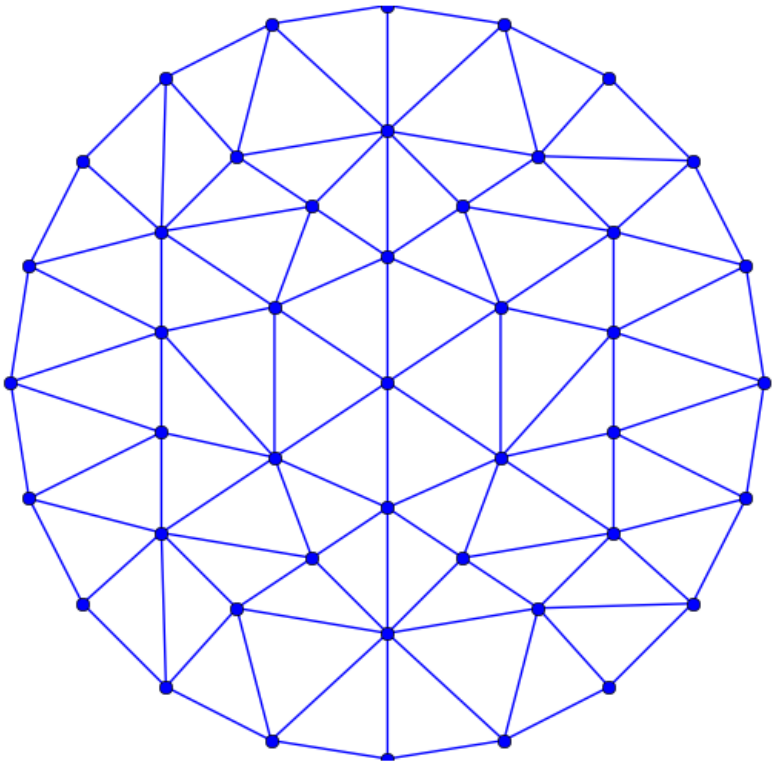


Figure 4.11: Visualisation of a *FreeFEM++* mesh (disk unit)

CHAPTER

FIVE

FEM3D MODULE

Author Francois Cuvelier <cuvelier@math.univ-paris13.fr>

Date 15/09/2013

Contains functions to build some finite element matrices using P_1-Lagrange finite elements on a 3D mesh. Each assembly matrix is computed by three differents versions called `base`, `OptV1` and `OptV2` (see here)

Contents

5.1 Assembly matrix (`base`, `OptV1` and `OptV2` versions)

Let Ω_h be a tetrahedral mesh of Ω corresponding to the following structure data:

name	type	dimension	description	Python
$\mathrm{n_q}$	integer	1	number of vertices	`nq`
$\mathrm{n_{me}}$	integer	1	number of elements	`nme`
q	double	$3 \times \mathrm{n_q}$	array of vertices coordinates. $\mathrm{q}(\nu, j)$ is the ν-th coordinate of the j-th vertex, $\nu \in \{1,2,3\}$, $j \in \{1,\ldots,\mathrm{n_q}\}$. The j-th vertex will be also denoted by q^j	`q` (transposed) `q[j-1]` = q^j
me	integer	$4 \times \mathrm{n_{me}}$	connectivity array. $\mathrm{me}(\beta, k)$ is the storage index of the β-th vertex of the k-th element, in the array q, for $\beta \in \{1,\ldots,4\}$ and $k \in \{1,\ldots,\mathrm{n_{me}}\}$	`me` (transposed)
volumes	double	$1 \times \mathrm{n_{me}}$	array of volumes. $\mathrm{volumes}(k)$ is the k-th tetrahedron volume, $k \in \{1,\ldots,\mathrm{n_{me}}\}$	`volumes`

The P_1-Lagrange basis functions associated to Ω_h are denoted by φ_i for all $i \in \{1,\ldots,\mathrm{n_q}\}$ and are defined by

$$\varphi_i(\mathrm{q^j}) = \delta_{\mathrm{i,j}}, \ \ \forall (\mathrm{i,j}) \in \{1,\ldots,\mathrm{n_q}\}^2$$

We also define the global *alternate* basis $\mathcal{B}_a$ by

$$\mathcal{B}_a = \{\boldsymbol{\psi}_1,\ldots,\boldsymbol{\psi}_{3\mathrm{n_q}}\} = \left\{ \begin{pmatrix}\varphi_1\\0\\0\end{pmatrix}, \begin{pmatrix}0\\\varphi_1\\0\end{pmatrix}, \begin{pmatrix}0\\0\\\varphi_1\end{pmatrix}, \ldots, \begin{pmatrix}\varphi_{\mathrm{n_q}}\\0\\0\end{pmatrix}, \begin{pmatrix}0\\\varphi_{\mathrm{n_q}}\\0\end{pmatrix}, \begin{pmatrix}0\\0\\\varphi_{\mathrm{n_q}}\end{pmatrix} \right\}$$

and the global *block* basis $\mathcal{B}_b$ by

$$\mathcal{B}_b = \{\boldsymbol{\phi}_1,\ldots,\boldsymbol{\phi}_{3\mathrm{n_q}}\} = \left\{ \begin{pmatrix}\varphi_1\\0\\0\end{pmatrix}, \ldots, \begin{pmatrix}\varphi_{\mathrm{n_q}}\\0\\0\end{pmatrix}, \begin{pmatrix}0\\\varphi_1\\0\end{pmatrix}, \ldots, \begin{pmatrix}0\\\varphi_{\mathrm{n_q}}\\0\end{pmatrix}, \begin{pmatrix}0\\0\\\varphi_1\end{pmatrix}, \ldots, \begin{pmatrix}0\\0\\\varphi_{\mathrm{n_q}}\end{pmatrix} \right\}.$$

5.1.1 Mass Matrix

Assembly of the Mass Matrix by P_1-Lagrange finite elements using `base`, `OptV1` and `OptV2` versions respectively (see report). The Mass Matrix $\mathbb{M}$ is given by

$$\mathbb{M}_{i,j} = \int_{\Omega_h} \varphi_j(q)\, \varphi_i(q) dq, \ \ \forall (i,j) \in \{1,...,\mathrm{n_q}\}^2.$$

Note: generic syntax:

```
M = MassAssembling3DP1<version>(nq,nme,me,volumes)
```

- `nq`: total number of nodes of the mesh, also denoted by $\mathrm{n_q}$,
- `nme`: total number of tetrahedra, also denoted by $\mathrm{n_{me}}$,

- `me`: Connectivity array, `(nme,4)` array,
- `volumes`: Array of tetrahedra volumes, `(nme,)` array,
- **returns** a *Scipy* CSC sparse matrix of size $n_q \times n_q$

where `<version>` is `base`, `OptV1` or `OptV2`

```
>>> from pyOptFEM.FEM3D import *
>>> Th=CubeMesh(5)
>>> Mbase =  MassAssembling3DP1base(Th.nq,Th.nme,Th.me,Th.volumes)
>>> MOptV1= MassAssembling3DP1OptV1(Th.nq,Th.nme,Th.me,Th.volumes)
>>> print(" NormInf(Mbase-MOptV1)=%e " % NormInf(Mbase-MOptV1))
NormInf(Mbase-MOptV1)=1.734723e-18
>>> MOptV2= MassAssembling3DP1OptV2(Th.nq,Th.nme,Th.me,Th.volumes)
>>> print(" NormInf(Mbase-MOptV2)=%e " % NormInf(Mbase-MOptV2))
NormInf(Mbase-MOptV2)=1.734723e-18
```

We can show sparsity of the Mass matrix :

```
>>> from pyOptFEM.FEM3D import *
>>> Th=CubeMesh(5)
>>> M=MassAssembling3DP1OptV2(Th.nq,Th.nme,Th.me,Th.areas)
>>> showSparsity(M)
```

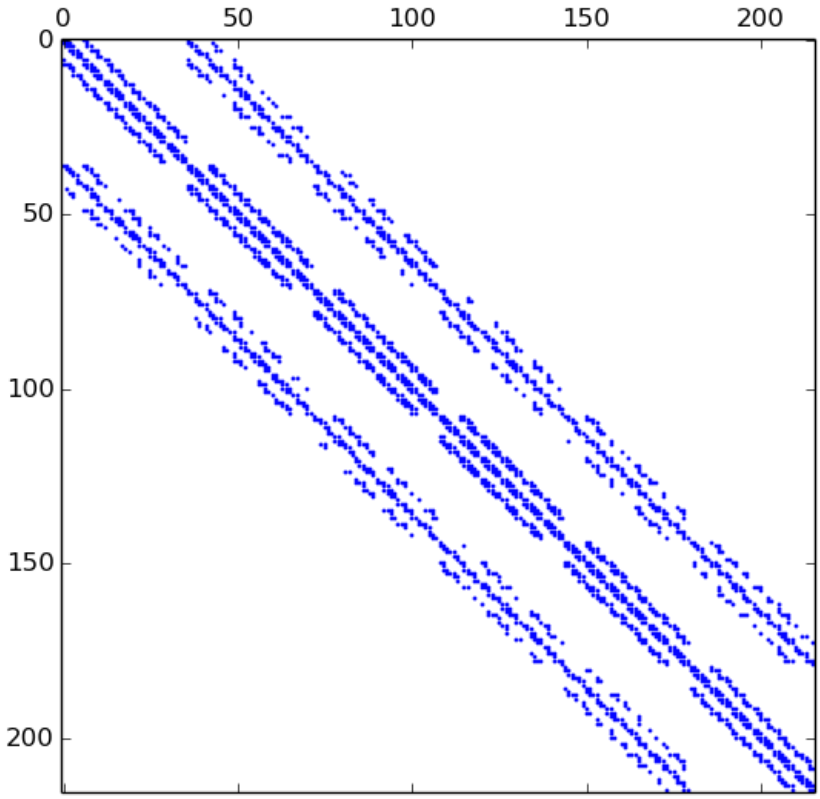


Figure 5.1: Sparsity of Mass Matrix generated with command *showSparsity(M)*

Note: source code

`pyOptFEM.FEM3D.assembly.`**`MassAssembling3DP1base`**(*nq*, *nme*, *me*, *volumes*)
Assembly of the Mass Matrix by P_1-Lagrange finite elements using `base` version (see report).

`pyOptFEM.FEM3D.assembly.`**`MassAssembling3DP1OptV1`**(*nq*, *nme*, *me*, *volumes*)
Assembly of the Mass Matrix by P_1-Lagrange finite elements using `OptV1` version (see report).

`pyOptFEM.FEM3D.assembly.`**`MassAssembling3DP1OptV2`**(*nq*, *nme*, *me*, *volumes*)
Assembly of the Mass Matrix by P_1-Lagrange finite elements using `OptV2` version (see report).

5.1.2 Stiffness Matrix

Assembly of the Stiffness Matrix by P_1-Lagrange finite elements using `base`, `OptV1` and `OptV2` versions respectively (see report). The Stiffness Matrix $\mathbb{S}$ is given by

$$\mathbb{S}_{i,j} = \int_{\Omega_h} \nabla\varphi_j(q).\nabla\varphi_i(q)dq, \quad \forall (i,j) \in \{1,...,\mathrm{n_q}\}^2.$$

Note: generic syntax

`M=StiffAssembling3DP1<version>(nq,nme,q,me,volumes)`
Compute the **stiffness** sparse matrix where `<version>` is `base`, `OptV1` or `OptV2`

Parameters

- **nq** – total number of nodes of the mesh, also denoted by $\mathrm{n_q}$,
- **nme** – total number of tetrahedra, also denoted by $\mathrm{n_{me}}$,
- **q** (*numpy* array of float) –

  array of vertices coordinates,

  - `(nq,3)` array for `base` and `OptV1` versions,
  - `(3,nq)` array for `OptV2` version,
- **me** (*numpy* array of int) –

  Connectivity array,

  - `(nme,4)` array for `base` and `OptV1` versions,
  - `(4,nme)` array for `OptV2` version,
- **volumes** (*numpy* array of floats,) – `(nme,)` array of tetrahedra volumes,

Returns a *Scipy* CSC sparse matrix of size $2\mathrm{n_q} \times 2\mathrm{n_q}$

Benchmarks of theses functions are presented in *Stiffness Matrix*. We give a simple usage :

```
>>> from pyOptFEM.FEM3D import *
>>> Th=CubeMesh(5)
>>> Sbase =  StiffAssembling3DP1base(Th.nq,Th.nme,Th.q,Th.me,Th.volumes)
>>> SOptV1= StiffAssembling3DP1OptV1(Th.nq,Th.nme,Th.q,Th.me,Th.volumes)
>>> print(" NormInf(Sbase-SOptV1)=%e " % NormInf(Sbase-SOptV1))
NormInf(Sbase-SOptV1)=2.220446e-15
```

```
>>> SOptV2= StiffAssembling3DP1OptV2(Th.nq,Th.nme,Th.q,Th.me,Th.volumes)
>>> print(" NormInf(Sbase-SOptV2)=%e " % NormInf(Sbase-SOptV2))
NormInf(Sbase-SOptV2)=2.220446e-15
```

Note: source code

pyOptFEM.FEM3D.assembly.**StiffAssembling3DP1base**(*nq*, *nme*, *q*, *me*, *volumes*)
Assembly of the Stiffness Matrix by P_1-Lagrange finite elements using base version (see report).

pyOptFEM.FEM3D.assembly.**StiffAssembling3DP1OptV1**(*nq*, *nme*, *q*, *me*, *volumes*)
Assembly of the Stiffness Matrix by P_1-Lagrange finite elements using OptV1 version (see report).

pyOptFEM.FEM3D.assembly.**StiffAssembling3DP1OptV2**(*nq*, *nme*, *q*, *me*, *volumes*)
Assembly of the Stiffness Matrix by P_1-Lagrange finite elements using OptV2 version (see report).

5.1.3 Elastic Stiffness Matrix

Assembly of the Elastic Stiffness Matrix by P_1-Lagrange finite elements using base, OptV1 and OptV2 versions respectively (see report). The Elastic Stiffness Matrix $\mathbb{K}$ is given by

$$\mathbb{K}_{m,l} = \int_{\Omega_h} \underline{\boldsymbol{\epsilon}}^t(\psi_l(q))\, \underline{\boldsymbol{\sigma}}(\psi_m(q)), \quad \forall (m,l) \in \{1,...,3\mathrm{n_q}\}^2.$$

where $\underline{\boldsymbol{\sigma}} = (\sigma_{xx}, \sigma_{yy}, \sigma_{zz}, \sigma_{xy}, \sigma_{yz}, \sigma_{xz})^t$ and $\underline{\boldsymbol{\epsilon}} = (\epsilon_{xx}, \epsilon_{yy}, \epsilon_{zz}, 2\epsilon_{xy}, 2\epsilon_{yz}, 2\epsilon_{xz})^t$ are the elastic stress and strain tensors respectively.

Note: generic syntax

M=StiffElasAssembling3DP1<version>(nq,nme,q,me,volumes,la,mu,Num)
Compute the **elastic stiffness** sparse matrix where <version> is base, OptV1 or OptV2

Parameters

- **nq** – total number of nodes of the mesh,
- **nme** – total number of tetrahedrons,
- **q** (*numpy* array of float) –

  vertices coordinates,

  – (nq,3) array for base and OptV1 versions,

  – (3,nq) array for OptV2 version,
- **me** (*numpy* array of int) –

  connectivity array,

  – (nme,4) array for base and OptV1 versions,

  – (4,nme) array for OptV2 version,
- **volumes** (*numpy* array of floats,) – (nme,) array of tetrahedra volumes,
- **la** – the first Lame coefficient in Hooke's law, denoted by λ,
- **mu** – the second Lame coefficient in Hooke's law, denoted by μ,
- **Num** –

- 0 global alternate numbering with local alternate numbering (classical method),
- 1 global block numbering with local alternate numbering,
- 2 global alternate numbering with local block numbering,
- 3 global block numbering with local block numbering.

Returns a *Scipy* CSC sparse matrix of size $3nq \times 3nq$

```
>>> from pyOptFEM.FEM3D import *
>>> Th=CubeMesh(5)
>>> Kbase =  StiffElasAssembling3DP1base(Th.nq,Th.nme,Th.q,Th.me,Th.volumes,2,0.5,0)
>>> KOptV1= StiffElasAssembling3DP1OptV1(Th.nq,Th.nme,Th.q,Th.me,Th.volumes,2,0.5,0)
>>> print(" NormInf(Kbase-KOptV1)=%e " % NormInf(Kbase-KOptV1))
NormInf(Kbase-KOptV1)=1.332268e-15
>>> KOptV2= StiffElasAssembling3DP1OptV2(Th.nq,Th.nme,Th.q,Th.me,Th.volumes,2,0.5,0)
>>> print(" NormInf(Kbase-KOptV2)=%e " % NormInf(Kbase-KOptV2))
NormInf(Kbase-KOptV2)=1.332268e-15
```

We now illustrate the consequences of the choice of the global basis on matrix sparsity

- global *alternate* basis $\mathcal{B}_a$ (`Num=0` or `Num=2`)

```
>>> from pyOptFEM.FEM3D import *
>>> Th=CubeMesh(5)
>>> K0=StiffElasAssembling3DP1OptV1(Th.nq,Th.nme,Th.q,Th.me,Th.volumes,2,0.5,0)
>>> showSparsity(K0)
```

- global *block* basis $\mathcal{B}_a$ (`Num=1` or `Num=3`)

```
>>> K3=StiffElasAssembling3DP1OptV1(Th.nq,Th.nme,Th.q,Th.me,Th.volumes,2,0.5,3)
>>> showSparsity(K3)
```

Note: source code

`pyOptFEM.FEM3D.assembly.`**`StiffElasAssembling3DP1base`**(*nq*, *nme*, *q*, *me*, *volumes*, *la*, *mu*, *Num*)

Assembly of the Elasticity Stiffness Matrix by P_1-Lagrange finite elements using `base` version (see report).

`pyOptFEM.FEM3D.assembly.`**`StiffElasAssembling3DP1OptV1`**(*nq*, *nme*, *q*, *me*, *volumes*, *la*, *mu*, *Num*)

Assembly of the Elasticity Stiffness Matrix by P_1-Lagrange finite elements using `OptV1` version (see report).

`pyOptFEM.FEM3D.assembly.`**`StiffElasAssembling3DP1OptV2`**(*nq*, *nme*, *q*, *me*, *volumes*, *la*, *mu*, *Num*)

Assembly of the Elasticity Stiffness Matrix by P_1-Lagrange finite elements using `OptV2` version (see report).

5.2 Element matrix (used by `base` and `OptV1` versions)

Let T be a tetrahedron of volume $|T|$ and with $\tilde{q}^1$, $\tilde{q}^2$, $\tilde{q}^3$ and $\tilde{q}^4$ its four vertices. We denote by $\tilde{\varphi}_1$, $\tilde{\varphi}_2$, $\tilde{\varphi}_3$ and $\tilde{\varphi}_4$ the P_1-Lagrange local basis functions such that $\tilde{\varphi}_i(\boldsymbol{q}^j) = \delta_{i,j},\ \forall (i,j) \in \{1,\ldots,4\}^2$.

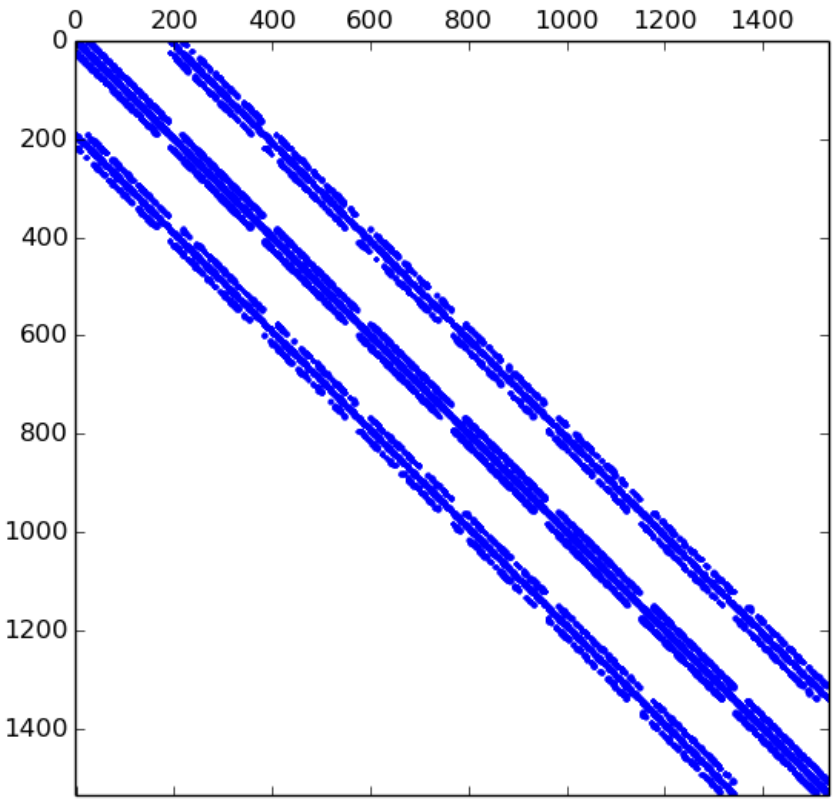


Figure 5.2: Sparsity of the Elastic Stiffness Matrix generated with global alternate numbering (Num=0 or 2)

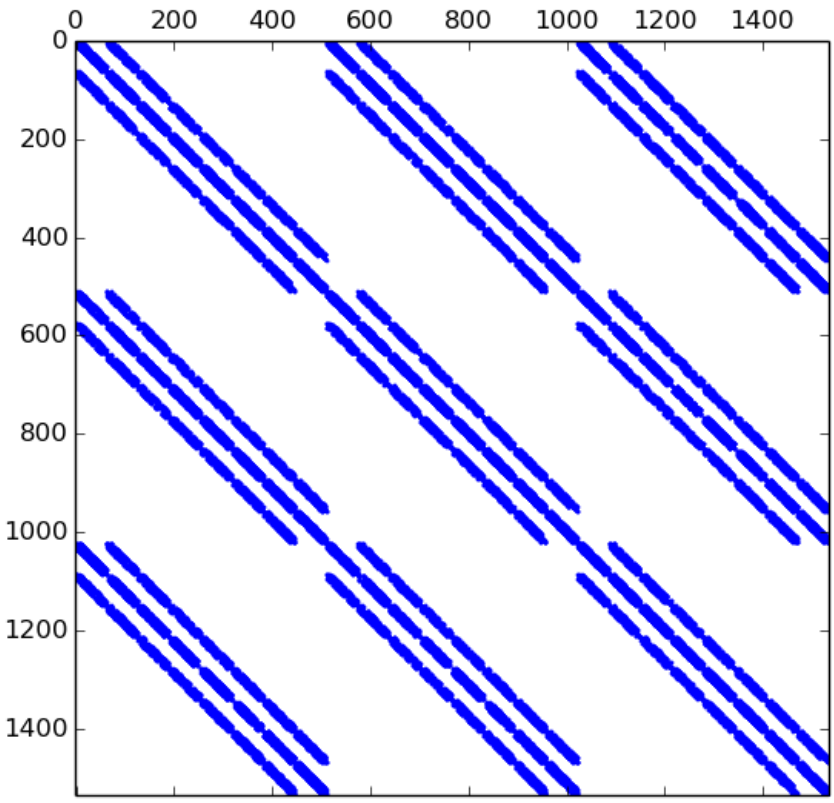


Figure 5.3: Sparsity of the Elastic Stiffness Matrix generated with global block numbering (Num=1 or 3)

We also define the local *alternate* basis $\tilde{\mathcal{B}}_a$ by

$$\tilde{\mathcal{B}}_a = \{\tilde{\boldsymbol{\psi}}_1, \dots, \tilde{\boldsymbol{\psi}}_{12}\} = \left\{ \begin{pmatrix} \tilde{\varphi}_1 \\ 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ \tilde{\varphi}_1 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 0 \\ \tilde{\varphi}_1 \end{pmatrix}, \dots, \begin{pmatrix} \tilde{\varphi}_4 \\ 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ \tilde{\varphi}_4 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 0 \\ \tilde{\varphi}_4 \end{pmatrix}, \right\}$$

and the local *block* basis $\tilde{\mathcal{B}}_b$ by

$$\tilde{\mathcal{B}}_b = \{\tilde{\boldsymbol{\phi}}_1, \dots, \tilde{\boldsymbol{\phi}}_{12}\} = \left\{ \begin{pmatrix} \tilde{\varphi}_1 \\ 0 \\ 0 \end{pmatrix}, \begin{pmatrix} \tilde{\varphi}_2 \\ 0 \\ 0 \end{pmatrix}, \begin{pmatrix} \tilde{\varphi}_3 \\ 0 \\ 0 \end{pmatrix}, \begin{pmatrix} \tilde{\varphi}_4 \\ 0 \\ 0 \end{pmatrix}, \dots, \begin{pmatrix} 0 \\ 0 \\ \tilde{\varphi}_1 \end{pmatrix}, \begin{pmatrix} 0 \\ 0 \\ \tilde{\varphi}_2 \end{pmatrix}, \begin{pmatrix} 0 \\ 0 \\ \tilde{\varphi}_3 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \\ \tilde{\varphi}_4 \end{pmatrix} \right\}.$$

The **elasticity tensor**, $\mathbb{H}$, obtained from Hooke's law with an isotropic material, defined with the Lamé parameters λ and μ is given by

$$\mathbb{H} = \begin{pmatrix} \lambda+2\mu & \lambda & \lambda & 0 & 0 & 0 \\ \lambda & \lambda+2\mu & \lambda & 0 & 0 & 0 \\ \lambda & \lambda & \lambda+2\mu & 0 & 0 & 0 \\ 0 & 0 & 0 & \mu & 0 & 0 \\ 0 & 0 & 0 & 0 & \mu & 0 \\ 0 & 0 & 0 & 0 & 0 & \mu \end{pmatrix}$$

and, for a function $\boldsymbol{u} = (u_1, u_2, u_3)$ the strain tensors is given by

$$\underline{\boldsymbol{\epsilon}}(\boldsymbol{u}) = \left(\frac{\partial u_1}{\partial x}, \quad \frac{\partial u_2}{\partial y}, \quad \frac{\partial u_3}{\partial z}, \quad \frac{\partial u_2}{\partial x} + \frac{\partial u_1}{\partial y}, \quad \frac{\partial u_2}{\partial z} + \frac{\partial u_3}{\partial y}, \quad \frac{\partial u_1}{\partial z} + \frac{\partial u_3}{\partial x} \right)^t$$

5.2.1 Element Mass Matrix

The element Mass matrix $\mathbb{M}^e(T)$ for the tetrahedron T, is defined by

$$\mathbb{M}^e_{i,j}(T) = \int_T \tilde{\varphi}_i(\mathrm{q})\tilde{\varphi}_j(\mathrm{q})\, dq, \quad \forall (i,j) \in \{1,2,3,4\}^2$$

We obtain :

$$\mathbb{M}^e(T) = \frac{|T|}{20} \begin{pmatrix} 2 & 1 & 1 & 1 \\ 1 & 2 & 1 & 1 \\ 1 & 1 & 2 & 1 \\ 1 & 1 & 1 & 2 \end{pmatrix}$$

Note: source code

`pyOptFEM.FEM3D.elemMatrix.`**`ElemMassMat3DP1`**`(V)`

Computes the element mass matrix $\mathbb{M}^e(T)$ for a given tetrahedron T of volume $|T|$

Parameters **V** (*float*) – volume of the tetrahedron.

Returns 4×4 *numpy* array of floats.

5.2.2 Element Stiffness Matrix

The element stiffness matrix, $\mathbb{S}^e(T)$, for the T is defined by

$$\mathbb{S}^e_{i,j}(T) = \int_T < \nabla\tilde{\varphi}_i(\boldsymbol{q}), \nabla\tilde{\varphi}_j(\boldsymbol{q}) > \, d\boldsymbol{q}, \;\; (i,j) \in \{1,2,3,4\}^2$$

Note: source code

pyOptFEM.FEM3D.elemMatrix.**ElemStiffMat3DP1**(*ql*, *volume*)

Computes the element stiffness matrix $\mathbb{S}^e(T)$ for a given tetrahedron T

Parameters

- **ql** (2×4 *numpy* array) – the four vertices of the tetrahedron,
- **volume** (*float*) – volume of the tetrahedron.

Returns

Type 4×4 *numpy* array of floats.

5.2.3 Element Elastic Stiffness Matrix

- The element elastic stiffness matrix, $\mathbb{K}^e(T)$, for a given tetrahedron T in the local *alternate* basis $\tilde{\mathcal{B}}_a$ is defined by

$$\mathbb{K}^e_{i,j}(T) = \int_T \underline{\boldsymbol{\epsilon}}(\tilde{\boldsymbol{\psi}}_j)^t(\boldsymbol{q})\mathbb{H}\underline{\boldsymbol{\epsilon}}(\tilde{\boldsymbol{\psi}}_i)(\boldsymbol{q})\, d\boldsymbol{q}, \quad \forall (i,j) \in \{1,\ldots,12\}^2.$$

We also have

$$\mathbb{K}^e(T) = \frac{1}{4|T|}\mathbb{B}^t\mathbb{H}\mathbb{B}$$

where $\mathbb{H}$ is the elasticity tensor and $\mathbb{B}$ is a 6×12 matrix defined by

$$\mathbb{B} = \left(\underline{\boldsymbol{\epsilon}}(\tilde{\boldsymbol{\psi}}_1) \;\; | \;\; \underline{\boldsymbol{\epsilon}}(\tilde{\boldsymbol{\psi}}_2) \;\; \ldots \;\; \underline{\boldsymbol{\epsilon}}(\tilde{\boldsymbol{\psi}}_{11}) \;\; | \;\; \underline{\boldsymbol{\epsilon}}(\tilde{\boldsymbol{\psi}}_{12})\right)$$

So in $\tilde{\mathcal{B}}_a$ basis we obtain

$$\mathbb{B} = \begin{pmatrix} \frac{\partial\tilde{\varphi}_1}{\partial x} & 0 & 0 & & \frac{\partial\tilde{\varphi}_4}{\partial x} & 0 & 0 \\ 0 & \frac{\partial\tilde{\varphi}_1}{\partial y} & 0 & & 0 & \frac{\partial\tilde{\varphi}_4}{\partial y} & 0 \\ 0 & 0 & \frac{\partial\tilde{\varphi}_1}{\partial z} & & 0 & 0 & \frac{\partial\tilde{\varphi}_4}{\partial z} \\ \frac{\partial\tilde{\varphi}_1}{\partial y} & \frac{\partial\tilde{\varphi}_1}{\partial x} & 0 & \cdots\cdots & \frac{\partial\tilde{\varphi}_4}{\partial y} & \frac{\partial\tilde{\varphi}_4}{\partial x} & 0 \\ 0 & \frac{\partial\tilde{\varphi}_1}{\partial z} & \frac{\partial\tilde{\varphi}_1}{\partial y} & & 0 & \frac{\partial\tilde{\varphi}_4}{\partial z} & \frac{\partial\tilde{\varphi}_4}{\partial y} \\ \frac{\partial\tilde{\varphi}_1}{\partial z} & 0 & \frac{\partial\tilde{\varphi}_1}{\partial x} & & \frac{\partial\tilde{\varphi}_4}{\partial z} & 0 & \frac{\partial\tilde{\varphi}_4}{\partial x} \end{pmatrix}$$

Note: source code

`pyOptFEM.FEM3D.elemMatrix.`**`ElemStiffElasMatBa3DP1`**`(`*ql*, *V*, *C*`)`

Returns the element elastic stiffness matrix $\mathbb{K}^e(T)$ for a given tetrahedron T in the local *alternate* basis $\mathcal{B}_a$

Parameters

- **ql** (4×2 *numpy* array) – contains the four vertices of the tetrahedron,
- **V** (*float*) – volume of the tetrahedron
- **H** (6×6 *numpy* array) – Elasticity tensor, $\mathbb{H}$.

Returns $\mathbb{K}^e(T)$ in $\mathcal{B}_a$ basis.

Type 12×12 *numpy* array of floats.

- The element elastic stiffness matrix, $\mathbb{K}^e(T)$, for a given triangle T in the local *block* basis $\tilde{\mathcal{B}}_b$ is defined by

$$\mathbb{K}^e_{i,j}(T) = \int_T \underline{\boldsymbol{\epsilon}}(\tilde{\boldsymbol{\phi}}_j)^t(\boldsymbol{q})\mathbb{H}\underline{\boldsymbol{\epsilon}}(\tilde{\boldsymbol{\phi}}_i)(\boldsymbol{q})\, d\boldsymbol{q}, \quad \forall (i,j) \in \{1,\ldots,6\}^2.$$

We also have

$$\mathbb{K}^e(T) = \frac{1}{4|T|}\mathbb{B}^t\mathbb{H}\mathbb{B}$$

where $\mathbb{H}$ is the elasticity tensor and $\mathbb{B}$ is a 6×12 matrix defined by

$$\mathbb{B} = \begin{pmatrix} \underline{\boldsymbol{\epsilon}}(\tilde{\boldsymbol{\phi}}_1) & \vdots & \underline{\boldsymbol{\epsilon}}(\tilde{\boldsymbol{\phi}}_2) & \ldots & \underline{\boldsymbol{\epsilon}}(\tilde{\boldsymbol{\phi}}_{11}) & \vdots & \underline{\boldsymbol{\epsilon}}(\tilde{\boldsymbol{\phi}}_{12}) \end{pmatrix}$$

So in $\tilde{\mathcal{B}}_b$ basis we obtain

$$\mathbb{B} = \begin{pmatrix} \frac{\partial\tilde{\varphi}_1}{\partial x} & \ldots & \frac{\partial\tilde{\varphi}_4}{\partial x} & 0 & \ldots & 0 & 0 & \ldots & 0 \\ 0 & \ldots & 0 & \frac{\partial\tilde{\varphi}_1}{\partial y} & \ldots & \frac{\partial\tilde{\varphi}_4}{\partial y} & 0 & \ldots & 0 \\ 0 & \ldots & 0 & 0 & \ldots & 0 & \frac{\partial\tilde{\varphi}_1}{\partial z} & \ldots & \frac{\partial\tilde{\varphi}_4}{\partial z} \\ \frac{\partial\tilde{\varphi}_1}{\partial y} & \ldots & \frac{\partial\tilde{\varphi}_4}{\partial y} & \frac{\partial\tilde{\varphi}_1}{\partial x} & \ldots & \frac{\partial\tilde{\varphi}_4}{\partial x} & 0 & \ldots & 0 \\ 0 & \ldots & 0 & \frac{\partial\tilde{\varphi}_1}{\partial z} & \ldots & \frac{\partial\tilde{\varphi}_4}{\partial z} & \frac{\partial\tilde{\varphi}_1}{\partial y} & \ldots & \frac{\partial\tilde{\varphi}_4}{\partial y} \\ \frac{\partial\tilde{\varphi}_1}{\partial z} & \ldots & \frac{\partial\tilde{\varphi}_4}{\partial z} & 0 & \ldots & 0 & \frac{\partial\tilde{\varphi}_1}{\partial x} & \ldots & \frac{\partial\tilde{\varphi}_4}{\partial x} \end{pmatrix}$$

Note: source code

pyOptFEM.FEM3D.elemMatrix.**ElemStiffElasMatBb3DP1**(*ql*, *V*, *C*)

Returns the element elastic stiffness matrix $\mathbb{K}^e(T)$ for a given tetrahedron T in the local *block* basis $\mathcal{B}_b$

Parameters

- **ql** (4×2 *numpy* array) – contains the four vertices of the tetrahedron,
- **V** (*float*) – volume of the tetrahedron
- **H** (6×6 *numpy* array) – Elasticity tensor, $\mathbb{H}$.

Returns $\mathbb{K}^e(T)$ in $\mathcal{B}_b$ basis.

Type 12×12 *numpy* array of floats.

5.3 Vectorized tools (used by `OptV2` version)

5.3.1 Vectorized computation of basis functions gradients

By construction, the gradients of basis functions are constants on each element T_k. So, we denote, $\forall \alpha \in \{1, \ldots, 4\}$, by $\boldsymbol{G}^\alpha$ the $3 \times \mathrm{n_{me}}$ array defined, $\forall k \in \{1, \ldots, \mathrm{n_{me}}\}$, by

$$\boldsymbol{G}^\alpha(:, k) = \nabla \varphi_{\mathrm{me}(\alpha,k)}(\mathrm{q}),\ \forall \mathrm{q} \in T_k.$$

On T_k tetrahedra we set

$$\begin{array}{lcllcl} \boldsymbol{D}^{12} & = & \mathrm{q}^{\mathrm{me}(1,k)} - \mathrm{q}^{\mathrm{me}(2,k)}, & \boldsymbol{D}^{23} & = & \mathrm{q}^{\mathrm{me}(2,k)} - \mathrm{q}^{\mathrm{me}(3,k)} \\ \boldsymbol{D}^{13} & = & \mathrm{q}^{\mathrm{me}(1,k)} - \mathrm{q}^{\mathrm{me}(3,k)}, & \boldsymbol{D}^{24} & = & \mathrm{q}^{\mathrm{me}(2,k)} - \mathrm{q}^{\mathrm{me}(4,k)} \\ \boldsymbol{D}^{14} & = & \mathrm{q}^{\mathrm{me}(1,k)} - \mathrm{q}^{\mathrm{me}(4,k)}, & \boldsymbol{D}^{34} & = & \mathrm{q}^{\mathrm{me}(3,k)} - \mathrm{q}^{\mathrm{me}(4,k)} \end{array}$$

Then, we have

$$\nabla \varphi_1^k(\mathrm{q}) = \tfrac{1}{6|T_k|} \begin{pmatrix} -\boldsymbol{D}_y^{23}\boldsymbol{D}_z^{24} + \boldsymbol{D}_z^{23}\boldsymbol{D}_y^{24} \\ \boldsymbol{D}_x^{23}\boldsymbol{D}_z^{24} - \boldsymbol{D}_z^{23}\boldsymbol{D}_x^{24} \\ -\boldsymbol{D}_x^{23}\boldsymbol{D}_y^{24} + \boldsymbol{D}_y^{23}\boldsymbol{D}_x^{24} \end{pmatrix}, \quad \nabla \varphi_2^k(\mathrm{q}) = \tfrac{1}{6|T_k|} \begin{pmatrix} \boldsymbol{D}_y^{13}\boldsymbol{D}_z^{14} - \boldsymbol{D}_z^{13}\boldsymbol{D}_y^{14} \\ -\boldsymbol{D}_x^{13}\boldsymbol{D}_z^{14} + \boldsymbol{D}_z^{13}\boldsymbol{D}_x^{14} \\ \boldsymbol{D}_x^{13}\boldsymbol{D}_y^{14} - \boldsymbol{D}_y^{13}\boldsymbol{D}_x^{14} \end{pmatrix}$$

$$\nabla \varphi_3^k(\mathrm{q}) = \tfrac{1}{6|T_k|} \begin{pmatrix} -\boldsymbol{D}_y^{12}\boldsymbol{D}_z^{14} + \boldsymbol{D}_z^{12}\boldsymbol{D}_y^{14} \\ \boldsymbol{D}_x^{12}\boldsymbol{D}_z^{14} - \boldsymbol{D}_z^{12}\boldsymbol{D}_x^{14} \\ -\boldsymbol{D}_x^{12}\boldsymbol{D}_y^{14} + \boldsymbol{D}_y^{12}\boldsymbol{D}_x^{14} \end{pmatrix}, \quad \nabla \varphi_4^k(\mathrm{q}) = \tfrac{1}{6|T_k|} \begin{pmatrix} \boldsymbol{D}_y^{12}\boldsymbol{D}_z^{13} - \boldsymbol{D}_z^{12}\boldsymbol{D}_y^{13} \\ -\boldsymbol{D}_x^{12}\boldsymbol{D}_z^{13} + \boldsymbol{D}_z^{12}\boldsymbol{D}_x^{13} \\ \boldsymbol{D}_x^{12}\boldsymbol{D}_y^{13} - \boldsymbol{D}_y^{12}\boldsymbol{D}_x^{13} \end{pmatrix}$$

With these formulas, we obtain the vectorized algorithm given in Algorithm 5.4.

Algorithm 5.4

Input :

q	:	array of vertices coordinates $(3 \times \mathrm{n_q})$
me	:	connectivity array $(4 \times \mathrm{n_{me}})$
areas	:	array of mesh elements volumes $(1 \times \mathrm{n_{me}})$

Output :

$\boldsymbol{G}^1, \boldsymbol{G}^2, \boldsymbol{G}^3, \boldsymbol{G}^4$	:	gradients arrays $(3 \times \mathrm{n_{me}})$ $\boldsymbol{G}^\alpha(:,k) = \nabla\, \varphi^k_\alpha(\mathrm{q}),\ \forall \alpha \in \{1,\dots,4\}$

Function: $[\boldsymbol{G}^1, \boldsymbol{G}^2, \boldsymbol{G}^3, \boldsymbol{G}^4] \leftarrow$ GradientVec3D$(\mathrm{q}, \mathrm{me}, \mathrm{volumes})$

$\boldsymbol{D}^{12} \leftarrow \mathrm{q}(:, \mathrm{me}(1,:)) - \mathrm{q}(:, \mathrm{me}(2,:))$ ▷ $3 \times \mathrm{n_{me}}$ array

$\boldsymbol{D}^{13} \leftarrow \mathrm{q}(:, \mathrm{me}(1,:)) - \mathrm{q}(:, \mathrm{me}(3,:))$

$\boldsymbol{D}^{14} \leftarrow \mathrm{q}(:, \mathrm{me}(1,:)) - \mathrm{q}(:, \mathrm{me}(4,:))$

$\boldsymbol{D}^{23} \leftarrow \mathrm{q}(:, \mathrm{me}(2,:)) - \mathrm{q}(:, \mathrm{me}(3,:))$

$\boldsymbol{D}^{24} \leftarrow \mathrm{q}(:, \mathrm{me}(2,:)) - \mathrm{q}(:, \mathrm{me}(4,:))$

$C \leftarrow 1/(6 * \mathrm{volumes})$

$$\boldsymbol{G}^1 \leftarrow \begin{pmatrix} (-\boldsymbol{D}^{23}(2,:).*\boldsymbol{D}^{24}(3,:) + \boldsymbol{D}^{23}(3,:).*\boldsymbol{D}^{24}(2,:)).*C \\ (\boldsymbol{D}^{23}(1,:).*\boldsymbol{D}^{24}(3,:) - \boldsymbol{D}^{23}(3,:).*\boldsymbol{D}^{24}(1,:)).*C \\ (-\boldsymbol{D}^{23}(1,:).*\boldsymbol{D}^{24}(2,:) + \boldsymbol{D}^{23}(2,:).*\boldsymbol{D}^{24}(1,:)).*C \end{pmatrix}$$

$$\boldsymbol{G}^2 \leftarrow \begin{pmatrix} (\boldsymbol{D}^{13}(2,:).*\boldsymbol{D}^{14}(3,:) - \boldsymbol{D}^{13}(3,:).*\boldsymbol{D}^{14}(2,:)).*C \\ (-\boldsymbol{D}^{13}(1,:).*\boldsymbol{D}^{14}(3,:) + \boldsymbol{D}^{13}(3,:).*\boldsymbol{D}^{14}(1,:)).*C \\ (\boldsymbol{D}^{13}(1,:).*\boldsymbol{D}^{14}(2,:) - \boldsymbol{D}^{13}(2,:).*\boldsymbol{D}^{14}(1,:)).*C \end{pmatrix}$$

$$\boldsymbol{G}^3 \leftarrow \begin{pmatrix} (-\boldsymbol{D}^{12}(2,:).*\boldsymbol{D}^{14}(3,:) + \boldsymbol{D}^{12}(3,:).*\boldsymbol{D}^{14}(2,:)).*C \\ (\boldsymbol{D}^{12}(1,:).*\boldsymbol{D}^{14}(3,:) - \boldsymbol{D}^{12}(3,:).*\boldsymbol{D}^{14}(1,:)).*C \\ (-\boldsymbol{D}^{12}(1,:).*\boldsymbol{D}^{14}(2,:) + \boldsymbol{D}^{12}(2,:).*\boldsymbol{D}^{14}(1,:)).*C \end{pmatrix}$$

$$\boldsymbol{G}^4 \leftarrow \begin{pmatrix} (\boldsymbol{D}^{12}(2,:).*\boldsymbol{D}^{13}(3,:) - \boldsymbol{D}^{12}(3,:).*\boldsymbol{D}^{13}(2,:)).*C \\ (-\boldsymbol{D}^{12}(1,:).*\boldsymbol{D}^{13}(3,:) + \boldsymbol{D}^{12}(3,:).*\boldsymbol{D}^{13}(1,:)).*C \\ (\boldsymbol{D}^{12}(1,:).*\boldsymbol{D}^{13}(2,:) - \boldsymbol{D}^{12}(2,:).*\boldsymbol{D}^{13}(1,:)).*C \end{pmatrix}$$

end Function:

Figure 5.4: Vectorized algorithm for computation of basis functions gradients in 3d

5.4 Vectorized element matrix (used by `OptV2` version)

5.4.1 Element Mass Matrix

We have

$$\mathbb{M}^e(T) = \frac{|T|}{20}\begin{pmatrix} 2 & 1 & 1 & 1 \\ 1 & 2 & 1 & 1 \\ 1 & 1 & 2 & 1 \\ 1 & 1 & 1 & 2 \end{pmatrix}$$

Then with $\mathbb{K}_g$ definition (see Section *New Optimized assembly algorithm (OptV2 version)*) , we obtain

$$\mathbb{K}_g(4(i-1)+j,k) = |T_k|\frac{1+\delta_{i,j}}{20} \quad 1 \leq i,j \leq 4,$$

So the vectorized algorithm for $\mathbb{K}_g$ computation is simple and given in Algorithm 5.5.

Algorithm 5.5

Algorithm 1 $\mathbb{K}_g$ associated to Mass matrix in 3d

Function: $\mathbb{K}_g \leftarrow$ ElemMassMat3DP1Vec$(\mathrm{n_{me}}, \text{volumes})$
$\mathbb{K}_g \leftarrow \mathbb{O}_{16\times \mathrm{n_{me}}}$ ▷ $16 \times \mathrm{n_{me}}$ array of zeros
$\mathbb{K}_g(1,:) \leftarrow \mathbb{K}_g(6,:) \leftarrow \mathbb{K}_g(11,:) \leftarrow \mathbb{K}_g(16,:) \leftarrow \text{volumes}/10$
$\mathbb{K}_g(2,:) \leftarrow \mathbb{K}_g(3,:) \leftarrow \mathbb{K}_g(4,:) \leftarrow \mathbb{K}_g(5,:)$
$\leftarrow \mathbb{K}_g(7,:) \leftarrow \mathbb{K}_g(8,:) \leftarrow \mathbb{K}_g(9,:) \leftarrow \mathbb{K}_g(10,:)$
$\leftarrow \mathbb{K}_g(12,:) \leftarrow \mathbb{K}_g(13,:) \leftarrow \mathbb{K}_g(14,:) \leftarrow \mathbb{K}_g(15,:) \leftarrow \text{volumes}/20$
end Function:

Figure 5.5: Vectorized algorithm for $\mathbb{K}_g$ associated to 3d **Mass** matrix

Note:

`pyOptFEM.FEM3D.elemMatrixVec.`**`ElemMassMat3DP1Vec`**(*nme*, *volumes*)
Computes all the element Mass matrices $\mathbb{M}^e(T_k)$ for $k \in \{0,\ldots,\mathrm{n_{me}}-1\}$

Parameters **volumes** ($\mathrm{n_{me}}$ *numpy* array of floats) – volumes of all the mesh elements.

Returns a one dimensional *numpy* array of size $16\mathrm{n_{me}}$

5.4.2 Element Stiffness Matrix

We have $\forall(\alpha,\beta) \in \{1,\ldots,4\}^2$

$$\mathbb{S}^e_{\alpha,\beta}(T_k) = |T_k| \left\langle \nabla\varphi^k_\beta, \nabla\varphi^k_\alpha \right\rangle.$$

Using vectorized algorithm function GradientVec3D given in Algorithm 5.4, we obtain the vectorized algorithm 5.6 for $\mathbb{K}_g$ computation for the **Stiffness** matrix in 3d.

Algorithm 5.6

Function: $\mathbb{K}_g \leftarrow$ ElemStiffMat3DP1Vec$(\mathrm{n_{me}}, \mathrm{q}, \mathrm{me}, \mathrm{volumes})$
  $[\boldsymbol{G}^1, \boldsymbol{G}^2, \boldsymbol{G}^3, \boldsymbol{G}^4] \leftarrow$ GradientVec3D$(\mathrm{q}, \mathrm{me}, \mathrm{volumes})$
  $\mathbb{K}_g \leftarrow \mathbb{O}_{16 \times \mathrm{n_{me}}}$ ▷ $16 \times \mathrm{n_{me}}$ array of zeros
  $\mathbb{K}_g(1,:) \leftarrow \mathrm{volumes} .* \mathrm{sum}(\boldsymbol{G}^1 .* \boldsymbol{G}^1, 1)$
  $\mathbb{K}_g(2,:) \leftarrow \mathbb{K}_g(5,:) \leftarrow \mathrm{volumes} .* \mathrm{sum}(\boldsymbol{G}^1 .* \boldsymbol{G}^2, 1)$
  $\mathbb{K}_g(3,:) \leftarrow \mathbb{K}_g(9,:) \leftarrow \mathrm{volumes} .* \mathrm{sum}(\boldsymbol{G}^1 .* \boldsymbol{G}^3, 1)$
  $\mathbb{K}_g(4,:) \leftarrow \mathbb{K}_g(13,:) \leftarrow \mathrm{volumes} .* \mathrm{sum}(\boldsymbol{G}^1 .* \boldsymbol{G}^4, 1)$
  $\mathbb{K}_g(6,:) \leftarrow \mathrm{volumes} .* \mathrm{sum}(\boldsymbol{G}^2 .* \boldsymbol{G}^2, 1)$
  $\mathbb{K}_g(7,:) \leftarrow \mathbb{K}_g(10,:) \leftarrow \mathrm{volumes} .* \mathrm{sum}(\boldsymbol{G}^2 .* \boldsymbol{G}^3, 1)$
  $\mathbb{K}_g(8,:) \leftarrow \mathbb{K}_g(14,:) \leftarrow \mathrm{volumes} .* \mathrm{sum}(\boldsymbol{G}^2 .* \boldsymbol{G}^4, 1)$
  $\mathbb{K}_g(11,:) \leftarrow \mathrm{volumes} .* \mathrm{sum}(\boldsymbol{G}^3 .* \boldsymbol{G}^3, 1)$
  $\mathbb{K}_g(12,:) \leftarrow \mathbb{K}_g(15,:) \leftarrow \mathrm{volumes} .* \mathrm{sum}(\boldsymbol{G}^3 .* \boldsymbol{G}^4, 1)$
  $\mathbb{K}_g(16,:) \leftarrow \mathrm{volumes} .* \mathrm{sum}(\boldsymbol{G}^4 .* \boldsymbol{G}^4, 1)$
end Function:

Figure 5.6: Vectorized algorithm for $\mathbb{K}_g$ associated to 3d **Stiffness** matrix

Note:

`pyOptFEM.FEM3D.elemMatrixVec.`**`ElemStiffMat3DP1Vec`**(*nme*, *q*, *me*, *volumes*)
Computes all the element stiffness matrices $\mathbb{S}^e(T_k)$ for $k \in \{0, \ldots, \mathrm{n_{me}} - 1\}$

Parameters

- **nme** (*int*) – number of mesh elements,
- **q** ($3 \times \mathrm{n_q}$ *numpy* array of floats) – mesh vertices,
- **me** ($4 \times \mathrm{n_{me}}$ *numpy* array of integers) – mesh connectivity,
- **areas** ($\mathrm{n_{me}}$ *numpy* array of floats) – areas of all the mesh elements.

Returns a one dimensional *numpy* array of size $9\mathrm{n_{me}}$

5.4.3 Element Elastic Stiffness Matrix

- We define on the tetrahedron T_k the local *alternate* basis $\mathcal{B}_a^k$ by

$$\mathcal{B}_a^k = \{\boldsymbol{\psi}_1^k, \ldots, \boldsymbol{\psi}_{12}^k\} = \left\{\begin{pmatrix}\varphi_1^k\\0\\0\end{pmatrix}, \begin{pmatrix}0\\\varphi_1^k\\0\end{pmatrix}, \begin{pmatrix}0\\0\\\varphi_1^k\end{pmatrix}, \begin{pmatrix}\varphi_2^k\\0\\0\end{pmatrix}, \begin{pmatrix}0\\\varphi_2^k\\0\end{pmatrix}, \begin{pmatrix}0\\0\\\varphi_2^k\end{pmatrix}, \begin{pmatrix}\varphi_3^k\\0\\0\end{pmatrix}, \begin{pmatrix}0\\\varphi_3^k\\0\end{pmatrix}, \begin{pmatrix}0\\0\\\varphi_3^k\end{pmatrix}, \begin{pmatrix}\varphi_4^k\\0\\0\end{pmatrix}, \begin{pmatrix}0\\\varphi_4^k\\0\end{pmatrix}, \begin{pmatrix}0\\0\\\varphi_4^k\end{pmatrix}\right\}$$

where $\varphi_\alpha^k = \varphi_{\mathrm{me}(\alpha,k)}$. With notations of *Presentation*, we have $\forall(\alpha,\beta) \in \{1,\dots,12\}^2$

$$\mathbb{K}^e_{\alpha,\beta}(T_k) = \int_{T_k} \underline{\boldsymbol{\epsilon}}^t(\boldsymbol{\psi}_\beta^k)\mathbb{C}\underline{\boldsymbol{\epsilon}}(\boldsymbol{\psi}_\alpha^k)d\mathrm{q} = \int_{T_k} \mathcal{H}(\boldsymbol{\psi}_\beta^k, \boldsymbol{\psi}_\alpha^k)(\mathrm{q})d\mathrm{q}$$

with, $\forall \boldsymbol{u} = (u_1,u_2,u_3) \in H^1(\Omega)^3$, $\forall \boldsymbol{v} = (v_1,v_2,v_3) \in H^1(\Omega)^3$, by

$$\begin{array}{c}\mathcal{H}(\boldsymbol{u},\boldsymbol{v}) \\ = \\ \left\langle \begin{pmatrix}\gamma & 0 & 0\\ 0 & \mu & 0\\ 0 & 0 & \mu\end{pmatrix}\nabla u_1, \nabla v_1\right\rangle + \left\langle \begin{pmatrix}0 & \lambda & 0\\ \mu & 0 & 0\\ 0 & 0 & 0\end{pmatrix}\nabla u_2, \nabla v_1\right\rangle + \left\langle \begin{pmatrix}0 & 0 & \lambda\\ 0 & 0 & 0\\ \mu & 0 & 0\end{pmatrix}\nabla u_3, \nabla v_1\right\rangle \\ + \left\langle \begin{pmatrix}0 & \mu & 0\\ \lambda & 0 & 0\\ 0 & 0 & 0\end{pmatrix}\nabla u_1, \nabla v_2\right\rangle + \left\langle \begin{pmatrix}\mu & 0 & 0\\ 0 & \gamma & 0\\ 0 & 0 & \mu\end{pmatrix}\nabla u_2, \nabla v_2\right\rangle + \left\langle \begin{pmatrix}0 & 0 & 0\\ 0 & 0 & \lambda\\ 0 & \mu & 0\end{pmatrix}\nabla u_3, \nabla v_2\right\rangle \\ + \left\langle \begin{pmatrix}0 & 0 & \mu\\ 0 & 0 & 0\\ \lambda & 0 & 0\end{pmatrix}\nabla u_1, \nabla v_3\right\rangle + \left\langle \begin{pmatrix}0 & 0 & 0\\ 0 & 0 & \mu\\ 0 & \lambda & 0\end{pmatrix}\nabla u_2, \nabla v_3\right\rangle + \left\langle \begin{pmatrix}\mu & 0 & 0\\ 0 & \mu & 0\\ 0 & 0 & \gamma\end{pmatrix}\nabla u_3, \nabla v_3\right\rangle\end{array}$$

where λ and μ are the Lame coefficients and $\gamma = \lambda + 2\mu$.

For example, we can explicitly compute the first two terms in the first column of $\mathbb{K}^e(T_k)$ which are given by

$$\begin{array}{rcl}\mathbb{K}^e_{1,1}(T_k) & = & \int_{T_k} \mathcal{H}(\boldsymbol{\psi}_1^k, \boldsymbol{\psi}_1^k)(\mathrm{q})d\mathrm{q} \\ & = & \int_{T_k} \mathcal{H}\left(\begin{pmatrix}\varphi_1^k\\0\\0\end{pmatrix}, \begin{pmatrix}\varphi_1^k\\0\\0\end{pmatrix}\right)(\mathrm{q})d\mathrm{q} \\ & = & |T_k|\left\langle \begin{pmatrix}\gamma & 0 & 0\\ 0 & \mu & 0\\ 0 & 0 & \mu\end{pmatrix}\nabla\varphi_1^k, \nabla\varphi_1^k\right\rangle = |T_k|\left(\gamma\frac{\partial\varphi_1^k}{\partial x}\frac{\partial\varphi_1^k}{\partial x} + \mu(\frac{\partial\varphi_1^k}{\partial y}\frac{\partial\varphi_1^k}{\partial y} + \frac{\partial\varphi_1^k}{\partial z}\frac{\partial\varphi_1^k}{\partial z})\right).\end{array}$$

and

$$\begin{array}{rcl}\mathbb{K}^e_{2,1}(T_k) & = & \int_{T_k} \mathcal{H}(\boldsymbol{\psi}_1^k, \boldsymbol{\psi}_2^k)(\mathrm{q})d\mathrm{q} \\ & = & \int_{T_k} \mathcal{H}\left(\begin{pmatrix}\varphi_1^k\\0\\0\end{pmatrix}, \begin{pmatrix}0\\\varphi_1^k\\0\end{pmatrix}\right)(\mathrm{q})d\mathrm{q} \\ & = & |T_k|\left\langle \begin{pmatrix}0 & \mu & 0\\ \lambda & 0 & 0\\ 0 & 0 & 0\end{pmatrix}\nabla\varphi_1^k, \nabla\varphi_1^k\right\rangle = |T_k|(\lambda+\mu)\frac{\partial\varphi_1^k}{\partial x}\frac{\partial\varphi_1^k}{\partial y}.\end{array}$$

Using vectorized algorithm function GradientVec3D given in Algorithm 5.4, we obtain the vectorized algorithm 5.7 for $\mathbb{K}_g$ computation for the **Elastic Stiffness** matrix in 3d.

Algorithm 5.7

Function: $\mathbb{K}_g \leftarrow$ ElemStiffElasMatBaVec3DP1($\mathrm{n_{me}}$, q, me, volumes, λ, μ)
  $[\boldsymbol{G}^1, \boldsymbol{G}^2, \boldsymbol{G}^3, \boldsymbol{G}^4] \leftarrow$ GradientVec3D(q, me, volumes)
  $\mathbb{K}_g \leftarrow \mathbb{O}_{144\times \mathrm{n_{me}}}$
  $\mathbb{K}_g(1,:) \leftarrow ((\lambda+2\mu) * \boldsymbol{G}^1(1,:).\hat{}\,2 + \mu * (\boldsymbol{G}^1(2,:).\hat{}\,2 + \boldsymbol{G}^1(3,:).\hat{}\,2)). * \mathrm{volumes}$
  $\mathbb{K}_g(2,:) \leftarrow \mathbb{K}_g(13,:) \leftarrow (\lambda+\mu) * \boldsymbol{G}^1(1,:). * \boldsymbol{G}^1(2,:). * \mathrm{volumes}$
  $\vdots$
  $\mathbb{K}_g(132,:) \leftarrow \mathbb{K}_g(143,:) \leftarrow (\lambda+\mu) * \boldsymbol{G}^4(2,:). * \boldsymbol{G}^4(3,:). * \mathrm{volumes}$
  $\mathbb{K}_g(144,:) \leftarrow ((\lambda+2\mu) * \boldsymbol{G}^4(3,:).\hat{}\,2 + \mu * (\boldsymbol{G}^4(1,:).\hat{}\,2 + \boldsymbol{G}^4(2,:).\hat{}\,2). * \mathrm{volumes}$
end Function:

Figure 5.7: Vectorized algorithm for $\mathbb{K}_g$ associated to 3d **Elastic Stiffness** matrix

Note:

pyOptFEM.FEM3D.elemMatrixVec.**ElemStiffElasMatBa3DP1Vec**(*nme*, *q*, *me*, *volumes*, *la*, *mu*)

Computes all the element elastic stiffness matrices $\mathbb{K}^e(T_k)$ for $k \in \{0, \ldots, \mathrm{n_{me}} - 1\}$ in local *alternate* basis.

Parameters

- **nme** (*int*) – number of mesh elements,
- **q** (`(3,nq)` *numpy* array of floats) – mesh vertices,
- **me** (`(4,nme)` *numpy* array of integers) – mesh connectivity,
- **volumes** (`(nme,)` *numpy* array of floats) – volumes of all the mesh elements.
- **la** (*float*) – the $lambda$ Lame parameter,
- **mu** (*float*) – the mu Lame parameter.

Returns a `(144*nme,)` *numpy* array of floats.

- We define on T_k the local *block* basis $\mathcal{B}_b^k$ by

$$\mathcal{B}_b^k = \{\boldsymbol{\phi}_1^k, \ldots, \boldsymbol{\phi}_{12}^k\} = \left\{ \begin{pmatrix} \varphi_1^k \\ 0 \\ 0 \end{pmatrix}, \begin{pmatrix} \varphi_2^k \\ 0 \\ 0 \end{pmatrix}, \begin{pmatrix} \varphi_3^k \\ 0 \\ 0 \end{pmatrix}, \begin{pmatrix} \varphi_4^k \\ 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ \varphi_1^k \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ \varphi_2^k \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ \varphi_3^k \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ \varphi_4^k \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 0 \\ \varphi_1^k \end{pmatrix}, \begin{pmatrix} 0 \\ 0 \\ \varphi_2^k \end{pmatrix}, \begin{pmatrix} 0 \\ 0 \\ \varphi_3^k \end{pmatrix}, \begin{pmatrix} 0 \\ 0 \\ \varphi_4^k \end{pmatrix} \right\}$$

where $\varphi_\alpha^k = \varphi_{\mathrm{me}(\alpha,k)}$.

For example, using formula (**??**), we can explicitly compute the first two terms in the first column of $\mathbb{K}^e(T_k)$ which are given by

$$\begin{aligned} \mathbb{K}_{1,1}^e(T_k) &= \int_{T_k} \mathcal{H}(\boldsymbol{\phi}_1^k, \boldsymbol{\phi}_1^k)(\mathrm{q}) d\mathrm{q} \\ &= \int_{T_k} \mathcal{H}\left(\begin{pmatrix} \varphi_1^k \\ 0 \\ 0 \end{pmatrix}, \begin{pmatrix} \varphi_1^k \\ 0 \\ 0 \end{pmatrix} \right)(\mathrm{q}) d\mathrm{q} \\ &= |T_k| \left\langle \begin{pmatrix} \gamma & 0 & 0 \\ 0 & \mu & 0 \\ 0 & 0 & \mu \end{pmatrix} \nabla \varphi_1^k, \nabla \varphi_1^k \right\rangle = |T_k| \left(\gamma \frac{\partial \varphi_1^k}{\partial x} \frac{\partial \varphi_1^k}{\partial x} + \mu \left(\frac{\partial \varphi_1^k}{\partial y} \frac{\partial \varphi_1^k}{\partial y} + \frac{\partial \varphi_1^k}{\partial z} \frac{\partial \varphi_1^k}{\partial z} \right) \right). \end{aligned}$$

and

$$\begin{aligned} \mathbb{K}_{2,1}^e(T_k) &= \int_{T_k} \mathcal{H}(\boldsymbol{\phi}_1^k, \boldsymbol{\phi}_2^k)(\mathrm{q}) d\mathrm{q} \\ &= \int_{T_k} \mathcal{H}\left(\begin{pmatrix} \varphi_1^k \\ 0 \\ 0 \end{pmatrix}, \begin{pmatrix} \varphi_2^k \\ 0 \\ 0 \end{pmatrix} \right)(\mathrm{q}) d\mathrm{q} \\ &= |T_k| \left\langle \begin{pmatrix} \gamma & 0 & 0 \\ 0 & \mu & 0 \\ 0 & 0 & \mu \end{pmatrix} \nabla \varphi_1^k, \nabla \varphi_2^k \right\rangle = |T_k| \left(\gamma \frac{\partial \varphi_1^k}{\partial x} \frac{\partial \varphi_2^k}{\partial x} + \mu \left(\frac{\partial \varphi_1^k}{\partial y} \frac{\partial \varphi_2^k}{\partial y} + \frac{\partial \varphi_1^k}{\partial z} \frac{\partial \varphi_2^k}{\partial z} \right) \right). \end{aligned}$$

Using vectorized algorithm function GRADIENTVEC3D given in Algorithm 5.4, we obtain the vectorized algorithm 5.8 for $\mathbb{K}_g$ computation for the **Elastic Stiffness** matrix in 3d.

Algorithm 5.8

Note:

pyOptFEM.FEM3D.elemMatrixVec.**ElemStiffElasMatBb3DP1Vec**(*nme*, *q*, *me*, *volumes*, *L*, *M*)

Compute all the element elastic stiffness matrices, $\mathbb{K}^e(T_k)$ for $k \in \{0, \ldots, \mathrm{n_{me}} - 1\}$ in local *block* basis.

1: **Function:** $\mathbb{K}_g \leftarrow$ ElemStiffElasMatBbVec3DP1($\mathrm{n_{me}}$, q, me, volumes, λ, μ)
2: $[\boldsymbol{G}^1, \boldsymbol{G}^2, \boldsymbol{G}^3, \boldsymbol{G}^4] \leftarrow$ GradientVec3D(q, me, volumes)
3: $\mathbb{K}_g \leftarrow \mathbb{O}_{144 \times \mathrm{n_{me}}}$
4: $\mathbb{K}_g(1,:) \leftarrow ((\lambda + 2\mu) * \boldsymbol{G}^1(1,:).\hat{}\,2 + \mu * (\boldsymbol{G}^1(2,:).\hat{}\,2 + \boldsymbol{G}^1(3,:).\hat{}\,2)). * \mathrm{volumes}$
5: $\mathbb{K}_g(2,:) \leftarrow \mathbb{K}_g(13,:) \leftarrow ((\lambda + 2\mu) * \boldsymbol{G}^1(1,:). * \boldsymbol{G}^2(1,:) +$
$\mu(\boldsymbol{G}^1(2,:). * \boldsymbol{G}^2(2,:)) + \boldsymbol{G}^1(3,:). * \boldsymbol{G}^2(3,:))). * \mathrm{volumes}$
$\vdots$
7: $\mathbb{K}_g(132,:) \leftarrow \mathbb{K}_g(13,:) \leftarrow ((\lambda + 2\mu) * \boldsymbol{G}^3(1,:). * \boldsymbol{G}^4(1,:) +$
$\mu(\boldsymbol{G}^3(2,:). * \boldsymbol{G}^4(2,:)) + \boldsymbol{G}^3(3,:). * \boldsymbol{G}^4(3,:))). * \mathrm{volumes}$
8: $\mathbb{K}_g(144,:) \leftarrow ((\lambda + 2\mu) * \boldsymbol{G}^4(3,:).\hat{}\,2 + \mu * (\boldsymbol{G}^4(1,:).\hat{}\,2 + \boldsymbol{G}^4(2,:).\hat{}\,2). * \mathrm{volumes}$
9: **end Function:**

Figure 5.8: Vectorized algorithm for $\mathbb{K}_g$ associated to 3d **Elastic Stiffness** matrix

Parameters

- **nme** (*int*) – number of mesh elements,
- **q** (`(3,nq)` *numpy* array of floats) – mesh vertices,
- **me** (`(4,nme)` *numpy* array of integers) – mesh connectivity,
- **volumes** (`(nme,)` *numpy* array of floats) – volumes of all the mesh elements.
- **la** (*float*) – the
  lambda Lame parameter,
- **mu** (*float*) – the
  mu Lame parameter.

Returns a `(144*nme,)` *numpy* array of floats.

5.5 Mesh

class `pyOptFEM.FEM3D.mesh.`**`CubeMesh`**(*N*, ***kwargs*)

Creates meshes of the unit cube $[0,1]^3$. Class attributes are :

•**nq**, total number of mesh vertices (points), also denoted $\mathrm{n_q}$.

•**nme**, total number of mesh elements (tetrahedra in 3d),

•**version**, mesh structure version,

•**q**, *Numpy* array of vertices coordinates, dimension `(nq,3)` *(version 0)* or `(3,nq)` *(version 1)*.

`q[j]` *(version 0)* or `q[:,j]` *(version 1)* are the three coordinates of the j-th vertex, $j \in \{0, .., nq-1\}$

•**me**, *Numpy* connectivity array, dimension `(nme,4)` *(version 0)* or `(4,nme)` *(version 1)*.

`me[k]` *(version 0)* or `me[:,k]` *(version 1)* are the storage index of the four vertices of the k-th tetrahedron in the array `q` of vertices coordinates, $k \in \{0, ..., nme-1\}$.

•**volumes**, Array of mesh elements volumes, `(nme,)` *Numpy* array.

`volumes[k]` is the volume of k-th tetrahedron, `k in range(0,nme)`

Parameters **N** – number of points on each edge of the cube

optional parameter : `version=0` or `version=1`

class `pyOptFEM.FEM3D.mesh.`**`getMesh`**(*filename*, ***kwargs*)

Reads a *medit* mesh from file `meshfile`. Class attributes are :

- **nq**, total number of mesh vertices (points), also denoted $\mathrm{n_q}$.
- **nme**, total number of mesh elements (tetrahedra in 3d),
- **version**, mesh structure version,
- **q**, *Numpy* array of vertices coordinates, dimension `(nq,3)` *(version 0)* or `(3,nq)` *(version 1)*.

  `q[j]` *(version 0)* or `q[:,j]` *(version 1)* are the three coordinates of the j-th vertex, $j \in \{0, .., nq-1\}$
- **me**, *Numpy* connectivity array, dimension `(nme,4)` *(version 0)* or `(4,nme)` *(version 1)*.

  `me[k]` *(version 0)* or `me[:,k]` *(version 1)* are the storage index of the four vertices of the k-th tetrahedron in the array `q` of vertices coordinates, $k \in \{0, ..., nme-1\}$.
- **volumes**, Array of mesh elements volumes, `(nme,)` *Numpy* array.

  `volumes[k]` is the volume of k-th tetrahedron, `k in range(0,nme)`

Parameters **meshfile** – *medit* mesh file

optional parameter : `version=0` or `version=1`

CHAPTER

SIX

VALID2D MODULE

Contents

6.1 Mass Matrix

Validation program for the assembly of the **Mass** matrix $\mathbb{M}$ for P_1-Lagrange finite element method in 2d (see *Mass Matrix*).

- *Test 1:* Computation of the **Mass** Matrix using the `base`, `OptV1` and `OptV2` versions : it gives errors and computation times
- *Test 2:* Computation of the integral

$$\int_{\Omega_h} u(x,y)v(x,y)dxdy \approx \boldsymbol{V}^t \mathbb{M} \boldsymbol{U}$$

  where $\boldsymbol{U}_i = u(q^i)$ and $\boldsymbol{V}_i = v(q^i)$. Functions u and v are those defined in ...

- *Test 3:* Ones retrieves the order 2 of P_1-Lagrange integration

$$|\int_{\Omega_h} u\, v - \Pi_h(u)\, \Pi_h(v) d\Omega| \leq Ch^2$$

Note: source code

`pyOptFEM.valid2D.validMass2DP1.`**`validMass2DP1`**`(`*`**kwargs`*`)`

Validation of Mass Matrix for P1-Lagrange finite elements in 2D

```
>>> from pyOptFEM.valid2D import validMass2DP1
>>> validMass2DP1()
****************************************
*     Mass Assembling P1 validations     *
****************************************
----------------------------------------
```

```
Test 1: Matrices errors and CPU times
-------------------------------------
    Matrix size           : (121,121)
    Error P1base vs OptV1 : 8.673617e-19
    Error P1base vs OptV2 : 8.673617e-19
    CPU times base (ref)  : 0.2638 (s)
    CPU times OptV1       : 0.0259 (s) - Speed Up X10.179
    CPU times OptV2       : 0.0016 (s) - Speed Up X160.286
-----------------------------
  Test 1 (results): OK
-----------------------------
-----------------------------------------------------
  Test 2: Validations by integration on [0,1]x[0,1]
-----------------------------------------------------
    function 0 : u(x,y)=x+2*y, v(x,y)=3*x+y+1,
            -> Mass error=0.000000e+00
    function 1 : u(x,y)=x**2+2*y*x+y, v(x,y)=3*x*y+y**2+1,
            -> Mass error=5.736389e-03
    function 2 : u(x,y)=x**3+2*y**2*x+y**2+x, v(x,y)=2*x*y+y**3+x*y,
            -> Mass error=1.141472e-02
-----------------------------
  Test 2 (results): OK
-----------------------------
--------------------------------
  Test 3: Validations by order
--------------------------------
  functions 2: u(x,y)=x**3+2*y**2*x+y**2+x, v(x,y)=2*x*y+y**3+x*y
      Matrix size                     : (121,121)
      MassAssemblingP1OptV2 CPU times : 0.001(s)
      Error                           : 1.141472e-02
      Matrix size                     : (441,441)
      MassAssemblingP1OptV2 CPU times : 0.002(s)
      Error                           : 2.825605e-03
      ...
      Matrix size                     : (10201,10201)
      MassAssemblingP1OptV2 CPU times : 0.019(s)
      Error                           : 1.125457e-04
```

At last, this program plots the figure :

6.2 Stiffness matrix

Validation function for the assembly of the **Stiffness** matrix, $\mathbb{S}$, for P_1-Lagrange finite element method in 2d (see *Stiffness Matrix*)

- *Test 1:* Computation of the **Stiffness** Matrix using the `base`, `OptV1` and `OptV2` versions : it gives errors and computation times
- *Test 2:* Computation of the integral

$$\int_{\Omega_h} \nabla u(q).\nabla v(q) dq \approx \boldsymbol{V}^t \mathbb{S} \boldsymbol{U}$$

where $\boldsymbol{U}_i = u(q^i)$ and $\boldsymbol{V}_i = v(q^i)$. Functions u and v are those defined in ...

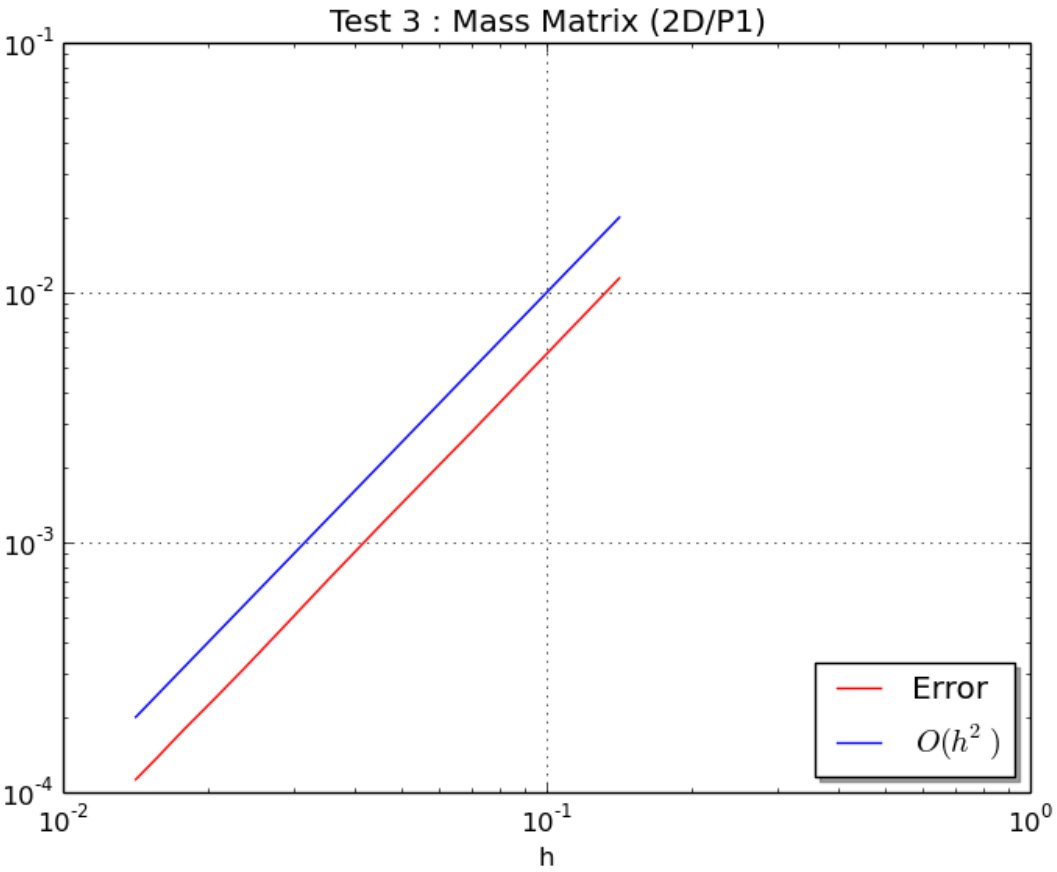


Figure 6.1: Graphical results of validMass2DP1-Test 3

- *Test 3:* Ones retrieves the order 2 of P_1-Lagrange integration

$$|\int_{\Omega_h} \nabla u.\nabla v - \nabla\Pi_h(u).\nabla\Pi_h(v)d\Omega| \leq Ch^2$$

Note: source code

pyOptFEM.valid2D.validStiff2DP1.**validStiff2DP1**(***kwargs*)
validation of stiffness matrix assembling functions

```
>>> from pyOptFEM.valid2D import validStiff2DP1
>>> validStiff2DP1()
*******************************************
*     2D Stiff Assembling P1 validations     *
*******************************************
----------------------------------------
  Test 1: Matrices errors and CPU times
----------------------------------------
    Matrix size           : (121,121)
    Error P1base vs OptV1 : 8.881784e-16
    Error P1base vs OptV2 : 4.440892e-16
    CPU times base (ref)  : 0.1519 (s)
    CPU times OptV1       : 0.0106 (s) - Speed Up X14.270
    CPU times OptV2       : 0.0009 (s) - Speed Up X174.238
----------------------------
  Test 1 (results): OK
----------------------------
------------------------------------------------------
  Test 2: Validations by integration on [0,1]x[0,1]
------------------------------------------------------
    function 0 : u(x,y)=x+2*y, v(x,y)=3*x+y+1,
          -> Stiff error=2.486900e-14
    function 1 : u(x,y)=x**2+2*y*x+y, v(x,y)=3*x*y+y**2+1,
          -> Stiff error=2.000000e-02
    function 2 : u(x,y)=x**3+2*y**2*x+y**2+x, v(x,y)=2*x*y+y**3+x*y,
          -> Stiff error=1.500000e-02
----------------------------
  Test 2 (results): OK
----------------------------
--------------------------------
  Test 3: Validations by order
--------------------------------
Test 2: u(x,y)=x**3+2*y**2*x+y**2+x, v(x,y)=2*x*y+y**3+x*y
      Matrix size                     : (121,121)
      StiffAssemblingP1OptV2 CPU times : 0.001(s)
      Error                           : 1.500000e-02
      ...
      Matrix size                     : (10201,10201)
      StiffAssemblingP1OptV2 CPU times : 0.021(s)
      Error                           : 1.500000e-04
```

At last, this program plots this figure :

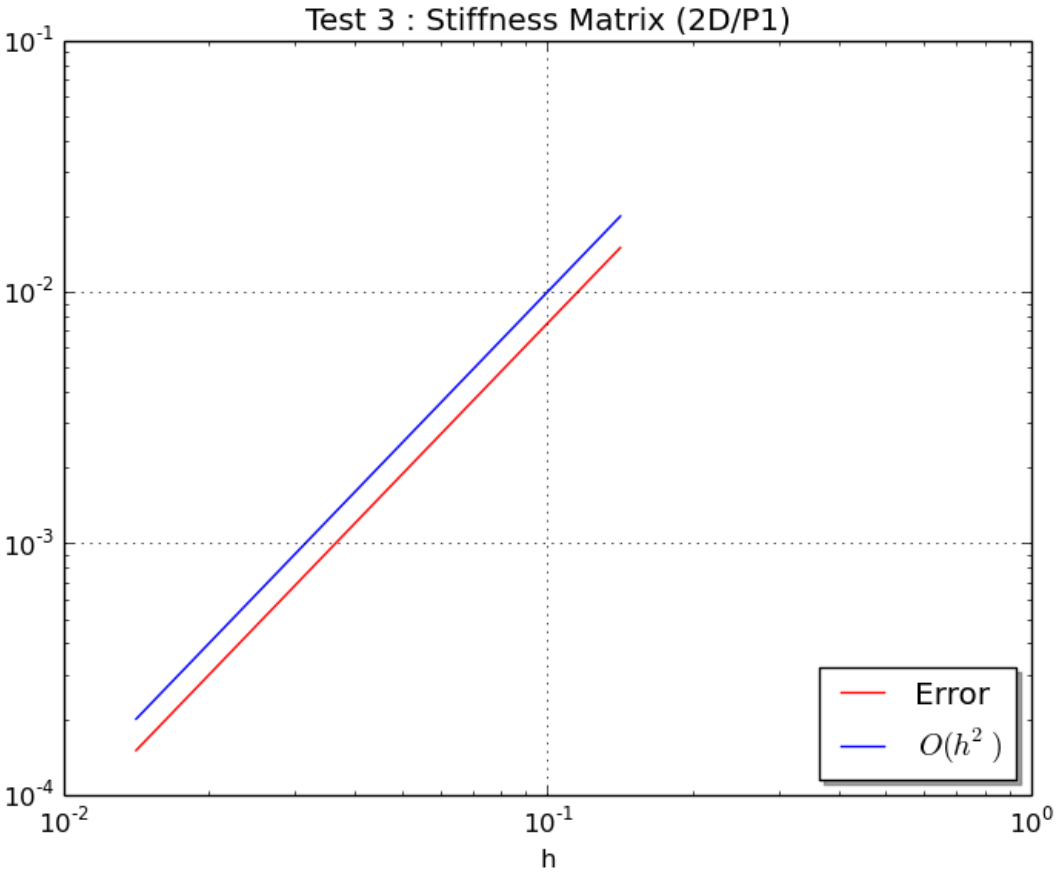


Figure 6.2: Graphical results of validStiff2DP1-Test 3

6.3 Elastic Stiffness Matrix

Validation function for the assembly of the **Elastic Stiffness** matrix, $\mathbb{K}$, for P_1-Lagrange finite element method in 2d (see *Elastic Stiffness Matrix*)

- *Test 1:* Computation of the **Elastic Stiffness** Matrix using the `base`, `OptV1` and `OptV2` versions : it gives errors and computation times
- *Test 2:* Computation of the integral

$$\int_{\Omega_h} \underline{\boldsymbol{\epsilon}}^t(u)\underline{\boldsymbol{\sigma}}(v)dq \approx \boldsymbol{V}^t\mathbb{K}\boldsymbol{U}$$

  where $\boldsymbol{u} = (u_1, u_2)$ and $\boldsymbol{v} = (v_1, v_2)$ are given 2d-vector functions and $\boldsymbol{U}$ and $\boldsymbol{V}$ are the vectors in $\mathbb{R}^{2\mathrm{n_q}}$ defined by

  - if `Num=0` or `Num=2` (i.e. in global *alternate* basis $\mathcal{B}_a$, see *Assembly matrices (base, OptV1 and OptV2 versions)*)

$$\forall i \in \{1, \ldots, \mathrm{n_q}\},\ U_{2i-1} = u_1(\mathrm{q}^i),\ U_{2i} = u_2(\mathrm{q}^i),\ \text{ and }\ V_{2i-1} = v_1(\mathrm{q}^i),\ V_{2i} = v_2(\mathrm{q}^i)$$

  - if `Num=1` or `Num=3` (i.e. in global *block* basis $\mathcal{B}_b$, see *Assembly matrices (base, OptV1 and OptV2 versions)*)

$$\forall i \in \{1, \ldots, \mathrm{n_q}\},\ U_{i} = u_1(\mathrm{q}^i),\ U_{i+\mathrm{n_q}} = u_2(\mathrm{q}^i),\ \text{ and }\ V_{i} = v_1(\mathrm{q}^i),\ V_{i+\mathrm{n_q}} = v_2(\mathrm{q}^i)$$

- *Test 3:* Ones retrieves the order 2 of P_1-Lagrange integration

$$|\int_{\Omega_h} \underline{\boldsymbol{\epsilon}}^t(u)\underline{\boldsymbol{\sigma}}(v) - \underline{\boldsymbol{\epsilon}}^t(\Pi_h(u))\underline{\boldsymbol{\sigma}}(\Pi_h(v))d\Omega| \leq Ch^2$$

Note:

`pyOptFEM.valid2D.validStiff2DP1.`**`validStiffElas2DP1`** ([*Num=0*, *la=1.5*, *mu=0.5*])

Parameters

- **Num** – *(optional)* Numbering choice.
- **la** – *(optional)* the first Lame coefficient in Hooke's law, denoted by λ.
- **mu** – *(optional)* the second Lame coefficient in Hooke's law, denoted by μ.

`pyOptFEM.valid2D.validStiffElas2DP1.`**`validStiffElas2DP1`** (***kwargs*)

validation of elasticity stiffness matrix assembly functions

```
>>> from pyOptFEM.valid2D import validStiffElas2DP1
>>> validStiffElas2DP1()
************************************************
*     2D StiffElas Assembling P1 validations     *
************************************************
```

```
  Numbering Choice : global alternate numbering with local alternate numbering
---------------------------------------
  Test 1: Matrices errors and CPU times
---------------------------------------
    Matrix size           : (1922,1922)
    Error P1base vs OptV1 : 1.776357e-15
    Error P1base vs OptV2 : 2.664535e-15
    CPU times base (ref)  : 1.9251 (s)
    CPU times OptV1       : 0.2817 (s) - Speed Up X6.835
    CPU times OptV2       : 0.0098 (s) - Speed Up X195.725
---------------------------
  Test 1 (results): OK
---------------------------
-----------------------------------------------------
  Test 2: Validations by integration on [0,1]x[0,1]
-----------------------------------------------------
    function 0 :
      u(x,y)=[x - 2*y,x + y],
      v(x,y)=[x + 2*y,2*x - y],
           -> StiffElas error=8.704149e-14
    function 1 :
      u(x,y)=[x**2+2*y*x+y,-2*y**2+x**2+x-y],
      v(x,y)=[3*x*y+y**2+1,3*x**2-x*y+1],
           -> StiffElas error=3.916049e-03
    function 2 :
      u(x,y)=[x**3+2*y**2*x+y**2+x,y**3-2*x**2*y],
      v(x,y)=[2*x*y+y**3+x*y,3*x**3-2*x*y+x-1],
           -> StiffElas error=6.574856e-03
---------------------------
  Test 2 (results): OK
---------------------------
--------------------------------
  Test 3: Validations by order
--------------------------------
    functions 2:
      u(x,y)=['x**3+2*y**2*x+y**2+x', 'y**3-2*x**2*y'],
      v(x,y)=['2*x*y+y**3+x*y', '3*x**3-2*x*y+x-1'],
      lambda=1.500000, mu=0.500000
        Matrix size                          : (242,242)
        StiffElasAssemblingP1OptV2 CPU times : 0.002(s)
        Error                                : 5.767667e-02
        Matrix size                          : (882,882)
        StiffElasAssemblingP1OptV2 CPU times : 0.005(s)
        Error                                : 1.447542e-02
        Matrix size                          : (1922,1922)
        ...
        Matrix size                          : (20402,20402)
        StiffElasAssemblingP1OptV2 CPU times : 0.110(s)
        Error                                : 5.797263e-04
```

At last, this program plots this figure :

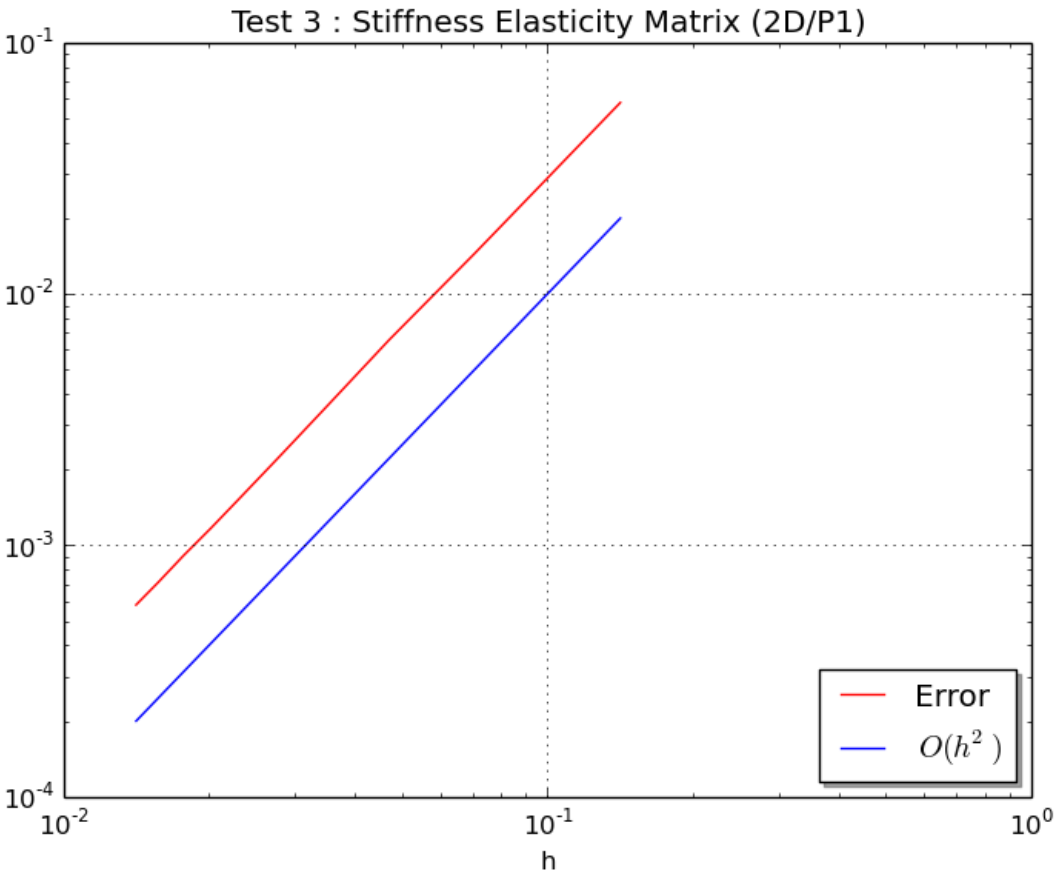


Figure 6.3: Graphical results of validStiffElas2DP1-Test 3

CHAPTER

SEVEN

VALID3D MODULE

Contents

7.1 Mass Matrix

Validation program for the assembly of the **Mass** matrix for P_1-Lagrange finite element method in 3d (see *Mass Matrix*). Meshes of the unit cube are used.

- Test 1: Computation of the **Mass** Matrix using the `base`, `OptV1` and `OptV2` versions : it gives errors and computation times
- Test 2: Computation of the integral

$$\int_{\Omega_h} u(\mathrm{q})v(\mathrm{q})d\mathrm{q} \approx \boldsymbol{V}^t \mathbb{M} \boldsymbol{U}$$

  where $\boldsymbol{U}_i = u(q^i)$ and $\boldsymbol{V}_i = v(q^i)$. Functions u and v are those defined in ...

- Test 3: Ones retrieves the order 2 of P_1-Lagrange integration

$$|\int_{\Omega_h} u\, v - \Pi_h(u)\, \Pi_h(v) d\Omega| \leq Ch^2$$

Note: source code

`pyOptFEM.valid3D.validMass3DP1.`**`validMass3DP1`**`(`***kwargs*`)`

Validation of Mass Matrix for P1-Lagrange finite elements in 3D

```
>>> from pyOptFEM.valid3D import validMass3DP1
>>> validMass3DP1()
*****************************************
*     3D Mass Assembling P1 validations     *
*****************************************
----------------------------------------
```

```
  Test 1: Matrices errors and CPU times
----------------------------------------
    Matrix size           : (216,216)
    Error P1base vs OptV1 : 1.734723e-18
    Error P1base vs OptV2 : 1.734723e-18
    CPU times base (ref)  : 0.9355 (s)
    CPU times OptV1       : 0.0481 (s) - Speed Up X19.438
    CPU times OptV2       : 0.0016 (s) - Speed Up X592.083
----------------------------
  Test 1 (results): OK
----------------------------
-----------------------------------------------------
  Test 2: Validations by integration on [0,1]x[0,1]
-----------------------------------------------------
    function 0 :
      u(x,y,z)=x + y + z,
      v(x,y,z)=x - y - z,
           -> Stiff error=3.330669e-16
    function 1 :
      u(x,y,z)=3*x + 2*y - z - 1,
      v(x,y,z)=2*x - 2*y + 2*z + 1,
           -> Stiff error=8.881784e-16
    function 2 :
      u(x,y,z)=3*x**2 - x*y + 2*y**2 + y*z - z**2 - 3,
      v(x,y,z)=2*x**2 + x*y - 3*y**2 - x*z - y,
           -> Stiff error=1.173244e-02
----------------------------
  Test 2 (results): OK
----------------------------
--------------------------------
  Test 3: Validations by order
--------------------------------
    function 2 :
      u(x,y,z)=3*x**2 - x*y + 2*y**2 + y*z - z**2 - 3,
      v(x,y,z)=2*x**2 + x*y - 3*y**2 - x*z - y
        Matrix size                     : (216,216)
        MassAssemblingP1OptV2 CPU times : 0.002(s)
        -> Error                         : 1.173244e-02
        Matrix size                     : (1331,1331)
        MassAssemblingP1OptV2 CPU times : 0.009(s)
        -> Error                         : 2.800865e-03
        ...
        Matrix size                     : (132651,132651)
        MassAssemblingP1OptV2 CPU times : 1.418(s)
        -> Error                         : 1.110034e-04
```

At last, this program plots the figure :

7.2 Stiffness matrix

Validation function for the assembly of the **Stiffness** matrix for P_1-Lagrange finite element method in 3d (see *Stiffness Matrix*). Meshes of the unit cube are used.

- Test 1: Computation of the **Stiffness** Matrix using the `base`, `OptV1` and `OptV2` versions : it gives errors and computation times

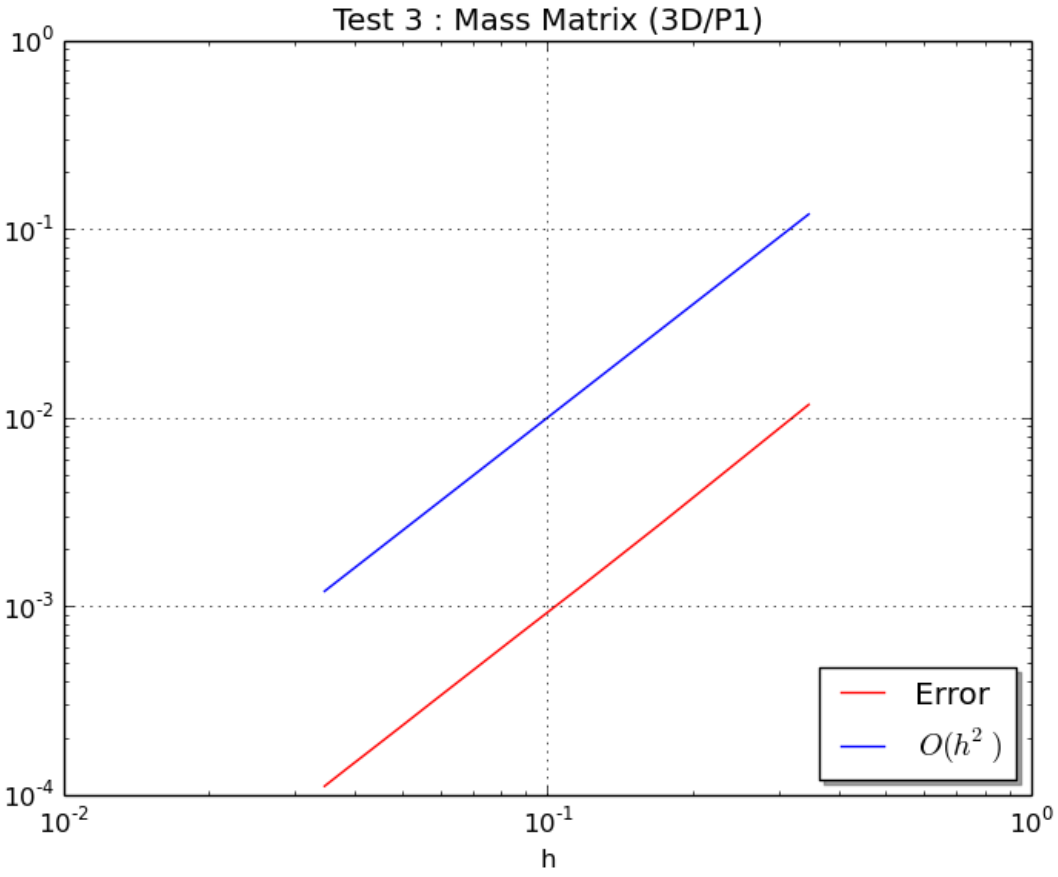


Figure 7.1: Graphical results of validMass3DP1-Test 3

- Test 2: Computation of the integral

$$\int_{\Omega_h} \nabla u(q).\nabla v(q) dq \approx \boldsymbol{V}^t \mathbb{M} \boldsymbol{U}$$

  where $\boldsymbol{U}_i = u(q^i)$ and $\boldsymbol{V}_i = v(q^i)$. Functions u and v are those defined in ...

- Test 3: Ones retrieves the order 2 of P_1-Lagrange integration

$$|\int_{\Omega_h} \nabla u.\nabla v - \nabla \Pi_h(u).\nabla \Pi_h(v) d\Omega| \leq Ch^2$$

Note: source code

pyOptFEM.valid3D.validStiff3DP1.**validStiff3DP1**(***kwargs*)

Validation of Stiffness Matrix for P1-Lagrange finite elements in 3D

```
>>> from pyOptFEM.valid3D import validStiff3DP1
>>> validStiff3DP1()
*******************************************
*     3D Stiff Assembling P1 validations     *
*******************************************
-----------------------------------------
  Test 1: Matrices errors and CPU times
-----------------------------------------
    Matrix size           : (216,216)
    Error P1base vs OptV1 : 2.220446e-15
    Error P1base vs OptV2 : 2.220446e-15
    CPU times base (ref)  : 0.9563 (s)
    CPU times OptV1       : 0.0566 (s) - Speed Up X16.901
    CPU times OptV2       : 0.0032 (s) - Speed Up X299.656
----------------------------
  Test 1 (results): OK
----------------------------
-----------------------------------------------------
  Test 2: Validations by integration on [0,1]x[0,1]
-----------------------------------------------------
    function 0 :
      u(x,y,z)=x + y + z,
      v(x,y,z)=x - y - z,
           -> Stiff error=6.217249e-15
    function 1 :
      u(x,y,z)=3*x + 2*y - z - 1,
      v(x,y,z)=2*x - 2*y + 2*z + 1,
           -> Stiff error=7.771561e-15
    function 2 :
      u(x,y,z)=3*x**2 - x*y + 2*y**2 + y*z - z**2 - 3,
      v(x,y,z)=2*x**2 + x*y - 3*y**2 - x*z - y,
           -> Stiff error=2.077333e-02
----------------------------
  Test 2 (results): OK
----------------------------
--------------------------------
  Test 3: Validations by order
--------------------------------
```

```
function 2 :
  u(x,y,z)=3*x**2 - x*y + 2*y**2 + y*z - z**2 - 3,
  v(x,y,z)=2*x**2 + x*y - 3*y**2 - x*z - y
    Matrix size                          : (216,216)
    StiffAssembling3DP1OptV2 CPU times : 0.003(s)
    Error                                : 2.077333e-02
    Matrix size                          : (1331,1331)
    StiffAssembling3DP1OptV2 CPU times : 0.019(s)
    Error                                : 3.330000e-03
    ...
    Matrix size                          : (132651,132651)
    StiffAssembling3DP1OptV2 CPU times : 2.784(s)
    Error                                : 1.828928e-04
```

At last, this program plots the figure :

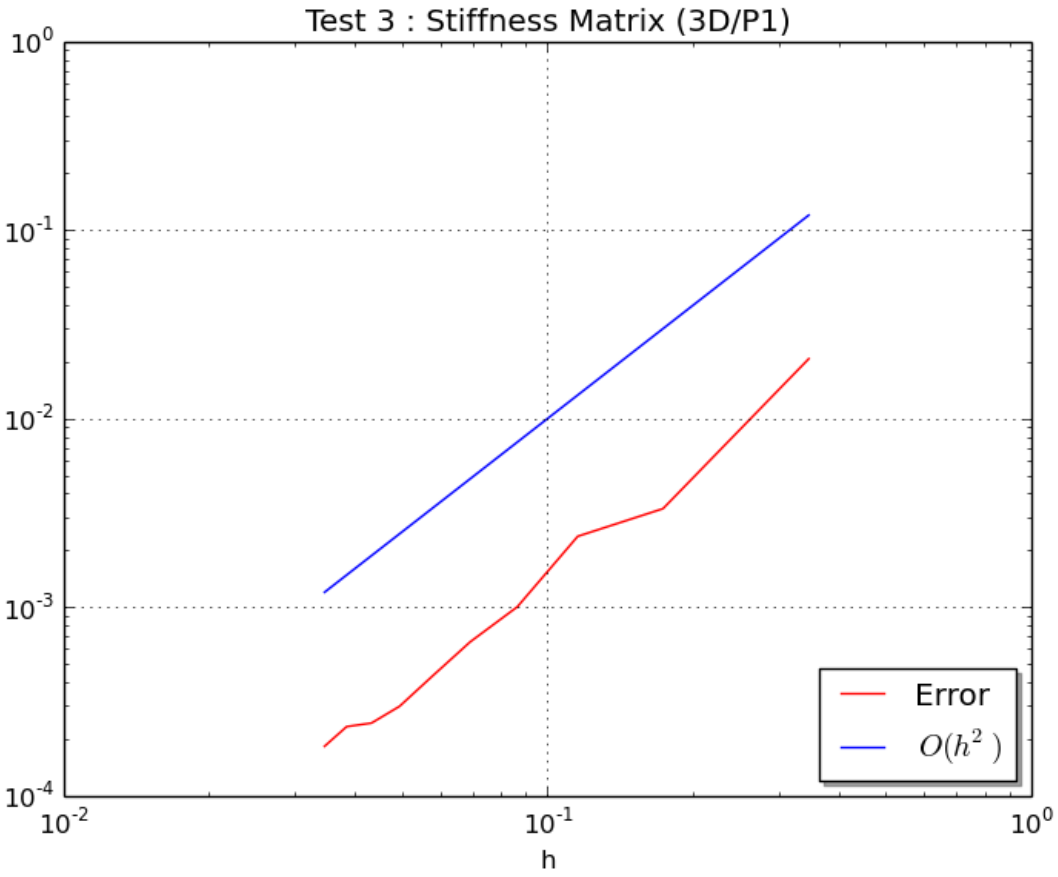


Figure 7.2: Graphical results of validStiff3DP1-Test 3

7.3 Elastic Stiffness Matrix

Validation function for the assembly of the **Elastic Stiffness** matrix for P_1-Lagrange finite element method in 3d (see *Elastic Stiffness Matrix*). Meshes of the unit cube are used.

- Test 1: Computation of the **Elastic Stiffness** Matrix using the `base`, `OptV1` and `OptV2` versions : it gives errors and computation times
- Test 2: Computation of the integral

$$\int_{\Omega_h} \underline{\boldsymbol{\epsilon}}^t(u)\underline{\boldsymbol{\sigma}}(v)dq \approx \boldsymbol{V}^t\mathbb{M}\boldsymbol{U}$$

  where $\boldsymbol{U}_i = u(q^i)$ and $\boldsymbol{V}_i = v(q^i)$. Functions u and v are those defined in ...
- Test 3: Ones retrieves the order 2 of P_1-Lagrange integration

$$|\int_{\Omega_h} \underline{\boldsymbol{\epsilon}}^t(u)\underline{\boldsymbol{\sigma}}(v) - \underline{\boldsymbol{\epsilon}}^t(\Pi_h(u))\underline{\boldsymbol{\sigma}}(\Pi_h(v))d\Omega| \leq Ch^2$$

Note:

`pyOptFEM.valid3D.validStiff3DP1.`**`validStiffElas3DP1`**`(`[*Num=0, la=1.5, mu=0.5*]`)`

Parameters

- **Num** – *(optional)* Numbering choice.
- **la** – *(optional)* the first Lame coefficient in Hooke's law, denoted by λ.
- **mu** – *(optional)* the second Lame coefficient in Hooke's law, denoted by μ.

```
>>> from pyOptFEM.valid3D import validStiffElas3DP1
>>> validStiffElas3DP1()
************************************************
*     3D StiffElas Assembling P1 validations     *
************************************************
----------------------------------------
  Test 1: Matrices errors and CPU times
----------------------------------------
    Matrix size           : (648,648)
    Error P1base vs OptV1 : 1.554312e-15
    Error P1base vs OptV2 : 1.776357e-15
    CPU times base (ref)  : 2.8850 (s)
    CPU times OptV1       : 0.2925 (s) - Speed Up X9.862
    CPU times OptV2       : 0.0152 (s) - Speed Up X189.919
----------------------------
  Test 1 (results): OK
----------------------------
------------------------------------------------------
  Test 2: Validations by integration on [0,1]x[0,1]
------------------------------------------------------
    functions 0 :
        u(x,y,z)=['x - 2*y', 'x + y - z', '3*x + 2*z'],
        v(x,y,z)=['x + 2*y + 4*z', '2*x - y + 4*z', '3*x - 2*y'],
           -> StiffElas error=5.329071e-14
    functions 1 :
        u(x,y,z)=['5*x - 2*y+z', 'x + y - 3*z', '3*x + -2*y+ 2*z'],
        v(x,y,z)=['2*x - 2*y + 4*z +1', '5*x - y + 4*z', '4*x - 2*y+4'],
           -> StiffElas error=2.131628e-14
    functions 2 :
        u(x,y,z)=['x**2 - 2*x*y + x*z', 'y**2 - y*z + z**2 + x', 'x**2 - x*z - y*z - z**2'],
```

```
        v(x,y,z)=['x**2 + 2*y**2 - x*z', '2*x**2 - x*y + y*z', 'x*y - y*z + z**2'],
           -> StiffElas error=5.712000e-02
    functions 3 :
        u(x,y,z)=['x**2 - 2*x*y + x*z', 'x**3 + y**2 - y*z + z**2', '-x**2*z - x*y*z + x**2 - z**2'],
        v(x,y,z)=['-x*z**2 + x**2 + 2*y**2', '2*x**2 - x*y + y*z', 'x*y - y*z + z**2'],
           -> StiffElas error=1.221381e-01
-----------------------------
  Test 2 (results): OK
-----------------------------
--------------------------------
  Test 3: Validations by order
--------------------------------
    functions 3 :
        u(x,y,z)=[x**2 - 2*x*y + x*z,x**3 + y**2 - y*z + z**2,-x**2*z - x*y*z + x**2 - z**2],
        v(x,y)=[-x*z**2 + x**2 + 2*y**2,2*x**2 - x*y + y*z,x*y - y*z + z**2]
      Matrix size                          : (3993,3993)
      StiffElasAssembling3DP1OptV2 CPU times : 0.121(s)
      -> Error                             : 3.036230e-02
      Matrix size                          : (12288,12288)
      StiffElasAssembling3DP1OptV2 CPU times : 0.422(s)
      -> Error                             : 1.515389e-02
      Matrix size                          : (27783,27783)
      StiffElasAssembling3DP1OptV2 CPU times : 0.999(s)
      -> Error                             : 8.314471e-03
      Matrix size                          : (52728,52728)
      StiffElasAssembling3DP1OptV2 CPU times : 2.019(s)
      -> Error                             : 5.618296e-03
      Matrix size                          : (89373,89373)
      StiffElasAssembling3DP1OptV2 CPU times : 3.528(s)
      -> Error                             : 3.666314e-03
      Matrix size                          : (139968,139968)
      StiffElasAssembling3DP1OptV2 CPU times : 5.668(s)
      -> Error                             : 2.932322e-03
```

At last, this program plots the figure :

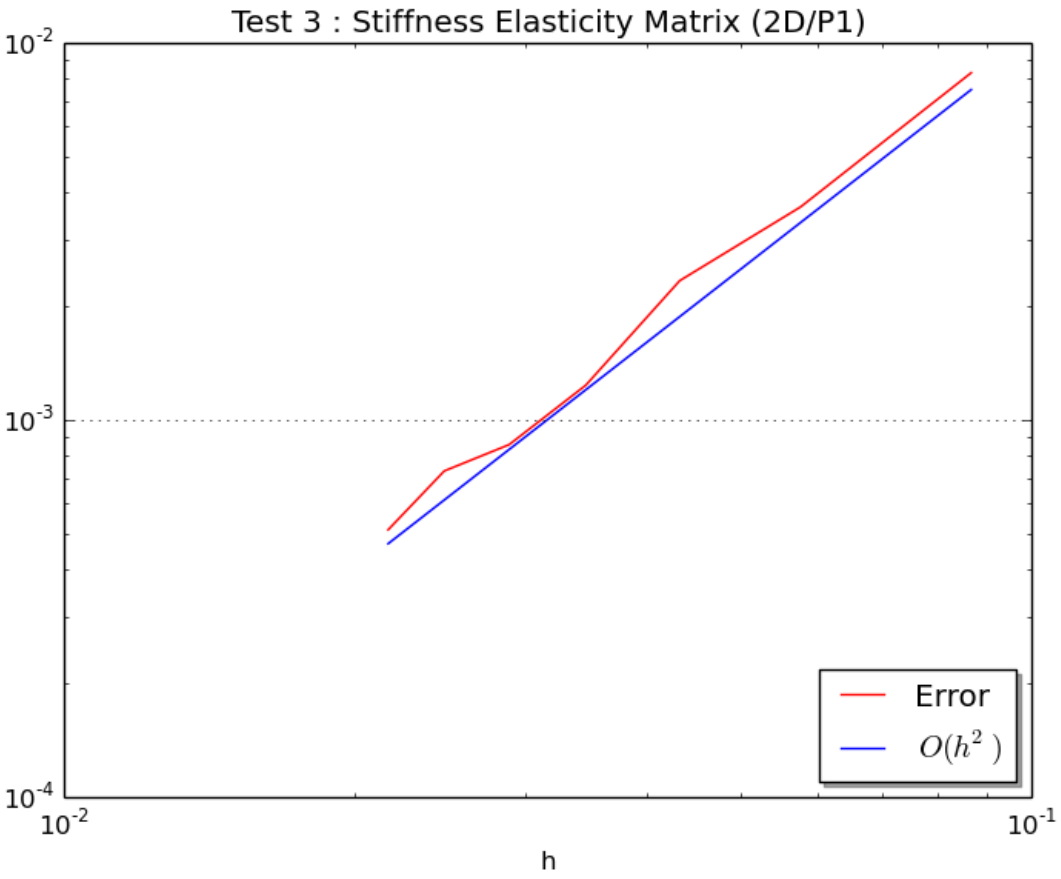


Figure 7.3: Graphical results of validStiffElas3DP1-Test 3

CHAPTER

EIGHT

QUICK TESTING

To test the **pyOptFEM** library, you can run validation tests `runValids()` or some benchmarks (see `runBenchs()`)

CHAPTER

NINE

TESTING AND WORKING

- Ubuntu 12.04 LTS (x86_64) with
  - python3.3
  - python3.2
- Works on Windows 7 64 bits (???)

CHAPTER

TEN

REQUIREMENTS

pyOptFEM works with python3.3.

How to install python3.3.2?

A description of how to compile/install python3.3 under Ubuntu 12.04 LTS (x86_64) is given on François Cuvelier's website, (section Python in the left menu) and repeated here.

Note: How to install python3.3.2

```
wget http://www.python.org/ftp/python/3.3.2/Python-3.3.2.tar.bz2
tar jxvf ./Python-3.3.2.tar.bz2
cd Python-3.3.2
./configure --prefix=/opt/python3.3.2
make
sudo make install
```

As a result, python is installed in `/opt/python3.3.2` directory.

For **pyOptFEM**, you need some python modules. To install them you need easy_install which can be installed like this.

Note: How to install setuptools module

```
wget --check-certificate https://pypi.python.org/packages/source/s/setuptools/setuptools-1.1.4.tar.gz
tar zxvf setuptools-1.1.4.tar.gz
cd setuptools-1.1.4
sudo /opt/python3.3.2/bin/python3 setup.py install
```

For **pyOptFEM**, you need numpy, scipy, sympy and matplotlib. (For matplotlib, the package libfreetype6-dev may be necessary!)

Note: How to install scientific computation modules : numpy, scipy, sympy, matplotlib

```
sudo /opt/python3.3.2/bin/easy_install numpy
sudo /opt/python3.3.2/bin/easy_install scipy
sudo /opt/python3.3.2/bin/easy_install sympy
sudo /opt/python3.3.2/bin/easy_install matplotlib
```

CHAPTER

ELEVEN

INSTALLATION

Note: To install **pyOptFEM** as root, you should run the following command:

```
/opt/python3.3.2/bin/python3 setup.py install
```

Note: To install **pyOptFEM** as a single user, you should run the following command:

```
/opt/python3.3.2/bin/python3 setup.py install --user
```

CHAPTER

TWELVE

LICENSE ISSUES

pyOptFEM is published under the terms of the GNU General Public License.

pyOptFEM is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

pyOptFEM is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program. If not, see http://www.gnu.org/licenses/.

Software using source files of this project or significant parts of it, should include the following attribution notice:

ATTRIBUTION NOTICE: This product includes software developed for the **pyOptFEM** project at (C) University Paris XIII, Galilee Institute, LAGA, France.

pyOptFEM is a python software package for *P_1*-Lagrange Finite Element Methods in 3D. The project is maintained by F. Cuvelier, C. Japhet and G. Scarella. For Online Documentation and Download we refer to

http://www.math.univ-paris13.fr/~cuvelier

CHAPTER

THIRTEEN

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

p

C

E

G

P

S

V