



**HAL**  
open science

## An efficient way to perform the assembly of finite element matrices in vector languages

François Cuvelier, Caroline Japhet, Gilles Scarella

► **To cite this version:**

François Cuvelier, Caroline Japhet, Gilles Scarella. An efficient way to perform the assembly of finite element matrices in vector languages. 2014. hal-00931066v1

**HAL Id: hal-00931066**

**<https://sorbonne-paris-nord.hal.science/hal-00931066v1>**

Preprint submitted on 14 Jan 2014 (v1), last revised 15 Apr 2015 (v2)

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

# AN EFFICIENT WAY TO PERFORM THE ASSEMBLY OF FINITE ELEMENT MATRICES IN VECTOR LANGUAGES

FRANÇOIS CUVELIER\*, CAROLINE JAPHET\*<sup>†</sup>, AND GILLES SCARELLA\*

**Abstract.** We describe different optimization techniques to perform the assembly of finite element matrices in vector languages (e.g. Matlab, Octave, Python), from the standard approach to recent vectorized ones, without any low level language used. We finally obtain a simple and efficient vectorized algorithm. A comparison is given with a compiled language such as C. Unlike what was expected and generally accepted, examples show that the performances of Matlab, Octave and Python are not as far as the ones of C. The principle of this assembly algorithm is general, we present it for a large class of partial differential operators in the scalar or vector cases in 2d and 3d, in the  $\mathcal{P}_1$  finite elements case, with an application to linear elasticity. We present numerical results which illustrate the computational costs of the different approaches.

**Key words.** finite elements, matrix assembly, vectorization, vector languages, Matlab, Octave, Python

**1. Introduction.** Vector languages<sup>1</sup> such as Matlab [22], GNU Octave [13] and Python [23] are very widely used for scientific computing (see for example [25, 20, 17, 19, 2, 14]) and there is significant interest in programming techniques in these languages for two reasons. The first concerns how to make clear, compact code to ease implementation and understanding, which is important for teaching and rapid-prototyping in research and industry. The second concerns how to make the code fast enough for large-scale real-life simulations. A particular strength is that ease of programming can often be combined with high performance if the techniques are of the right type.

In this paper we describe various algorithms for making the assembly of large sparse matrices in finite element computations [24, 27, 3, 4, 18] in vector language (e.g. Matlab, Octave or Python). The last one is fully vectorized which makes use, reading and extension of the code easier while achieving performance close to C. Moreover, we propose quantitative studies to illustrate the efficiency of the vector languages and the various speed-up of the algorithms, relatively to each other and to C.

In a vector language, the loops are the critical performance degrading aspect and removing them is named a vectorization step. In the finite element programming, the classical finite element assembly is based on a loop over the elements (see for example [21]). In [9] T. Davis describes different assembly techniques applied to random matrices of finite element type. A first vectorization technique is proposed in [9]. Other more efficient algorithms in Matlab have been proposed recently in [1, 2, 8, 12, 15, 26].

The description of the fully vectorized algorithm is done in three steps: we recall the classical (non-vectorized) version, then a *first optimized* version, named OptV1, based on sparse matrices tools usually found in most of the languages used for computational science and engineering, and finally our full vectorized algorithm. This new algorithm, named OptV2, is similar to the ideas given in [1, 2, 12] and has been tested for several matrices (e.g. the elastic stiffness matrix) and in different languages. A full generalization including the vectorization of operators  $\mathcal{D}$  and  $\mathcal{H}$  (defined in 2) is the subject of a future paper.

These algorithms can be efficiently implemented in many languages if the language has a sparse matrix implementation. For the OptV1 and OptV2 versions, a particular sparse matrix constructor is also needed (see Section 3). Moreover, the OptV2 version requires that

---

\*Université Paris 13, Sorbonne Paris Cité, LAGA, CNRS UMR 7539, 99 Avenue J-B Clément, F-93430 Villetaneuse, France, cuvelier@math.univ-paris13.fr, japhet@math.univ-paris13.fr, scarella@math.univ-paris13.fr

<sup>†</sup>INRIA Paris-Rocquencourt, BP 105, F-78153 Le Chesnay, France.

This work was partially supported by GNR MoMaS (PACEN/CNRS, ANDRA, BRGM, CEA, EDF, IRSN).

<sup>1</sup>which contain usual element-wise operators and functions on multidimensional arrays

the language supports element-wise array operations. Examples of languages for which we obtained an efficient implementation of these algorithms are

- Matlab,
- Octave,
- Python with *Numpy* and *Scipy* modules,
- Scilab,
- *Thrust*, a C++ template library for CUDA.

Up to now, the implementation in Scilab has been done in 2d and the implementation in *Thrust* has been done for OptV1 and OptV2 algorithms in 2d.

Due to the length of the paper, we restrict ourselves to  $\mathcal{P}_1$  Lagrange finite elements in the scalar and vector cases, in 2d and 3d. However the ideas extend to  $\mathcal{P}_k$  finite elements,  $k \geq 2$ .

The remainder of this article is organized as follows: in Section 2 we introduce the notations associated to the mesh and we define the finite element matrices in a general setting for scalar and vector cases. In Section 3 we give the classical and OptV1 algorithms to perform the assembly of these matrices. Then in Section 4 we present the fully vectorized OptV2 method and show two applications: the assemblies of the stiffness and elastic stiffness matrices in 2d and 3d. In Section 5, benchmarks results illustrate the performance of the algorithms in Matlab, Octave and Python languages. We first show a comparison between the classical, OptV1 and OptV2 versions. Then we compare the performances of the OptV2 version to those obtained with a compiled language (using SuiteSparse [10] in C language), the latter being well-known to run at optimal speed and serving as a reference. A comparison is also given with FreeFEM++ as a simple and reliable finite element software. We also show in Matlab and Octave a comparison of the OptV2 algorithm and the codes given in [1, 2, 15, 26].

All the computations are done on our reference computer<sup>2</sup> with the releases R2012b for Matlab, 3.6.3 for Octave, 3.3.2 for Python and 3.20 for FreeFEM++. The entire Matlab/Octave code and Python code may be found in [6].

**2. Notations.** Let  $\Omega$  be the domain of the PDE problem in  $\mathbb{R}^d$ ,  $d = 2$  or  $3$ . Let  $\mathcal{D}$  be the first order bilinear differential operator acting on *scalar fields* defined by

$$\mathcal{D}(u, v) = \langle \mathbb{A} \nabla u, \nabla v \rangle - (u \langle \mathbf{b}, \nabla v \rangle - v \langle \nabla u, \mathbf{c} \rangle) + fuv, \quad \forall (u, v) \in (H^1(\Omega))^2, \quad (2.1)$$

where  $\mathbb{A} \in (L^\infty(\Omega))^{d \times d}$ ,  $\mathbf{b} \in (L^\infty(\Omega))^d$ ,  $\mathbf{c} \in (L^\infty(\Omega))^d$  and  $f \in L^\infty(\Omega)$  are given functions, and  $\langle \cdot, \cdot \rangle$  is the usual scalar product in  $\mathbb{R}^d$ . We denote by  $\mathcal{H}$  the first order bilinear differential operator acting on *vector fields* defined  $\forall \mathbf{u} = (u_1, \dots, u_d) \in H^1(\Omega)^d$ ,  $\forall \mathbf{v} = (v_1, \dots, v_d) \in H^1(\Omega)^d$ , by

$$\mathcal{H}(\mathbf{u}, \mathbf{v}) = \sum_{\alpha, \beta=1}^d \mathcal{D}^{\alpha, \beta}(u_\alpha, v_\beta), \quad (2.2)$$

where  $(\mathcal{D}^{\alpha, \beta})_{\alpha, \beta \in \{1, \dots, d\}}$  is a set of  $d^2$  first order bilinear differential operators acting on *scalar fields*.

These operators usually appear in variational formulations associated to second order linear partial differential equations in the following integrals

$$\mathcal{L}(u, v) = \int_{\Omega} \mathcal{D}(u, v)(\mathbf{q}) d\mathbf{q} \quad \text{or} \quad \mathcal{L}(\mathbf{u}, \mathbf{v}) = \int_{\Omega} \mathcal{H}(\mathbf{u}, \mathbf{v})(\mathbf{q}) d\mathbf{q}.$$

---

<sup>2</sup>2 x Intel Xeon E5645 (6 cores) at 2.40Ghz, 32Go RAM, supported by GNR MoMaS

For simplicity we use  $\mathcal{P}_1$  Lagrange finite elements to approximate these integrals.

We suppose that  $\Omega$  is provided with its mesh  $\mathcal{T}_h$  (classical and locally conforming). We consider that the set belonging to the mesh is of simplicial type (triangles or tetrahedra). We use a triangulation  $\Omega_h = \bigcup_{T_k \in \mathcal{T}_h} T_k$  of  $\Omega$  described by :

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

We now define a general class of finite element matrices whose assembly can be done efficiently with the optimized algorithm proposed hereafter.

**2.1. The scalar case.** 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 basis functions  $\{\varphi_i\}_{i \in \{1, \dots, n_q\}}$  in  $\mathbb{R}^d$ , where  $\mathcal{P}_1(T)$  denotes the space of all polynomials defined over  $T$  of total degree less than or equal to 1. With  $(u, v) \in (H^1(\Omega))^2$ , we have the  $\mathcal{P}_1$  Lagrange finite elements approximation

$$\mathcal{L}(u, v) = \int_{\Omega} \mathcal{D}(u, v)(q) dq \approx \langle \mathbb{D} \mathbf{U}, \mathbf{V} \rangle,$$

where  $\mathbf{U} = (u(q^i))_{i=1}^{n_q}$  and  $\mathbf{V} = (v(q^i))_{i=1}^{n_q}$  are  $n_q$ -by-1 vectors and  $\mathbb{D}$  is the  $n_q$ -by- $n_q$  finite element matrix defined by

$$\mathbb{D}_{i,j} = \int_{\Omega_h} \mathcal{D}(\varphi_j, \varphi_i)(q) dq, \quad \forall (i, j) \in \{1, \dots, n_q\}^2. \quad (2.3)$$

Let  $n_{dfe} = d+1$  be the local number of degrees of freedom for each mesh element. We introduce the  $n_{dfe}$ -by- $n_{dfe}$  local matrix  $\mathbb{D}_{\alpha, \beta}^e(T_k)$ , associated to  $\mathbb{D}$ , defined by

$$\mathbb{D}_{\alpha, \beta}^e(T_k) = \int_{T_k} \mathcal{D}(\varphi_{\beta}^k, \varphi_{\alpha}^k)(q) dq, \quad \forall (\alpha, \beta) \in \{1, \dots, n_{dfe}\}^2,$$

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

For simplicity, we will consider as an example the stiffness matrix (e.g. arising from the discretization of the Poisson equation), obtained from (2.1) with  $\mathbb{A} = \mathbb{I}$ ,  $\mathbf{b} = \mathbf{c} = 0$ , and  $f = 0$ . This matrix, denoted by  $\mathbb{S}$ , is defined by:

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

The matrix  $\mathbb{S}$  is a  $n_q$ -by- $n_q$  sparse matrix. Its corresponding local stiffness matrix, denoted by  $\mathbb{S}^e$ , is defined by:  $\forall (\alpha, \beta) \in \{1, \dots, n_{dfe}\}^2$ ,

$$\mathbb{S}_{\alpha, \beta}^e(T_k) = \int_{T_k} \mathcal{D}(\varphi_{\beta}^k, \varphi_{\alpha}^k)(q) dq = \int_{T_k} \langle \nabla \varphi_{\beta}^k(q), \nabla \varphi_{\alpha}^k(q) \rangle dq. \quad (2.4)$$

**2.2. The vector case.** The space  $(H_h^1(\Omega_h))^d$  is of dimension  $n_{\text{dof}} = d n_q$ . One can choose the *alternate* basis spanned by the functions  $\{\boldsymbol{\Psi}_l\}_{1 \leq l \leq n_{\text{dof}}}$ , defined by

$$\boldsymbol{\Psi}_{di-\alpha} = \boldsymbol{\varphi}_i (\delta_{d-1,\alpha}, \dots, \delta_{0,\alpha})^t, \quad \forall \alpha \in \{0, \dots, d-1\}, \forall i \in \{1, \dots, n_q\}. \quad (2.5)$$

Let  $(\mathbf{u}, \mathbf{v}) \in (H^1(\Omega)^d)^2$ . We denote by  $\mathbf{U}$  and  $\mathbf{V}$  the  $n_{\text{dof}}$ -by-1 vectors defined by  $U_{di-(\alpha-1)} = u_\alpha(\mathbf{q}^i)$ ,  $V_{di-(\alpha-1)} = v_\alpha(\mathbf{q}^i)$ ,  $\forall i \in \{1, \dots, n_q\}$ ,  $\forall \alpha \in \{1, \dots, d\}$ . Then, we have the  $\mathcal{P}_1$  Lagrange finite element approximation

$$\mathcal{L}(\mathbf{u}, \mathbf{v}) = \int_{\Omega} \mathcal{H}(\mathbf{u}, \mathbf{v})(\mathbf{q}) d\mathbf{q} \approx \langle \mathbb{H} \mathbf{U}, \mathbf{V} \rangle,$$

where  $\mathbb{H}$  is the  $n_{\text{dof}}$ -by- $n_{\text{dof}}$  finite element matrix defined by

$$\mathbb{H}_{l,m} = \int_{\Omega_h} \mathcal{H}(\boldsymbol{\Psi}_m, \boldsymbol{\Psi}_l)(\mathbf{q}) d\mathbf{q}, \quad \forall (l, m) \in \{1, \dots, n_{\text{dof}}\}^2.$$

Let  $n_{\text{dfe}} = d(d+1)$ . The  $n_{\text{dfe}}$ -by- $n_{\text{dfe}}$  matrix  $\mathbb{H}_{\alpha,\beta}^e(T_k)$ , associated to  $\mathbb{H}$ , is defined by

$$\mathbb{H}_{\alpha,\beta}^e(T_k) = \int_{T_k} \mathcal{H}(\boldsymbol{\Psi}_\beta^k, \boldsymbol{\Psi}_\alpha^k)(\mathbf{q}) d\mathbf{q}, \quad \forall (\alpha, \beta) \in \{1, \dots, n_{\text{dfe}}\}^2, \quad (2.6)$$

where  $\boldsymbol{\Psi}_\alpha^k = \boldsymbol{\Psi}_{\mathcal{J}_\alpha^k}$  is the  $\alpha$ -th local basis function associated to the  $k$ -th element with  $\mathcal{J}_{da-b}^k = d \text{me}(a, k) - b$ ,  $\forall a \in \{1, \dots, d+1\}$ ,  $\forall b \in \{0, \dots, d-1\}$ .

For simplicity, we will consider as an example the elastic stiffness matrix arising in linear elasticity when Hooke's law is used and the material is isotropic, under small strain hypothesis (see for example [11]). Thus, for a sufficiently regular vector field  $\mathbf{u} = (u_1, \dots, u_d) : \Omega \rightarrow \mathbb{R}^d$ , we define the linearized strain tensor  $\underline{\boldsymbol{\varepsilon}}$  by

$$\underline{\boldsymbol{\varepsilon}}(\mathbf{u}) = \frac{1}{2} (\nabla(\mathbf{u}) + \nabla^t(\mathbf{u})).$$

We set  $\underline{\boldsymbol{\varepsilon}} = (\varepsilon_{11}, \varepsilon_{22}, 2\varepsilon_{12})^t$  in 2d and  $\underline{\boldsymbol{\varepsilon}} = (\varepsilon_{11}, \varepsilon_{22}, \varepsilon_{33}, 2\varepsilon_{12}, 2\varepsilon_{23}, 2\varepsilon_{13})^t$  in 3d, with  $\varepsilon_{ij}(\mathbf{u}) = \frac{1}{2} \left( \frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i} \right)$ . Then the Hooke's law reads

$$\underline{\boldsymbol{\sigma}} = \mathbb{C} \underline{\boldsymbol{\varepsilon}},$$

where  $\underline{\boldsymbol{\sigma}}$  is the elastic stress tensor and  $\mathbb{C}$  the elasticity tensor defined by the Lamé parameters  $\lambda$  and  $\mu$  (supposed constant on  $\Omega$  for simplicity), which satisfy  $\lambda + \mu > 0$ . For  $d = 2$  or  $d = 3$ ,  $\mathbb{C}$  is given by

$$\mathbb{C} = \begin{pmatrix} \lambda \mathbb{1}_2 + 2\mu \mathbb{I}_2 & 0 \\ 0 & \mu \end{pmatrix}_{3 \times 3} \quad \text{or} \quad \mathbb{C} = \begin{pmatrix} \lambda \mathbb{1}_3 + 2\mu \mathbb{I}_3 & 0 \\ 0 & \mu \mathbb{I}_3 \end{pmatrix}_{6 \times 6},$$

respectively, where  $\mathbb{1}_d$  is a  $d$ -by- $d$  matrix of ones, and  $\mathbb{I}_d$  the  $d$ -by- $d$  identity matrix. Then the elastic stiffness matrix  $\mathbb{K}$  is defined by

$$\mathbb{K}_{l,m} = \int_{\Omega_h} \underline{\boldsymbol{\varepsilon}}^t(\boldsymbol{\Psi}_m) \mathbb{C} \underline{\boldsymbol{\varepsilon}}(\boldsymbol{\Psi}_l) d\mathbf{q}, \quad \forall (l, m) \in \{1, \dots, n_{\text{dof}}\}^2. \quad (2.7)$$

Details of obtaining the matrix  $\mathbb{K}$  from the operator  $\mathcal{H}$  are given in Appendix A.2. The corresponding local elastic stiffness matrix, denoted by  $\mathbb{K}^e$ , is defined for all  $(\alpha, \beta) \in \{1, \dots, n_{\text{dfe}}\}^2$  by

$$\mathbb{K}_{\alpha,\beta}^e(T_k) = \int_{T_k} \underline{\boldsymbol{\varepsilon}}^t(\boldsymbol{\Psi}_\beta^k) \mathbb{C} \underline{\boldsymbol{\varepsilon}}(\boldsymbol{\Psi}_\alpha^k) d\mathbf{q} = \int_{T_k} \mathcal{H}(\boldsymbol{\Psi}_\beta^k, \boldsymbol{\Psi}_\alpha^k)(\mathbf{q}) d\mathbf{q}. \quad (2.8)$$

In the scalar or vector cases, by approximating the  $L^\infty(\Omega)$  functions arising in  $\mathcal{D}$  and  $\mathcal{H}$  by functions in  $H_h^1(\Omega_h)$ , and using the following (well-known) formulas:

$$\int_T \varphi_1^m \varphi_2^m \varphi_3^p dT = \frac{2!|T|m!n!p!}{(2+m+n+p)!}, \quad d=2, \quad (2.9)$$

$$\int_T \varphi_1^m \varphi_2^m \varphi_3^p \varphi_4^q dT = \frac{3!|T|m!n!p!q!}{(3+m+n+p+q)!}, \quad d=3, \quad (2.10)$$

the computation of the local matrices associated to  $\mathbb{D}$  and  $\mathbb{H}$  can be done explicitly without any quadrature formula. Remind that these matrices are sparse, due to the basis functions properties.

**REMARK 2.1.** *All the (non-vectorized or vectorized) finite element assembly algorithms presented in this article can be used to compute matrices of  $\mathbb{D}$  (or  $\mathbb{H}$ ) type. They also apply to  $P_k$ -Lagrange finite element methods (i.e. finite element matrices obtained from bilinear differential operators of order  $k$ ). Indeed, one can express the  $P_k$ -Lagrange basis functions versus the  $P_1$ -Lagrange basis functions and then use the formulas (2.9) or (2.10).*

**3. Classical and first optimized finite element assemblies.** In this part we consider the finite element assembly of a generic  $n_{\text{dof}}$ -by- $n_{\text{dof}}$  sparse matrix  $\mathbb{M}$  with its corresponding  $n_{\text{dfe}}$ -by- $n_{\text{dfe}}$  local matrix  $\mathbb{E}$  (also denoted by  $\mathbb{E}(T_k)$  when referring to a specific element  $T_k$ ). An element of  $\mathbb{E}(T_k)$  is denoted by  $e_{\alpha,\beta}^k$ .

In Algorithm 1<sup>3</sup>, we recall the classical finite element assembly method to calculate  $\mathbb{M}$ . In this algorithm a  $n_{\text{dof}}$ -by- $n_{\text{dof}}$  sparse matrix  $\mathbb{M}$  is first declared, then each contribution of element  $T_k$  is added to the matrix  $\mathbb{M}$ . These successive operations are very expensive. Thus, we compute and store all elementary contributions and use them to generate the sparse matrix  $\mathbb{M}$ . This is the purpose of the first optimized algorithm, already suggested in [9], and shown in Algorithm 2.

---

**Algorithm 1** Classical assembly - base version

```

1:  $\mathbb{M} \leftarrow \mathbb{O}_{n_{\text{dof}} \times n_{\text{dof}}}$  ▷ Sparse matrix
2: for  $k \leftarrow 1$  to  $n_{\text{me}}$  do
3:    $\mathbb{E} \leftarrow \text{ElemMat}(\text{vol}(k), \dots)$ 
4:   for  $il \leftarrow 1$  to  $n_{\text{dfe}}$  do
5:      $i \leftarrow \mathcal{I}^k(il)$ 
6:     for  $jl \leftarrow 1$  to  $n_{\text{dfe}}$  do
7:        $j \leftarrow \mathcal{I}^k(jl)$ 
8:        $\mathbb{M}_{i,j} \leftarrow \mathbb{M}_{i,j} + \mathbb{E}_{il,jl}$ 
9:     end for
10:  end for
11: end for

```

---



---

**Algorithm 2** Optimized assembly - OptV1 version

```

1:  $dim \leftarrow n_{\text{dfe}}^2 n_{\text{me}}$ 
2:  $\mathbf{I}_g \leftarrow \mathbb{O}_{dim \times 1}$  ▷ 1d array of size dim
3:  $\mathbf{J}_g \leftarrow \mathbb{O}_{dim \times 1}$  ▷ 1d array of size dim
4:  $\mathbf{K}_g \leftarrow \mathbb{O}_{dim \times 1}$  ▷ 1d array of size dim
5:  $ii \leftarrow [1 : n_{\text{dfe}}]^t * \mathbb{1}_{1 \times n_{\text{dfe}}}; ii \leftarrow ii(:);$ 
6:  $jj \leftarrow \mathbb{1}_{n_{\text{dfe}} \times 1} * [1 : n_{\text{dfe}}]; jj \leftarrow jj(:)$ 
7:  $kk \leftarrow [1 : n_{\text{dfe}}^2]$ 
8: for  $k \leftarrow 1$  to  $n_{\text{me}}$  do
9:    $\mathbb{E} \leftarrow \text{ElemMat}(\text{vol}(k), \dots)$ 
10:   $\mathbf{I}_g(kk) \leftarrow \mathcal{I}^k(ii)$ 
11:   $\mathbf{J}_g(kk) \leftarrow \mathcal{I}^k(jj)$ 
12:   $\mathbf{K}_g(kk) \leftarrow \mathbb{E}(:)$ 
13:   $kk \leftarrow kk + n_{\text{dfe}}^2$ 
14: end for
15:  $\mathbb{M} \leftarrow \text{sparse}(\mathbf{I}_g, \mathbf{J}_g, \mathbf{K}_g, n_q, n_q)$ 

```

---

This optimized version (named “version 1” or OptV1) is non-vectorized and based on the use of the sparse function as follows:

<sup>3</sup>The functions and operators used in the algorithms are given in Appendix C.

$$M \leftarrow \text{sparse}(\text{Ig}, \text{Jg}, \text{Kg}, m, n);$$

This command returns a  $m \times n$  sparse matrix  $M$  such that  $M(\text{Ig}(k), \text{Jg}(k)) \leftarrow \text{Kg}(k)$ . The vectors  $\text{Ig}$ ,  $\text{Jg}$  and  $\text{Kg}$  have the same length. The zero elements of  $\text{Kg}$  are not taken into account and the elements of  $\text{Kg}$  having the same indices in  $\text{Ig}$  and  $\text{Jg}$  are summed.

Examples of vector languages and their corresponding sparse function are:

- Python (*scipy.sparse* module) :  

$$M = \text{sparse}.\langle \text{format} \rangle.\text{matrix}((\text{Kg}, (\text{Ig}, \text{Jg})), \text{shape}=(m, n))$$
 where  $\langle \text{format} \rangle$  is the sparse matrix format (e.g. *csc*, *csr*, *lil*, ...),
- Matlab :  $M = \text{sparse}(\text{Ig}, \text{Jg}, \text{Kg}, m, n)$ , only *csc* format,
- Octave :  $M = \text{sparse}(\text{Ig}, \text{Jg}, \text{Kg}, m, n)$ , only *csc* format,
- Scilab :  $M = \text{sparse}([\text{Ig}, \text{Jg}], \text{Kg}, [m, n])$ , only row-by-row format.

In compiled languages, there are some libraries with such functions. For example, in C language one can use the *SuiteSparse* [10] developed by T. Davis. With a Nvidia GPU, one can use the *Thrust* [28] and *Cusp* [5] libraries for vectorization and sparse computations respectively.

In the OptV1 version, the main idea is to create three global 1d-arrays  $\mathbf{K}_g$ ,  $\mathbf{I}_g$  and  $\mathbf{J}_g$  allowing the storage of the local matrices as well as the position of their elements in the global matrix. Thus, the size of these three arrays is  $n_{\text{dfe}}^2 n_{\text{me}} - 1$ . These arrays will be used in the *sparse* function to obtain the global matrix. To create the arrays  $\mathbf{K}_g$ ,  $\mathbf{I}_g$  and  $\mathbf{J}_g$ , we define three local arrays  $\mathbf{K}_k^e$ ,  $\mathbf{I}_k^e$  and  $\mathbf{J}_k^e$  of length  $m = n_{\text{dfe}}^2$  obtained from the  $n_{\text{dfe}}$ -by- $n_{\text{dfe}}$  local matrix  $\mathbb{E}(T_k)$  as follows:

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

Using  $\mathbf{K}_k^e$ ,  $\mathbf{I}_k^e$ ,  $\mathbf{J}_k^e$  and a loop over all the mesh elements  $T_k$ , one can calculate the  $n_{\text{dfe}}^2 n_{\text{me}} - 1$  global arrays  $\mathbf{I}_g$ ,  $\mathbf{J}_g$  and  $\mathbf{K}_g$ . These operations are illustrated in Figure 3.1 and the corresponding OptV1 algorithm is given in Algorithm 2. The functions and operators used in the algorithms proposed in this article are given in Appendix C.

In Section 5.1 and in Tables B.1 and B.2 we show that the OptV1 algorithm is more efficient than the classical algorithm. The inefficiency of the base version compared to the OptV1 version is mainly due to the repetition of elements insertions in the sparse structure and to some dynamic reallocation troubles that may also occur.

However, the OptV1 algorithm still uses a loop over elements. To improve the efficiency of this algorithm, we propose in the next section a second optimized version, in a fully vectorized form.

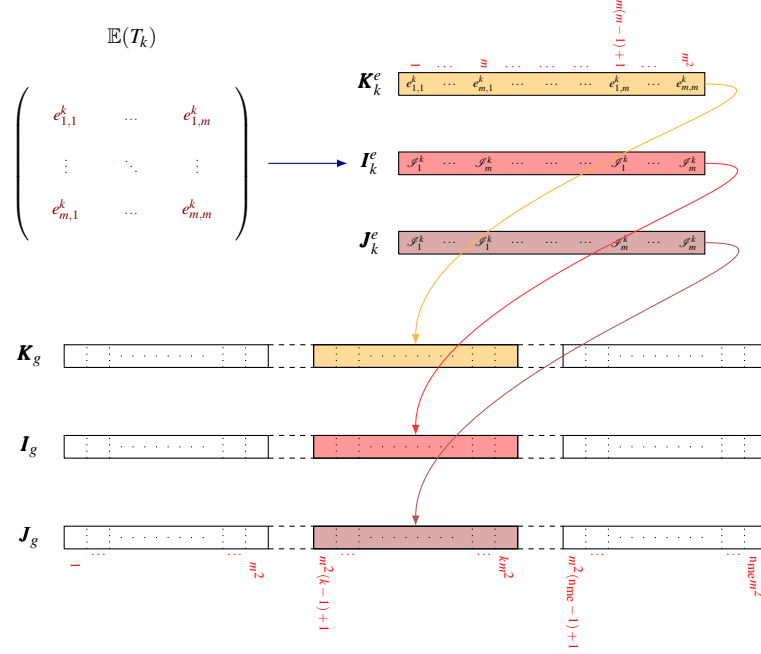


Fig. 3.1: Classical insertion of a local matrix in global arrays - Version 1

**4. Optimized finite element assembly.** In this part we present the optimized version 2 (OptV2) algorithm, only available in vector languages. The idea is to define three  $n_{\text{dof}}^2$ -by- $n_{\text{me}}$  arrays  $\mathbb{K}_g$ ,  $\mathbb{I}_g$  and  $\mathbb{J}_g$  that allow to store all the local matrices as well as their positions in the global matrix. Then, as for the OptV1 version, the matrix assembly is obtained with the sparse function:

$$\mathbb{M} \leftarrow \text{sparse}(\mathbb{I}_g(\cdot), \mathbb{J}_g(\cdot), \mathbb{K}_g(\cdot), n_{\text{dof}}, n_{\text{dof}});$$

A non-vectorized approach, inspired from OptV1, is as follows: for each mesh element  $T_k$ , one adds the local arrays  $\mathbf{K}_k^e$ ,  $\mathbf{I}_k^e$ ,  $\mathbf{J}_k^e$  defined above in the  $k$ -th column of the global arrays  $\mathbb{K}_g$ ,  $\mathbb{I}_g$  and  $\mathbb{J}_g$  respectively. This method is shown in Figure 4.1. Thus,  $\mathbb{K}_g$ ,  $\mathbb{I}_g$  and  $\mathbb{J}_g$  are defined by:  $\forall k \in \{1, \dots, n_{\text{me}}\}, \forall l \in \{1, \dots, n_{\text{dof}}^2\}$ ,

$$\mathbb{K}_g(l, k) = \mathbf{K}_k^e(l), \quad \mathbb{I}_g(l, k) = \mathbf{I}_k^e(l), \quad \mathbb{J}_g(l, k) = \mathbf{J}_k^e(l). \quad (4.1)$$

A natural way to calculate these three arrays is column-wise. In that case, for each array one needs to calculate  $n_{\text{me}}$  columns (where  $n_{\text{me}}$  depends on the number of mesh elements). The OptV2 method consists in calculating these arrays row-wise. In that case, for each array one needs to calculate  $n_{\text{dof}}$  rows (where  $n_{\text{dof}}$  is independent of the number of mesh elements). This vectorization method is represented in Figure 4.2.



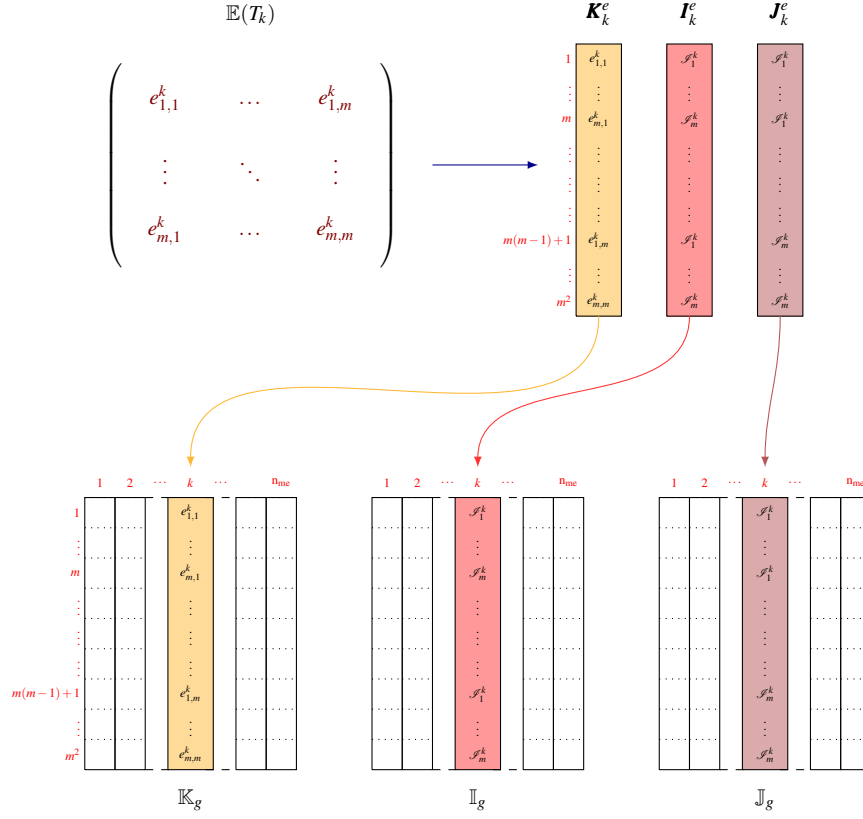


Fig. 4.1: Classical insertion of a local matrix in 2d global arrays

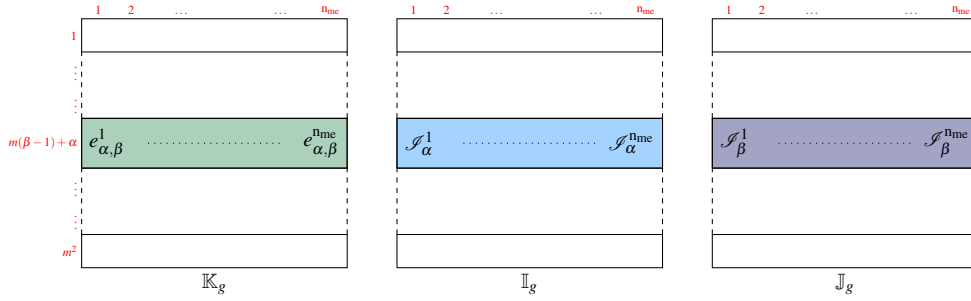


Fig. 4.2: Row-wise operations on 2d global arrays

The calculation of  $\mathbb{K}_g$ ,  $\mathbb{I}_g$  and  $\mathbb{J}_g$  uses also the following equalities with  $m = n_{dfe}$ , for all  $(\alpha, \beta) \in \{1, \dots, n_{dfe}\}$ ,

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

where  $\mathcal{S}_{\alpha}^k$  is defined in Sections 2.1 and 2.2 in the scalar and vector cases respectively. The calculation of  $\mathbb{I}_g$  and  $\mathbb{J}_g$  can be vectorized using a row-wise computation (see Algorithms 3

to 6). In the vector case, we calculate a matrix  $\mathbb{T}$  such that  $\mathbb{T}(\alpha, k) = \mathcal{S}_\alpha^k, \forall \alpha \in \{1, \dots, n_{\text{dfe}}\}$ .

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

```

function [ $\mathbb{I}_g, \mathbb{J}_g$ ]  $\leftarrow$  BUILDIGJG2D(me)
   $ii \leftarrow [1, 2, 3, 1, 2, 3, 1, 2, 3]$ 
   $jj \leftarrow [1, 1, 1, 2, 2, 2, 3, 3, 3]$ 
   $\mathbb{I}_g \leftarrow \text{me}(ii, :)$   $\triangleright 9 \times n_{\text{me}}$  array
   $\mathbb{J}_g \leftarrow \text{me}(jj, :)$   $\triangleright 9 \times n_{\text{me}}$  array
end function

```

**Algorithm 4**  $\mathbb{I}_g$  and  $\mathbb{J}_g$ : the scalar case in 3d

```

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

```

**Algorithm 5**  $\mathbb{I}_g$  and  $\mathbb{J}_g$ : the vector case in 2d

```

function [ $\mathbb{I}_g, \mathbb{J}_g$ ]  $\leftarrow$  BUILDIGJG2DVF(me)
   $\mathbb{T} \leftarrow \begin{pmatrix} 2 \text{me}(1, :) - 1 \\ 2 \text{me}(1, :) \\ 2 \text{me}(2, :) - 1 \\ 2 \text{me}(2, :) \\ 2 \text{me}(3, :) - 1 \\ 2 \text{me}(3, :) \end{pmatrix} \triangleright 6 \times n_{\text{me}}$ 
   $ii \leftarrow [1 : 6]^t * \mathbb{1}_{1 \times 6}; ii \leftarrow ii(:);$ 
   $jj \leftarrow \mathbb{1}_{6 \times 1} * [1 : 6]; jj \leftarrow jj(:)$ 
   $\mathbb{I}_g \leftarrow \mathbb{T}(ii, :)$   $\triangleright 36 \times n_{\text{me}}$  array
   $\mathbb{J}_g \leftarrow \mathbb{T}(jj, :)$   $\triangleright 36 \times n_{\text{me}}$  array
end function

```

**Algorithm 6**  $\mathbb{I}_g$  and  $\mathbb{J}_g$ : the vector case in 3d

```

function [ $\mathbb{I}_g, \mathbb{J}_g$ ]  $\leftarrow$  BUILDIGJG3DVF(me)
   $\mathbb{T} \leftarrow \begin{pmatrix} 3 \text{me}(1, :) - 2 \\ 3 \text{me}(1, :) - 1 \\ 3 \text{me}(1, :) \\ \vdots \\ 3 \text{me}(4, :) - 2 \\ 3 \text{me}(4, :) - 1 \\ 3 \text{me}(4, :) \end{pmatrix} \triangleright 12 \times n_{\text{me}}$ 
   $ii \leftarrow [1 : 12]^t * \mathbb{1}_{1 \times 12}; ii \leftarrow ii(:);$ 
   $jj \leftarrow \mathbb{1}_{12 \times 1} * [1 : 12]; jj \leftarrow jj(:)$ 
   $\mathbb{I}_g \leftarrow \mathbb{T}(ii, :)$   $\triangleright 144 \times n_{\text{me}}$  array
   $\mathbb{J}_g \leftarrow \mathbb{T}(jj, :)$   $\triangleright 144 \times n_{\text{me}}$  array
end function

```

Then, the computation of  $\mathbb{K}_g$  (containing all the local matrices associated to  $\mathbb{D}$  or  $\mathbb{H}$ ) is vectorized using row-wise vector operations. It is based on the calculation of the terms  $\mathcal{D}(\varphi_\beta^k, \varphi_\alpha^k)$  and  $\mathcal{H}(\Psi_\beta^k, \Psi_\alpha^k)$  which arise in the local matrices (2.3) and (2.6) respectively. This calculation only depends on the local basis functions  $\varphi_\beta^k, \forall \beta \in \{1, \dots, d+1\}$  and their first order derivatives. Thus, one need to calculate all or part of the gradients of the basis functions on all mesh elements. In the  $\mathcal{P}_1$  finite elements case, these gradients are constant on each element  $T_k$  and denoted by  $\mathbf{G}^\alpha, \forall \alpha \in \{1, \dots, d+1\}$ . The  $d$ -by- $n_{\text{me}}$  array  $\mathbf{G}^\alpha$  is thus defined by

$$\mathbf{G}^\alpha(:, k) = \nabla \varphi_{\text{me}(\alpha, k)}(\mathbf{q}), \forall \mathbf{q} \in T_k, \forall k \in \{1, \dots, n_{\text{me}}\}. \quad (4.2)$$

Vectorized algorithms are given in Appendix A.1 to calculate these gradients in 2d and 3d. Once the gradients are computed, the local matrices are calculated explicitly using the formulas (2.9) and (2.10). For simplicity, we show below the vectorized computation of the 2d-array  $\mathbb{K}_g$  on the two examples introduced in Section 2.

**4.1. Stiffness matrix assembly.** Using (4.2) and the definition of the local stiffness matrix given in (2.4), the array  $\mathbb{K}_g$  is defined by

$$\begin{aligned} \mathbb{K}_g((d+1)(i-1) + j, k) &= |T_k| \left\langle \nabla \varphi_{\text{me}(j, k)}, \nabla \varphi_{\text{me}(i, k)} \right\rangle, \quad 1 \leq i, j \leq d+1, 1 \leq k \leq n_{\text{me}}, \\ &= |T_k| \left\langle \mathbf{G}^j(:, k), \mathbf{G}^i(:, k) \right\rangle. \end{aligned}$$

The calculation of  $\mathbb{K}_g$  uses the vectorized functions BUILDIGJG (see Algorithms 3 and 4) and GRADIENTVEC (see Algorithms 12 and 13 in Appendix A.1). The fully vectorized

algorithms that calculate  $\mathbb{K}_g$  are given in Algorithm 7, lines 4 to 11, in 2d and in Algorithm 8, lines 4 to 15, in 3d respectively.

---

**Algorithm 7** Vectorized assembly algorithm for the 2d stiffness matrix - version 2

---

```

1: function  $\mathbb{S} \leftarrow \text{STIFFASSEMBLING2DP1OPTV2}(n_q, n_{me}, q, me, vol)$ 
2:    $[\mathbf{G}^1, \mathbf{G}^2, \mathbf{G}^3] \leftarrow \text{GRADIENTVEC2D}(q, me, vol)$ 
3:    $[\mathbb{I}_g, \mathbb{J}_g] \leftarrow \text{BUILDIGJG2D}(me)$ 
4:    $\mathbb{K}_g \leftarrow \mathbf{0}_{9 \times n_{me}}$ 
5:    $\mathbb{K}_g(1, :) \leftarrow \text{sum}(\mathbf{G}^1 * \mathbf{G}^1, 1) * vol$ 
6:    $\mathbb{K}_g(2, :) \leftarrow \text{sum}(\mathbf{G}^2 * \mathbf{G}^1, 1) * vol$ 
7:    $\mathbb{K}_g(3, :) \leftarrow \text{sum}(\mathbf{G}^3 * \mathbf{G}^1, 1) * vol$ 
8:    $\mathbb{K}_g(5, :) \leftarrow \text{sum}(\mathbf{G}^2 * \mathbf{G}^2, 1) * vol$ 
9:    $\mathbb{K}_g(6, :) \leftarrow \text{sum}(\mathbf{G}^3 * \mathbf{G}^2, 1) * vol$ 
10:   $\mathbb{K}_g(9, :) \leftarrow \text{sum}(\mathbf{G}^3 * \mathbf{G}^3, 1) * vol$ 
11:   $\mathbb{K}_g([4, 7, 8], :) \leftarrow \mathbb{K}_g([2, 3, 6], :)$ 
12:   $\mathbb{S} \leftarrow \text{SPARSE}(\mathbb{I}_g(:), \mathbb{J}_g(:), \mathbb{K}_g(:), n_q, n_q)$ 
13: end function

```

---

**Algorithm 8** Vectorized assembly algorithm for the 3d stiffness matrix - version 2

---

```

1: function  $\mathbb{S} \leftarrow \text{STIFFASSEMBLING3DP1OPTV2}(n_q, n_{me}, q, me, vol)$ 
2:    $[\mathbf{G}^1, \mathbf{G}^2, \mathbf{G}^3, \mathbf{G}^4] \leftarrow \text{GRADIENTVEC3D}(q, me, vol)$ 
3:    $[\mathbb{I}_g, \mathbb{J}_g] \leftarrow \text{BUILDIGJG3D}(me)$ 
4:    $\mathbb{K}_g \leftarrow \mathbf{0}_{16 \times n_{me}}$ 
5:    $\mathbb{K}_g(1, :) \leftarrow \text{sum}(\mathbf{G}^1 * \mathbf{G}^1, 1) * vol$ 
6:    $\mathbb{K}_g(2, :) \leftarrow \text{sum}(\mathbf{G}^2 * \mathbf{G}^1, 1) * vol$ 
7:    $\mathbb{K}_g(3, :) \leftarrow \text{sum}(\mathbf{G}^3 * \mathbf{G}^1, 1) * vol$ 
8:    $\mathbb{K}_g(4, :) \leftarrow \text{sum}(\mathbf{G}^4 * \mathbf{G}^1, 1) * vol$ 
9:    $\mathbb{K}_g(6, :) \leftarrow \text{sum}(\mathbf{G}^2 * \mathbf{G}^2, 1) * vol$ 
10:   $\mathbb{K}_g(7, :) \leftarrow \text{sum}(\mathbf{G}^2 * \mathbf{G}^3, 1) * vol$ 
11:   $\mathbb{K}_g(8, :) \leftarrow \text{sum}(\mathbf{G}^2 * \mathbf{G}^4, 1) * vol$ 
12:   $\mathbb{K}_g(11, :) \leftarrow \text{sum}(\mathbf{G}^3 * \mathbf{G}^3, 1) * vol$ 
13:   $\mathbb{K}_g(12, :) \leftarrow \text{sum}(\mathbf{G}^3 * \mathbf{G}^4, 1) * vol$ 
14:   $\mathbb{K}_g(16, :) \leftarrow \text{sum}(\mathbf{G}^4 * \mathbf{G}^4, 1) * vol$ 
15:   $\mathbb{K}_g([5, 9, 10, 13, 14, 15], :) \leftarrow \mathbb{K}_g([2, 3, 7, 4, 8, 12], :)$ 
16:   $\mathbb{S} \leftarrow \text{SPARSE}(\mathbb{I}_g(:), \mathbb{J}_g(:), \mathbb{K}_g(:), n_q, n_q)$ 
17: end function

```

---

**4.2. Elastic stiffness matrix assembly.** The local elastic stiffness matrix is given in (2.8) with  $n_{dfe} = d(d+1)$ . For simplicity, let us consider the case  $d = 2$ , and the calculation of the first two terms in the first column of  $\mathbb{K}^e(T_k)$  in (2.8). Then, using the expression of  $\mathcal{H}$  given in Appendix A.2, we obtain, for all  $k \in \{1, \dots, n_{me}\}$ ,

$$\mathbb{K}_{1,1}^e(T_k) = |T_k| \left( (\lambda + 2\mu) \left( \frac{\partial \varphi_1^k}{\partial x} \right)^2 + \mu \left( \frac{\partial \varphi_1^k}{\partial y} \right)^2 \right), \quad \mathbb{K}_{2,1}^e(T_k) = |T_k| (\lambda + \mu) \frac{\partial \varphi_1^k}{\partial x} \frac{\partial \varphi_1^k}{\partial y},$$

that is (using the array notations defined above),

$$\begin{aligned} \mathbb{K}_g(1, k) &= \mathbb{K}_{1,1}^e(T_k) = |T_k| \left( (\lambda + 2\mu) \mathbf{G}^1(1, k)^2 + \mu \mathbf{G}^1(2, k)^2 \right), \\ \mathbb{K}_g(2, k) &= \mathbb{K}_{2,1}^e(T_k) = |T_k| (\lambda + \mu) \mathbf{G}^1(1, k) \mathbf{G}^1(2, k). \end{aligned}$$

The vectorized calculation of the first two rows of  $\mathbb{K}_g$  is shown in Algorithm 9, lines 5 to 6. The other rows of  $\mathbb{K}_g$  are computed using the same method and using the symmetry of the matrix. We present in Algorithm 9 the fully vectorized assembly algorithm of the elastic stiffness matrix in 2d (in this algorithm, we don't show all the terms of  $\mathbb{K}_g$ ; for the complete function in Matlab/octave see [7]). The same method is used to perform the assembly of the elastic stiffness matrix in 3d (see [7]).

---

**Algorithm 9** Vectorized assembly for 2d elasticity stiffness matrix - version 2

---

```

1: function  $\mathbb{K} \leftarrow \text{STIFFELAS2DP1OPTV2}(n_q, n_{me}, q, me, vol, \lambda, \mu)$ 
2:    $[\mathbf{G}^1, \mathbf{G}^2, \mathbf{G}^3] \leftarrow \text{GRADIENTVEC2D}(q, me, vol)$   $\triangleright$  see Algo. 12
3:    $[\mathbb{I}_g, \mathbb{J}_g] \leftarrow \text{BUILDIGJG2DVF}(me)$   $\triangleright$  see Algo. 5
4:    $\mathbb{K}_g \leftarrow \mathbb{O}_{36 \times n_{me}}$ 
5:    $\mathbb{K}_g(1, :) \leftarrow ((\lambda + 2\mu) * \mathbf{G}^1(1, :) * \mathbf{G}^1(1, :) + \mu * \mathbf{G}^1(2, :) * \mathbf{G}^1(2, :)) *$ 
       $vol$ 
6:    $\mathbb{K}_g(2, :) \leftarrow (\lambda + \mu) * \mathbf{G}^1(1, :) * \mathbf{G}^1(2, :) * vol$ 
7:    $\vdots$ 
8:    $\mathbb{K}_g(30, :) \leftarrow (\lambda + \mu) * \mathbf{G}^3(1, :) * \mathbf{G}^3(2, :) * vol$ 
9:    $\mathbb{K}_g(36, :) \leftarrow ((\lambda + 2\mu) * \mathbf{G}^3(2, :) * \mathbf{G}^3(2, :) + \mu * \mathbf{G}^3(1, :) * \mathbf{G}^3(1, :)) *$ 
       $vol$ 
10:   $\mathbb{K}_g([7, 13, 14, 19, 20, 21, 25, 26, 27, 28, 31, 32, 33, 34, 35], :)$ 
       $\leftarrow \mathbb{K}_g([2, 3, 9, 4, 10, 16, 5, 11, 17, 23, 6, 12, 18, 24, 30], :)$ 
11:   $\mathbb{S} \leftarrow \text{SPARSE}(\mathbb{I}_g(:), \mathbb{J}_g(:), \mathbb{K}_g(:), 2n_q, 2n_q)$ 
12: end function

```

---

We now present numerical results that illustrate the performances of the different finite element assembly methods presented in this article.

**5. Benchmarks results.** We consider the assembly of the stiffness and elastic stiffness matrices in 2d and 3d, in the following vector languages

- Matlab (R2012b),
- Octave (3.6.3),
- Python 3.3.2 with *Numpy*[1.7.1] and *scipy*[0.12.0].

For each language, we first compare the computation times of the different versions of the matrix assembly code (base, OptV1 and OptV2), for each language considered. Then, we compare the performances of the OptV2 code with those of an implementation of the matrix assembly in C language, using the *SuiteSparse* library [10] (“CXSparse”), and in FreeFEM++. A comparison of the performances of recent and efficient Matlab/Octave codes and the OptV2 code is also given. In all benchmarks, the computational domain  $\Omega$  is the unit disk in 2d and the unit sphere in 3d. For each result in any language used, we present the average computation time for ten finite element assembly calculations.

**5.1. Comparison of the base, OptV1 and OptV2 assembly codes.** We show in Figures 5.1 and 5.2, in logarithmic scales and for each vector language, the performances of the matrix assembly codes versus the matrix dimension  $n_{dof}$ , for the 2d stiffness and 3d elastic stiffness matrices respectively. For each language, we observe that the OptV2 version is the fastest one and its complexity is about  $\mathcal{O}(n_{dof} \log(n_{dof}))$ . For the 2d stiffness matrix, the OptV1 version is about 20, 40 and 100 times slower in Matlab, Python and Octave respectively. Its numerical complexity is about  $\mathcal{O}(n_{dof} \log(n_{dof}))$ . The less performing method, the base version, has a complexity of  $\mathcal{O}(n_{dof}^2)$  in Matlab and Octave. However, its complexity seems to be  $\mathcal{O}(n_{dof})$  in Python. This is partially due to the use of the LIL format in the

sparse matrix assembly in Python, the conversion in CSC format being taken into account in the computation time. We obtain similar results with the stiffness matrix in 3d and the elastic stiffness matrix in 2d and 3d. Computation times and speedup are given in Tables 5.1 and 5.2 for the 2d stiffness and 3d elastic stiffness matrices respectively. In Appendix B, computation times and speedup are given in Tables B.1 and B.2 for the stiffness matrix in 3d and the elasticity stiffness matrix in 2d, respectively. We observe that the performance differences of the stiffness and elastic stiffness matrix assemblies in 2d and 3d are partially due to the increase of the data: on the unit disk (2d) and the unit sphere (3d), we have  $n_{me} \approx 2n_q$  and  $n_{me} \approx 6n_q$  respectively. With matrices of the same size (i.e. for an equal  $n_{dof}$ ), in comparison to the stiffness matrix in 2d, the number of local values to be computed are 2, 4 and 16 times larger for the elastic stiffness matrix in 2d, the stiffness matrix in 3d, and the elastic stiffness matrix in 3d respectively.

### 5.2. Comparison of the OptV2 version in vector languages with CXSparse and FreeFEM++.

We compare, in Tables 5.3 and 5.4, the OptV2 versions in Matlab/Octave/Python with a C implementation of the assembly (OptV1 version) using the *SuiteSparse* library [10] (“CXSparse”), and with a FreeFEM++ version for the stiffness matrix in 2d and the elasticity stiffness matrix in 3d. For the stiffness matrix in 3d and the elasticity stiffness matrix in 2d, one can refer to the Tables B.3 and B.4 in Appendix B. The matrix assembly codes with FreeFEM++, the complete listings and the values of the computation times are given in [7]. We observe that the OptV2 version is approximately 1.5 and 7.1 times in Matlab, 2.7 and 10.5 times in Octave and 2.4 and 7.8 times in Python faster than FreeFEM++ for the first two matrices. As might be expected, the vector languages are less efficient than the C language: in our benchmarks, the computation times are multiplied by a factor between 1.7 and 4.5 in Matlab, 1.25 and 3.1 in Octave, and 1.4 and 3.75 in Python. However, these performance differences are not as large as generally expected.

**5.3. Comparison with other matrix assemblies in Matlab and Octave.** In Matlab/Octave other efficient algorithms have been proposed recently in [1, 2, 15, 26]. More precisely, in [15], a vectorization is proposed, based on the permutation of two local loops with the one through the elements. This technique allows to easily assemble different matrices, from a reference element by affine transformation and by using a numerical integration. In [26], the implementation is based on extending element operations on arrays into operations on arrays of matrices, calling it a matrix-array operation, where the array elements are matrices rather than scalars, and the operations are defined by the rules of linear algebra. Thanks to these new tools and a quadrature formula, different matrices are computed without any loop. In [2], for the assembly of the stiffness matrix in 2d associated to  $\mathcal{P}_1$  finite elements, L. Chen builds vectorially the nine sparse matrices corresponding to the nine elements of the local stiffness matrix in 2d and adds them to obtain the global matrix. This method is close to the one proposed in Algorithm 10. For the 2d stiffness matrix, the performance of our algorithm is similar to the one of [12].

We compare these codes to the OptV2 version presented in this article, for the assembly of the 2d stiffness matrix. The computations have been done on our reference computer. On Tables 5.5 and 5.6, using Matlab and Octave respectively, we show the computation times versus the number of vertices of the mesh, for these different codes. For large sparse matrices, the OptV2 version allows gains in computational time of 5% to 20%, compared to the other vectorized codes (for sufficiently large meshes).

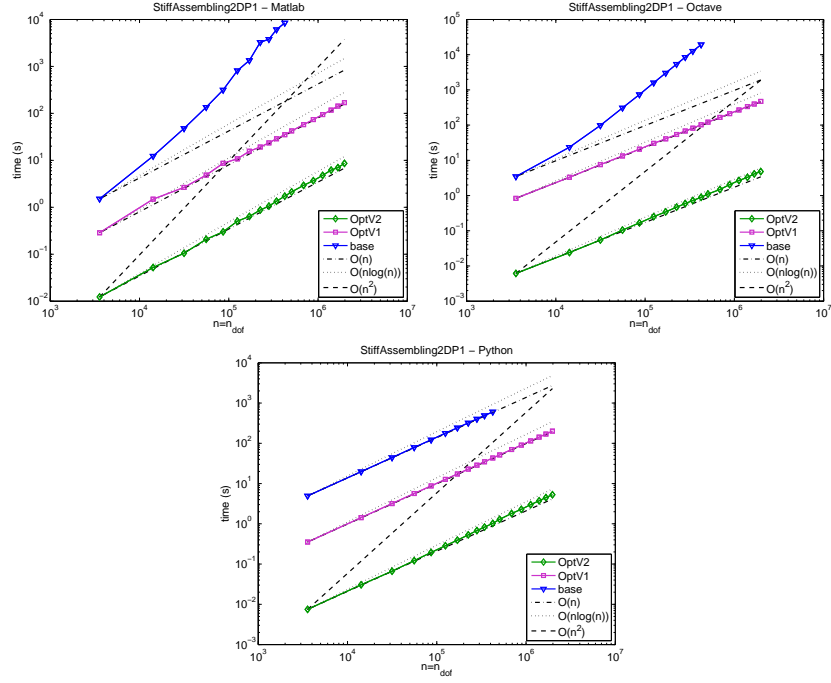


Fig. 5.1: 2d stiffness matrix : comparison of the classical, OptV1 and OptV2 matrix assembly codes in Matlab (top left), Octave (top right) and Python (bottom).

StiffAssembling2DP1 - Matlab				StiffAssembling2DP1 - Octave			
$n_{dof}$	OptV2	OptV1	base	$n_{dof}$	OptV2	OptV1	base
14222	0.052 (s) x 1.00	1.490 (s) x 28.53	12.294 (s) x 235.44	14222	0.024 (s) x 1.00	3.292 (s) x 136.71	23.518 (s) x 976.71
125010	0.506 (s) x 1.00	10.765 (s) x 21.27	812.572 (s) x 1605.24	125010	0.252 (s) x 1.00	30.296 (s) x 120.20	1594.866 (s) x 6327.48
343082	1.347 (s) x 1.00	28.760 (s) x 21.35	6085.713 (s) x 4518.24	343082	0.725 (s) x 1.00	81.988 (s) x 113.14	12558.432 (s) x 17330.32
885521	3.685 (s) x 1.00	73.819 (s) x 20.03	—	885521	2.040 (s) x 1.00	211.515 (s) x 103.69	—
1978602	8.615 (s) x 1.00	169.266 (s) x 19.65	—	1978602	4.795 (s) x 1.00	471.031 (s) x 98.24	—

StiffAssembling2DP1 - Python			
$n_{dof}$	OptV2	OptV1	base
14222	0.031 (s) x 1.00	1.423 (s) x 46.66	19.746 (s) x 647.27
125010	0.285 (s) x 1.00	12.678 (s) x 44.44	178.275 (s) x 624.89
343082	0.829 (s) x 1.00	34.801 (s) x 41.98	489.411 (s) x 590.36
885521	2.290 (s) x 1.00	90.190 (s) x 39.39	—
1978602	5.285 (s) x 1.00	201.295 (s) x 38.09	—

Table 5.1: 2d stiffness matrix : comparison of the OptV2, OptV1 and base matrix assembly codes in Matlab (top left), Octave (top right) and Python (bottom) giving time in seconds (top value) and speedup (bottom value). The speedup reference is OptV2 version.

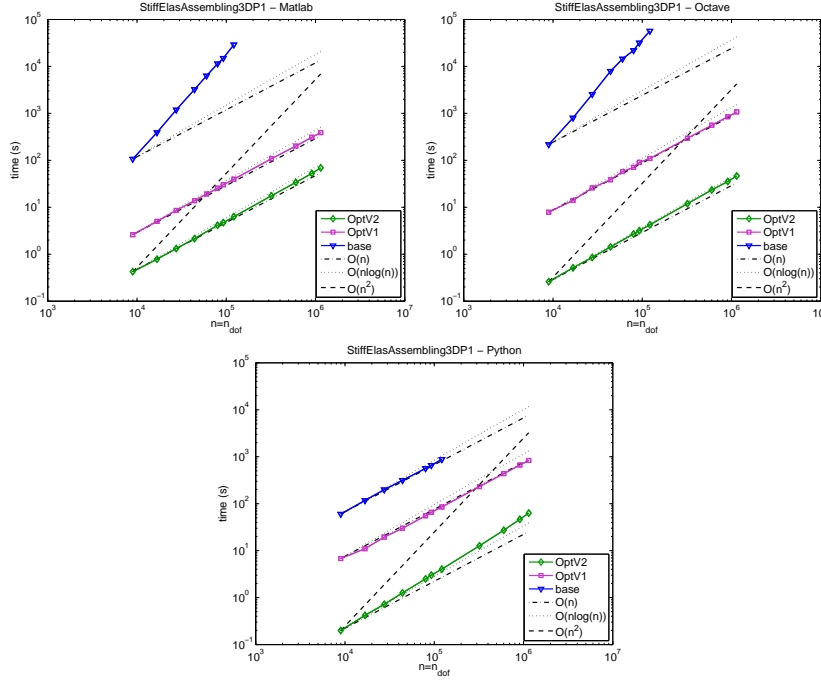


Fig. 5.2: 3d elastic stiffness matrix : comparison of the classical, OptV1 and OptV2 matrix assembly codes in Matlab (top left), Octave (top right) and Python (bottom).

StiffElasAssembling3DP1 - Matlab				StiffElasAssembling3DP1 - Octave			
$n_{dof}$	OptV2	OptV1	base	$n_{dof}$	OptV2	OptV1	base
16773	0.783 (s) x 1.00	5.016 (s) x 6.40	391.354 (s) x 499.65	16773	0.513 (s) x 1.00	13.998 (s) x 27.27	800.288 (s) x 1559.27
44124	2.147 (s) x 1.00	13.823 (s) x 6.44	3236.753 (s) x 1507.46	44124	1.435 (s) x 1.00	38.591 (s) x 26.94	7914.374 (s) x 5524.39
121710	6.309 (s) x 1.00	39.615 (s) x 6.28	29036.535 (s) x 4602.45	121710	4.232 (s) x 1.00	109.431 (s) x 25.86	56718.057 (s) x 13401.46
601272	33.798 (s) x 1.00	200.710 (s) x 5.94	—	601272	23.389 (s) x 1.00	558.996 (s) x 23.90	—
1144680	69.257 (s) x 1.00	389.074 (s) x 5.62	—	1144680	46.395 (s) x 1.00	1074.179 (s) x 23.15	—

StiffElasAssembling3DP1 - Python			
$n_{dof}$	OptV2	OptV1	base
16773	0.420 (s) x 1.00	10.982 (s) x 26.15	117.283 (s) x 279.27
44124	1.262 (s) x 1.00	29.991 (s) x 23.76	314.649 (s) x 249.27
121710	4.043 (s) x 1.00	85.132 (s) x 21.06	879.347 (s) x 217.52
601272	27.141 (s) x 1.00	436.719 (s) x 16.09	—
1144680	63.058 (s) x 1.00	834.281 (s) x 13.23	—

Table 5.2: 3d elastic stiffness matrix : comparison of the OptV2, OptV1 and base assembly codes in Matlab (top left), Octave (top right) and Python (bottom) giving time in seconds (top value) and speedup (bottom value). The speedup reference is OptV2 version.

$n_{dof}$	CXSparse (4.0.2)	Matlab (R2012b)	Octave (3.6.3)	Python (3.3.2)	FreeFem (3.2)
14222	0.010 (s) x 1.00	0.052 (s) x 5.22	0.024 (s) x 2.41	0.031 (s) x 3.05	0.081 (s) x 8.10
125010	0.192 (s) x 1.00	0.506 (s) x 2.64	0.252 (s) x 1.31	0.285 (s) x 1.49	0.735 (s) x 3.83
343082	0.579 (s) x 1.00	1.347 (s) x 2.33	0.725 (s) x 1.25	0.829 (s) x 1.43	2.042 (s) x 3.53
885521	1.552 (s) x 1.00	3.685 (s) x 2.37	2.040 (s) x 1.31	2.290 (s) x 1.48	5.333 (s) x 3.44
1978602	3.481 (s) x 1.00	8.615 (s) x 2.47	4.795 (s) x 1.38	5.285 (s) x 1.52	11.980 (s) x 3.44

Table 5.3: Computational cost of the 2d stiffness matrix assembly versus  $n_{dof}$ , with the OptV2 Matlab/Octave/Python version (2<sup>nd</sup>/3<sup>rd</sup>/4<sup>th</sup> columns), with CXSparse (1<sup>st</sup> column) and FreeFEM++ (5<sup>th</sup> column) : time in seconds (top value) and speedup (bottom value). The speedup reference is CXSparse code.

$n_{dof}$	CXSparse (4.0.2)	Matlab (R2012b)	Octave (3.6.3)	Python (3.3.2)	FreeFem (3.2)
16773	0.163 (s) x 1.00	0.783 (s) x 4.81	0.513 (s) x 3.15	0.420 (s) x 2.58	6.401 (s) x 39.27
44124	0.469 (s) x 1.00	2.147 (s) x 4.58	1.433 (s) x 3.05	1.262 (s) x 2.69	17.547 (s) x 37.41
121710	1.396 (s) x 1.00	6.309 (s) x 4.52	4.232 (s) x 3.03	4.043 (s) x 2.90	49.971 (s) x 35.80
601272	7.573 (s) x 1.00	33.798 (s) x 4.46	23.389 (s) x 3.09	27.141 (s) x 3.58	255.006 (s) x 33.67
1144680	16.832 (s) x 1.00	69.257 (s) x 4.11	46.395 (s) x 2.76	63.058 (s) x 3.75	489.377 (s) x 29.07

Table 5.4: Computational cost of the 3d elastic stiffness matrix assembly versus  $n_{dof}$ , with the OptV2 Matlab/Octave/Python version (2<sup>nd</sup>/3<sup>rd</sup>/4<sup>th</sup> columns), with CXSparse (1<sup>st</sup> column) and FreeFEM++ (5<sup>th</sup> column) : time in seconds (top value) and speedup (bottom value). The speedup reference is CXSparse code.

$n_q$	OptV2	Chen	iFEM	HanJun	RahVal
170355	0.638 (s) x 1.00	0.774 (s) x 0.82	0.663 (s) x 0.96	0.944 (s) x 0.68	0.995 (s) x 0.64
424178	1.733 (s) x 1.00	2.092 (s) x 0.83	1.771 (s) x 0.98	2.452 (s) x 0.71	2.634 (s) x 0.66
778415	3.113 (s) x 1.00	3.943 (s) x 0.79	3.694 (s) x 0.84	4.446 (s) x 0.70	4.984 (s) x 0.62
1251480	5.142 (s) x 1.00	6.595 (s) x 0.78	6.056 (s) x 0.85	7.320 (s) x 0.70	8.117 (s) x 0.63
1671052	6.937 (s) x 1.00	9.233 (s) x 0.75	8.557 (s) x 0.81	10.174 (s) x 0.68	10.886 (s) x 0.64
2349573	9.892 (s) x 1.00	12.778 (s) x 0.77	12.308 (s) x 0.80	14.384 (s) x 0.69	15.585 (s) x 0.63
3085628	13.157 (s) x 1.00	17.419 (s) x 0.76	16.575 (s) x 0.79	18.938 (s) x 0.69	20.767 (s) x 0.63

Table 5.5: Computational cost, in Matlab (R2012b), of the 2d stiffness matrix assembly versus  $n_q$ , with the OptV2 version (column 2) and with the codes in [1, 2, 15, 26] (columns 3-6) : time in seconds (top value) and speedup (bottom value). The speedup reference is OptV2 version.



$n_q$	OptV2	Chen	iFEM	HanJun	RahVal
170355	0.315 (s) x 1.00	0.353 (s) x 0.89	0.355 (s) x 0.89	0.740 (s) x 0.43	0.747 (s) x 0.42
424178	0.815 (s) x 1.00	0.970 (s) x 0.84	0.942 (s) x 0.86	1.917 (s) x 0.42	1.890 (s) x 0.43
778415	1.604 (s) x 1.00	1.945 (s) x 0.82	1.883 (s) x 0.85	3.985 (s) x 0.40	3.982 (s) x 0.40
1251480	2.662 (s) x 1.00	3.349 (s) x 0.79	3.307 (s) x 0.81	6.423 (s) x 0.41	6.752 (s) x 0.39
1671052	3.744 (s) x 1.00	4.533 (s) x 0.83	4.750 (s) x 0.79	9.310 (s) x 0.40	9.183 (s) x 0.41
2349573	5.253 (s) x 1.00	6.687 (s) x 0.79	7.227 (s) x 0.73	12.973 (s) x 0.40	13.195 (s) x 0.40
3085628	7.363 (s) x 1.00	8.833 (s) x 0.83	9.526 (s) x 0.77	18.001 (s) x 0.41	17.375 (s) x 0.42

Table 5.6: Computational cost, in Octave (3.6.3), of the 2d stiffness matrix assembly versus  $n_q$ , with the OptV2 version (column 2) and with the codes in [1, 2, 15, 26] (columns 3-6) : time in seconds (top value) and speedup (bottom value). The speedup reference is OptV2 version.

**6. Conclusion and work in progress.** We presented vectorized algorithms for the assembly of  $\mathcal{P}_1$  finite element matrices. The implementation of these algorithms (with very few difference from the algorithmic language) has been done in different vector languages such as Matlab, Octave and Python to calculate the stiffness and elastic stiffness matrices in 2d and 3d. We compared computation times between different versions (vectorized or not), different interpreted languages and C language. Numerical examples show the efficiency of the OptV2 algorithm: for the assembly of the stiffness matrix in 2d of size  $10^6$  the computation is performed in 3.9, 2.2 and 2.5 seconds with Matlab, Octave and Python respectively, and in 1.8 seconds with C. Less performance is obtained for the assembly of the elastic stiffness matrix in 3d: a matrix of size  $10^6$  is computed in 60.5, 40.5 and 55 seconds, with Matlab, Octave and Python respectively, and in 14.7 seconds with C.

We are now working on many variations for the OptV2 algorithm. Some of them may improve the performances and/or the memory usage. For example, to overcome the fact that the OptV2 algorithm is very memory consuming, one can split global arrays, generate incomplete sparse matrices and sum them up. This method is presented for the stiffness matrix in the vectorized Algorithm 10. This 2d version is about 15% faster with Matlab. However, it is about 10% and 30% slower under Octave and Python respectively. For the elastic stiffness matrix in 3d, a different implementation of the symmetry allows to obtain a similar algorithm with better performances: speedup by about a factor 2 for Matlab/Octave/Python and memory usage divided by a factor 6. Others improvements of the optimized assembly algorithm may be done using the specificities of each language.

Algorithm 10 Split vectorized assembly	Algorithm 11 Row assembly
<pre> <b>function</b> S ← STIFF( n<sub>q</sub>, n<sub>me</sub>, q, me, vol)   [G<sup>1</sup>, ..., G<sup>n<sub>dfe</sub></sup>] ← GRADIENTVEC(me, vol)   S ← O<sub>n<sub>q</sub> × n<sub>q</sub></sub>   <b>for</b> α ← 1 <b>to</b> n<sub>dfe</sub> <b>do</b>     S ← S + ELEM(..., G<sup>α</sup>, G<sup>α</sup>, α, α)     <b>for</b> β ← α + 1 <b>to</b> n<sub>dfe</sub> <b>do</b>       S<sub>imp</sub> ← ELEM(..., G<sup>α</sup>, G<sup>β</sup>, α, β)       S ← S + S<sub>imp</sub> + S<sub>imp</sub><sup>t</sup>     <b>end for</b>   <b>end for</b> <b>end function</b> </pre>	<pre> <b>function</b> S ← ELEM(..., G<sup>α</sup>, G<sup>β</sup>, α, β)   K<sub>gr</sub> ← sum(G<sup>α</sup>. * G<sup>β</sup>, 1). * vol   I<sub>gr</sub> ← me(α, :)   J<sub>gr</sub> ← me(β, :)   S ← sparse(I<sub>gr</sub>, J<sub>gr</sub>, K<sub>gr</sub>, n<sub>q</sub>, n<sub>q</sub>) <b>end function</b> </pre>

## Appendix A. Remaining routines.

**A.1. Vectorized computation of the gradients of the basis functions.** By construction, the gradients of the basis functions are constant on each element  $T_k$ . In what follows, we use the definition (4.2).

**A.1.1. The 2d case.** On a triangle  $T_k$ , we define  $\mathbf{D}^{12} = \mathbf{q}^{\text{me}(1,k)} - \mathbf{q}^{\text{me}(2,k)}$ ,  $\mathbf{D}^{13} = \mathbf{q}^{\text{me}(1,k)} - \mathbf{q}^{\text{me}(3,k)}$  and  $\mathbf{D}^{23} = \mathbf{q}^{\text{me}(2,k)} - \mathbf{q}^{\text{me}(3,k)}$ . Then we have

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

These formulas lead to a simple vectorization, given in Algorithm 12

---

**Algorithm 12** Vectorized computation of the gradients of the basis functions in 2d

---

**Input :**

- q : vertex coordinates array ( $2 \times n_q$ )
- me : connectivity array ( $3 \times n_{\text{me}}$ )
- vol : array of mesh elements areas ( $1 \times n_{\text{me}}$ )

**Output :**

- $\mathbf{G}^1, \mathbf{G}^2, \mathbf{G}^3$  : gradients arrays ( $2 \times n_{\text{me}}$ )
- $\mathbf{G}^\alpha(:, k) = \nabla \varphi_\alpha^k(\mathbf{q}), \forall \alpha \in \{1, \dots, 3\}$

**function** [ $\mathbf{G}^1, \mathbf{G}^2, \mathbf{G}^3$ ]  $\leftarrow$  GRADIENTVEC2D(q, me, vol)

$$\mathbf{D}^{12} \leftarrow \mathbf{q}(:, \text{me}(1,:)) - \mathbf{q}(:, \text{me}(2,:)) \quad \triangleright 2 \times n_{\text{me}} \text{ array}$$

$$\mathbf{D}^{13} \leftarrow \mathbf{q}(:, \text{me}(1,:)) - \mathbf{q}(:, \text{me}(3,:)) \quad \triangleright 2 \times n_{\text{me}} \text{ array}$$

$$\mathbf{D}^{23} \leftarrow \mathbf{q}(:, \text{me}(2,:)) - \mathbf{q}(:, \text{me}(3,:)) \quad \triangleright 2 \times n_{\text{me}} \text{ array}$$

$$\mathbf{G}^1 \leftarrow \begin{pmatrix} \mathbf{D}^{23}(2,:) ./ (2 * \text{vol}) \\ -\mathbf{D}^{23}(1,:) ./ (2 * \text{vol}) \end{pmatrix}$$

$$\mathbf{G}^2 \leftarrow \begin{pmatrix} -\mathbf{D}^{13}(2,:) ./ (2 * \text{vol}) \\ \mathbf{D}^{13}(1,:) ./ (2 * \text{vol}) \end{pmatrix}$$

$$\mathbf{G}^3 \leftarrow \begin{pmatrix} \mathbf{D}^{12}(2,:) ./ (2 * \text{vol}) \\ -\mathbf{D}^{12}(1,:) ./ (2 * \text{vol}) \end{pmatrix}$$

**end function**

---

**A.1.2. The 3d case.** On a tetrahedron  $T_k$ , we define

$$\mathbf{D}^{12} = \mathbf{q}^{\text{me}(1,k)} - \mathbf{q}^{\text{me}(2,k)}, \quad \mathbf{D}^{23} = \mathbf{q}^{\text{me}(2,k)} - \mathbf{q}^{\text{me}(3,k)},$$

$$\mathbf{D}^{13} = \mathbf{q}^{\text{me}(1,k)} - \mathbf{q}^{\text{me}(3,k)}, \quad \mathbf{D}^{24} = \mathbf{q}^{\text{me}(2,k)} - \mathbf{q}^{\text{me}(4,k)},$$

$$\mathbf{D}^{14} = \mathbf{q}^{\text{me}(1,k)} - \mathbf{q}^{\text{me}(4,k)}, \quad \mathbf{D}^{34} = \mathbf{q}^{\text{me}(3,k)} - \mathbf{q}^{\text{me}(4,k)}.$$

Then, we have

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

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

With these formulas, the vectorization is given in Algorithm 13

---

**Algorithm 13** Vectorized computation of the gradients of the basis functions in 3d

---

**Input :**

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

**Output :**

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

```

1: function [ $\mathbf{G}^1, \mathbf{G}^2, \mathbf{G}^3, \mathbf{G}^4$ ] ← GRADIENTVEC3D(q, me, vol)
2:    $\mathbf{D}^{12} \leftarrow \mathbf{q}(:, \text{me}(1, :)) - \mathbf{q}(:, \text{me}(2, :))$ 
3:    $\mathbf{D}^{13} \leftarrow \mathbf{q}(:, \text{me}(1, :)) - \mathbf{q}(:, \text{me}(3, :))$ 
4:    $\mathbf{D}^{14} \leftarrow \mathbf{q}(:, \text{me}(1, :)) - \mathbf{q}(:, \text{me}(4, :))$ 
5:    $\mathbf{D}^{23} \leftarrow \mathbf{q}(:, \text{me}(2, :)) - \mathbf{q}(:, \text{me}(3, :))$ 
6:    $\mathbf{D}^{24} \leftarrow \mathbf{q}(:, \text{me}(2, :)) - \mathbf{q}(:, \text{me}(4, :))$ 
7:    $C \leftarrow 1/(6 * \text{vol})$ 
8:    $\mathbf{G}^1 \leftarrow \begin{pmatrix} (-\mathbf{D}^{23}(2, :) * \mathbf{D}^{24}(3, :) + \mathbf{D}^{23}(3, :) * \mathbf{D}^{24}(2, :)) * C \\ (\mathbf{D}^{23}(1, :) * \mathbf{D}^{24}(3, :) - \mathbf{D}^{23}(3, :) * \mathbf{D}^{24}(1, :)) * C \\ (-\mathbf{D}^{23}(1, :) * \mathbf{D}^{24}(2, :) + \mathbf{D}^{23}(2, :) * \mathbf{D}^{24}(1, :)) * C \end{pmatrix}$ 
9:    $\mathbf{G}^2 \leftarrow \begin{pmatrix} (\mathbf{D}^{13}(2, :) * \mathbf{D}^{14}(3, :) - \mathbf{D}^{13}(3, :) * \mathbf{D}^{14}(2, :)) * C \\ (-\mathbf{D}^{13}(1, :) * \mathbf{D}^{14}(3, :) + \mathbf{D}^{13}(3, :) * \mathbf{D}^{14}(1, :)) * C \\ (\mathbf{D}^{13}(1, :) * \mathbf{D}^{14}(2, :) - \mathbf{D}^{13}(2, :) * \mathbf{D}^{14}(1, :)) * C \end{pmatrix}$ 
10:   $\mathbf{G}^3 \leftarrow \begin{pmatrix} (-\mathbf{D}^{12}(2, :) * \mathbf{D}^{14}(3, :) + \mathbf{D}^{12}(3, :) * \mathbf{D}^{14}(2, :)) * C \\ (\mathbf{D}^{12}(1, :) * \mathbf{D}^{14}(3, :) - \mathbf{D}^{12}(3, :) * \mathbf{D}^{14}(1, :)) * C \\ (-\mathbf{D}^{12}(1, :) * \mathbf{D}^{14}(2, :) + \mathbf{D}^{12}(2, :) * \mathbf{D}^{14}(1, :)) * C \end{pmatrix}$ 
11:   $\mathbf{G}^4 \leftarrow \begin{pmatrix} (\mathbf{D}^{12}(2, :) * \mathbf{D}^{13}(3, :) - \mathbf{D}^{12}(3, :) * \mathbf{D}^{13}(2, :)) * C \\ (-\mathbf{D}^{12}(1, :) * \mathbf{D}^{13}(3, :) + \mathbf{D}^{12}(3, :) * \mathbf{D}^{13}(1, :)) * C \\ (\mathbf{D}^{12}(1, :) * \mathbf{D}^{13}(2, :) - \mathbf{D}^{12}(2, :) * \mathbf{D}^{13}(1, :)) * C \end{pmatrix}$ 
12: end function

```

▷  $3 \times n_{me}$  array

---

**A.2. Elastic stiffness matrix.** The elastic stiffness matrix  $\mathbb{K}$  defined in Section 2.2 can be expressed with the operator  $\mathcal{H}$  (see (2.2)). This operator is defined in 2d, for all  $\mathbf{u} = (u_1, u_2) \in H^1(\Omega)^2$ , and for all  $\mathbf{v} = (v_1, v_2) \in H^1(\Omega)^2$ , by

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

and in 3d, for all  $\mathbf{u} = (u_1, u_2, u_3) \in (H^1(\Omega))^3$ , for all  $\mathbf{v} = (v_1, v_2, v_3) \in (H^1(\Omega))^3$ , by

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

where  $\lambda$  and  $\mu$  are the Lamé coefficients, and  $\gamma = \lambda + 2\mu$ . With this operator, we have

$$\mathbb{K}_{l,m} = \int_{\Omega_h} \underline{\boldsymbol{\varepsilon}}^t(\boldsymbol{\Psi}_m) \mathbb{C} \underline{\boldsymbol{\varepsilon}}(\boldsymbol{\Psi}_l) d\mathbf{q} = \int_{\Omega_h} \mathcal{H}(\boldsymbol{\Psi}_m, \boldsymbol{\Psi}_l) d\mathbf{q}, \quad \forall (l, m) \in \{1, \dots, n_{\text{dof}}\}^2.$$

**Appendix B. Benchmarks additional results.** In this section, we consider the assembly of the 3d stiffness and 2d elastic stiffness matrices. In Tables B.1 and B.2 we compare the computation times of the OptV2, OptV1 and base versions, in Matlab, Octave and Python. In Tables B.3 and B.4, we compare the OptV2 versions in Matlab/Octave/Python with a C implementation of the assembly (OptV1 version) using the *SuiteSparse* library [10] (“CXSparse”), and with a FreeFEM++ version.

StiffAssembling3DP1 - Matlab				StiffAssembling3DP1 - Octave			
$n_{dof}$	OptV2	OptV1	base	$n_{dof}$	OptV2	OptV1	base
14708	0.289 (s) x 1.00	4.570 (s) x 15.80	52.189 (s) x 180.42	14708	0.200 (s) x 1.00	26.175 (s) x 130.96	92.095 (s) x 460.76
40570	0.889 (s) x 1.00	13.052 (s) x 14.69	352.710 (s) x 396.91	40570	0.637 (s) x 1.00	74.022 (s) x 116.17	602.971 (s) x 946.33
200424	4.775 (s) x 1.00	67.004 (s) x 14.03	9591.131 (s) x 2008.56	200424	3.587 (s) x 1.00	377.162 (s) x 105.13	15970.084 (s) x 4451.66
580975	15.117 (s) x 1.00	196.405 (s) x 12.99	—	580975	11.457 (s) x 1.00	1104.374 (s) x 96.39	—
1747861	48.582 (s) x 1.00	599.601 (s) x 12.34	—	1747861	44.064 (s) x 1.00	3359.580 (s) x 76.24	—

StiffAssembling3DP1 - Python			
$n_{dof}$	OptV2	OptV1	base
14708	0.178 (s) x 1.00	5.423 (s) x 30.44	106.954 (s) x 600.31
40570	0.524 (s) x 1.00	15.418 (s) x 29.42	303.555 (s) x 579.26
200424	3.531 (s) x 1.00	78.638 (s) x 22.27	1451.837 (s) x 411.18
580975	12.090 (s) x 1.00	231.443 (s) x 19.14	—
1747861	49.006 (s) x 1.00	707.233 (s) x 14.43	—

Table B.1: 3d stiffness matrix : comparison of the OptV2, OptV1 and base assembly codes in Matlab (top left), Octave (top right) and Python (bottom) giving time in seconds (top value) and speedup (bottom value). The speedup reference is OptV2 version.

StiffElasAssembling2DP1 - Matlab				StiffElasAssembling2DP1 - Octave			
$n_{dof}$	OptV2	OptV1	base	$n_{dof}$	OptV2	OptV1	base
28444	0.197 (s) x 1.00	1.509 (s) x 7.67	108.713 (s) x 552.45	28444	0.088 (s) x 1.00	7.873 (s) x 89.28	225.183 (s) x 2553.53
111838	0.769 (s) x 1.00	6.084 (s) x 7.91	1885.628 (s) x 2452.81	111838	0.428 (s) x 1.00	31.230 (s) x 73.01	4086.153 (s) x 9563.00
250020	1.757 (s) x 1.00	13.490 (s) x 7.68	—	250020	0.997 (s) x 1.00	70.025 (s) x 70.21	—
1013412	7.728 (s) x 1.00	55.489 (s) x 7.18	—	1013412	4.304 (s) x 1.00	284.692 (s) x 66.15	—
2802258	22.862 (s) x 1.00	154.827 (s) x 6.77	—	2802258	12.893 (s) x 1.00	786.855 (s) x 61.03	—

StiffElasAssembling2DP1 - Python			
$n_{dof}$	OptV2	OptV1	base
28444	0.093 (s) x 1.00	4.283 (s) x 46.08	27.884 (s) x 300.00
111838	0.451 (s) x 1.00	16.864 (s) x 37.36	110.468 (s) x 244.72
250020	1.041 (s) x 1.00	37.878 (s) x 36.39	244.301 (s) x 234.74
1013412	4.455 (s) x 1.00	154.028 (s) x 34.57	—
2802258	13.301 (s) x 1.00	427.890 (s) x 32.17	—

Table B.2: 2d elastic stiffness matrix : comparison of the OptV2, OptV1 and base assembly codes in Matlab (top left), Octave (top right) and Python (bottom) giving time in seconds (top value) and speedup (bottom value). The speedup reference is OptV2 version.

**Appendix C. Vectorized algorithmic language.** In computer science, vector languages (also known as array programming or multidimensional languages) are often used in scientific and engineering computations. They generalize the operations on scalars to higher dimensional arrays, matrices and vectors : such operations are named vectorized operations.

$n_{dof}$	CXSparse (4.0.2)	Matlab (R2012b)	Octave (3.6.3)	Python (3.3.2)	FreeFem (3.2)
14708	0.078 (s) x 1.00	0.289 (s) x 3.71	0.200 (s) x 2.56	0.178 (s) x 2.28	0.824 (s) x 10.56
40570	0.247 (s) x 1.00	0.889 (s) x 3.60	0.637 (s) x 2.58	0.524 (s) x 2.12	2.386 (s) x 9.66
200424	2.081 (s) x 1.00	4.775 (s) x 2.29	3.587 (s) x 1.72	3.531 (s) x 1.70	12.914 (s) x 6.21
580975	7.648 (s) x 1.00	15.117 (s) x 1.98	11.457 (s) x 1.50	12.090 (s) x 1.58	39.185 (s) x 5.12
1747861	29.595 (s) x 1.00	48.582 (s) x 1.64	44.064 (s) x 1.49	49.006 (s) x 1.66	122.126 (s) x 4.13

Table B.3: Computational cost of the 3d stiffness matrix assembly versus  $n_{dof}$ , with the OptV2 Matlab/Octave/Python version (2<sup>nd</sup>/3<sup>rd</sup>/4<sup>th</sup> columns), with CXSparse (1<sup>st</sup> column) and FreeFEM++ (5<sup>th</sup> column) : time in seconds (top value) and speedup (bottom value). The speedup reference is CXSparse code.

$n_{dof}$	CXSparse (4.0.2)	Matlab (R2012b)	Octave (3.6.3)	Python (3.3.2)	FreeFem (3.2)
14222	0.039 (s) x 1.00	0.197 (s) x 5.05	0.088 (s) x 2.26	0.154 (s) x 3.94	1.137 (s) x 29.15
55919	0.156 (s) x 1.00	0.769 (s) x 4.93	0.428 (s) x 2.74	0.677 (s) x 4.34	4.530 (s) x 29.04
125010	0.377 (s) x 1.00	1.757 (s) x 4.66	0.997 (s) x 2.65	1.598 (s) x 4.24	10.165 (s) x 26.96
506706	1.574 (s) x 1.00	7.728 (s) x 4.91	4.304 (s) x 2.73	6.889 (s) x 4.38	41.459 (s) x 26.34
1401129	4.327 (s) x 1.00	22.862 (s) x 5.28	12.893 (s) x 2.98	20.048 (s) x 4.63	114.049 (s) x 26.36

Table B.4: Computational cost of the 2d elastic stiffness matrix assembly versus  $n_{dof}$ , with the OptV2 Matlab/Octave/Python version (2<sup>nd</sup>/3<sup>rd</sup>/4<sup>th</sup> columns), with CXSparse (1<sup>st</sup> column) and FreeFEM++ (5<sup>th</sup> column) : time in seconds (top value) and speedup (bottom value). The speedup reference is CXSparse code.

We provide below some common functions and operators of the vectorized algorithmic language used in this article, which is close to Matlab/Octave.

$\mathbb{A} \leftarrow \mathbb{B}$	Assignment
$\mathbb{A} * \mathbb{B}$	matrix multiplication,
$\mathbb{A} . * \mathbb{B}$	element-wise multiplication,
$\mathbb{A} ./ \mathbb{B}$	element-wise division,
$\mathbb{A}(:)$	all the elements of $\mathbb{A}$ , regarded as a single column.
$[ \ ]$	Horizontal concatenation,
$[ ; ]$	Vertical concatenation,
$\mathbb{A}(:, J)$	$J$ -th column of $\mathbb{A}$
$\mathbb{A}(I, :)$	$I$ -th row of $\mathbb{A}$
$\text{sum}(\mathbb{A}, dim)$	sums along the dimension $dim$ .
$\mathbb{1}_{m \times n}$	$m$ -by- $n$ array or sparse matrix of ones.
$\mathbb{O}_{m \times n}$	$m$ -by- $n$ array or sparse matrix of zeros.

**Acknowledgements.** The authors would like to thank Prof. H-P. Langtangen for his many constructive comments that led to a better presentation of the paper.

#### REFERENCES

- [1] L. CHEN, *Programming of Finite Element Methods in Matlab*, Preprint, University of California Irvine, <http://math.uci.edu/~chenlong/226/Ch3FEMCode.pdf>, 2011.
- [2] L. CHEN, *iFEM, a Matlab software package*, University of California Irvine, <http://math.uci.edu/~chenlong/programming.html>, 2013.
- [3] P. G. CIARLET, *The finite element method for elliptic problems*, SIAM, Philadelphia, 2002.
- [4] Z. CHEN, *Finite Element Methods and their Applications*, Scientific Computation, Springer, 2005.
- [5] *Cusp*, A C++ Templated Library for sparse linear algebra on CUDA, <https://developer.nvidia.com/cusp>.
- [6] F. CUVELIER, C. JAPHET, AND G. SCARELLA, *OptFEMPI, MATLAB, Octave, Python packages*, Université Paris 13, <https://github.com/gscarella/>, 2013.
- [7] F. CUVELIER, C. JAPHET, AND G. SCARELLA, *An efficient way to perform the assembly of finite element matrices in Matlab and Octave*, Preprint, Université Paris 13 and INRIA, <http://hal.inria.fr/hal-00821942>, 2013.
- [8] M. DABROWSKI, M. KROTKIEWSKI, AND D. W. SCHMID, *Milamin: Matlab-based finite element method solver for large problems*, Geophys. Geosyst., 9, 2008.
- [9] T. A. DAVIS, *Direct Methods for Sparse Linear Systems*, SIAM, 2006.
- [10] T. A. DAVIS, *SuiteSparse packages, release 4.0.2*, University of Florida, <http://www.cise.ufl.edu/research/sparse/SuiteSparse>, 2012.
- [11] G. DHATT, E. LEFRANÇOIS, AND G. TOUZOT, *Finite Element Method*, Wiley, 2012.
- [12] S. FUNKEN, D. PRAETORIUS, AND P. WISSGOTT, *Efficient implementation of adaptive P1-FEM in MATLAB*, Computational Methods in Applied Mathematics, 11(4), 460–490, 2011.
- [13] *GNU Octave*, <http://www.gnu.org/software/octave>.
- [14] M.J. GANDER AND C. JAPHET, *Algorithm 932: PANG: Software for Non-Matching Grid Projections in 2d and 3d with Linear Complexity*, ACM Transactions on Mathematical Software (TOMS), 40(1), pp. 6:1–6:25, 2013.
- [15] A. HANNUKAINEN, AND M. JUNTUNEN, *Implementing the Finite Element Assembly in Interpreted Languages*, Preprint, Aalto University, <http://users.tkk.fi/~mojuntun/preprints/matvecSISC.pdf>, 2012.
- [16] F. HECHT, *New development in freefem++*, J. Numer. Math. 20, no. 3-4, 251—265, 2012.
- [17] J.S. HESTHAVEN, T. WARBURTON, *Nodal Discontinuous Galerkin Methods, Algorithms, Analysis, and Applications*, Texts in Applied Mathematics, Vol. 54, Springer, 2008.
- [18] C. JOHNSON, *Numerical Solution of Partial Differential Equations by the Finite Element Method*, Dover Publications, Inc, 2009.
- [19] M.CERMAK, T. KOZUBEK, S. SYSALA AND J. VALDMAN *A FETI Domain Decomposition Solver for Elastoplastic Problems*, arXiv:1205.1926, 2012.
- [20] H.P LANGTANGEN AND X. CAI, *On the efficiency of Python for high-performance computing: A case study involving stencil updates for partial differential equations*. In H. G. Bock, E. Kostina, H. X. Phu, and R. Rannacher, editors, Modeling, Simulation and Optimization of Complex Processes, pages 337–358. Springer, 2008.
- [21] B. LUCQUIN, AND O. PIRONNEAU, *Introduction to Scientific Computing*, John Wiley & Sons Ltd, 1998.
- [22] *Matlab*, <http://www.mathworks.com>.
- [23] *Python*, <http://www.python.org/>.
- [24] A. QUARTERONI AND A. VALLI, *Numerical Approximation of Partial Differential Equations*, Springer, 1994.
- [25] A. QUARTERONI, F. SALERI AND P. GERVASIO, *Scientific Computing with MATLAB and Octave*, Texts in Computational Science and Engineering, Springer, 2013.
- [26] T. RAHMAN, AND J. VALDMAN, *Fast MATLAB assembly of FEM matrices in 2D and 3D: Nodal elements*, Appl. Math. Comput., 219(13), pp. 7151–7158, 2013.
- [27] V. THOMÉE, *Galerkin Finite Element Method for Parabolic Problems*, Springer, 1994.
- [28] *Thrust*, a C++ template library for CUDA based on the Standard Template Library (STL), <https://developer.nvidia.com/thrust>.