An efficient way to assemble finite element matrices in vector languages

François Cuvelier, Caroline Japhet, Gilles Scarella

An efficient way to assemble finite element matrices in vector languages

François Cuvelier · Caroline Japhet · Gilles Scarella

Received: date / Accepted: date

Abstract Efficient Matlab codes in 2D and 3D have been proposed recently to assemble finite element matrices. In this paper we present simple, compact and efficient vectorized algorithms, which are variants of these codes, in arbitrary dimension, without the use of any lower level language. They can be easily implemented in many vector languages (e.g. Matlab, Octave, Python, Scilab, R, Julia, C++ with STL,...). The principle of these techniques is general, we present it for the assembly of several finite element matrices in arbitrary dimension, in the $\mathbb{P}_1$ finite element case. We also provide an extension of the algorithms to the case of a system of PDE's. Then we give an extension to piecewise polynomials of higher order. We compare numerically the performance of these algorithms in Matlab, Octave and Python, with that in FreeFEM++ and in a compiled language such as C. Examples show that, unlike what is commonly believed, the performance is not radically worse than that of C : in the best/worst cases, selected vector languages are respectively 2.3/3.5 and 2.9/4.1 times slower than C in the scalar and vector cases. We also present numerical results which illustrate the computational costs of these algorithms compared to standard algorithms and to other recent ones.

Keywords finite elements, matrix assembly, vectorization, vector languages, Matlab, Octave, Python

Mathematics Subject Classification (2000) 65N30, 65Y20, 74S05

This work was partially funded by GNR MoMaS, CoCOA LEFE project, ANR DEDALES and MathSTIC (University Paris 13)

F. Cuvelier · G. Scarella
Université Paris 13, Sorbonne Paris Cité, LAGA, CNRS UMR 7539, 99 Avenue J-B Clément, F-93430 Villetaneuse, France, E-mail: cuvelier@math.univ-paris13.fr, scarella@math.univ-paris13.fr

C. Japhet
Université Paris 13, Sorbonne Paris Cité, LAGA, CNRS UMR 7539, 99 Avenue J-B Clément, F-93430 Villetaneuse, France. INRIA Paris-Rocquencourt, BP 105, F-78153 Le Chesnay, France, E-mail: japhet@math.univ-paris13.fr

1 Introduction

Vector languages[1] such as Matlab [27], GNU Octave [30], Python [13], R [14], Scilab [12], Julia [22], C++ with STL,..., are very widely used for scientific computing (see for example [3,25,20,24,32]) 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 this compact code fast enough for realistic simulations.

On the other hand, in finite element simulations [4,5,21,33,35], the need for efficient algorithms for assembling the matrices may be crucial, especially when the matrices may need to be assembled several times. This is the case for example when simulating time-dependent problems with explicit or implicit schemes with time-dependent coefficients (e.g. in ocean-atmosphere coupling or porous medium applications). Other examples are computations with a posteriori estimates when one needs to reassemble the matrix equation on a finer mesh, or in the context of eigenvalue problems where assembling the matrix may be costly. In any event, assembly remains a critical part of code optimization since solution of linear systems, which asymptotically dominates in large-scale computing, could be done with the linear solvers of the different vector languages.

In a vector language, the inclusion of loops is a critical performance degrading aspect and removing them is known as a vectorization. In finite element programming, the classical finite element assembly is based on a loop over the elements (see for example [26]). 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 [1,2,3,8,15,17,23,34].

In this paper we describe vectorized algorithms, which are variants of the codes in [2,3,15,23], extended to arbitrary dimension $d \geq 1$, for assembling large sparse matrices in finite element computations. A particular strength of these algorithms is that they make using, reading and extending the codes easier while achieving performance close to that of C.

The aim of this article is the quantitative studies for illustrating the efficiency of the vector languages and the various speed-up of the algorithms, relatively to each other, to C and to FreeFem++ [19]. We also propose a vectorized algorithm in arbitrary dimension which is easily transposable to matrices arising from PDE's such as (see [33])

$$-\nabla \cdot (\mathbb{A}\nabla u) + \nabla \cdot (\boldsymbol{b}u) + \boldsymbol{c} \cdot \nabla u + a_0 u = f \quad \text{in } \Omega, \tag{1.1}$$

where Ω is a bounded domain of $\mathbb{R}^d$ ($d \geq 1$), $\mathbb{A} \in (L^\infty(\Omega))^{d\times d}$, $\boldsymbol{b} \in (L^\infty(\Omega))^d$, $\boldsymbol{c} \in (L^\infty(\Omega))^d$, $a_0 \in L^\infty(\Omega)$ and $f \in L^2(\Omega)$ are given functions. The description of the vectorized algorithm is done in three steps: we recall (non-

[1] which contain usual element-wise operators and functions on multidimensional arrays

vectorized) versions called `base` and `OptV1`. The latter requires sparse matrix tools found in most of the languages used for computational science and engineering. Then we give vectorized algorithms which are much faster: `OptV2` (memory consuming), `OptV` (less memory consuming) and `OptVS` (a symmetrized version of `OptV`). These algorithms have been tested for several matrices (e.g. weighted mass, stiffness and elastic stiffness matrices) and in different languages. We also provide an extension to the vector case in arbitrary dimension, where the algorithm is applied to the elastic stiffness matrix with variable coefficients, in 2D and 3D.

For space considerations, we restrict ourselves in this paper to $\mathbb{P}_1$ Lagrange finite elements. However, in the appendix we show that with slight modification, the algorithm is valid for piecewise polynomials of higher order.

These algorithms can be efficiently implemented in many languages if the language has a sparse matrix implementation. For the `OptV1`, `OptV2`, `OptV` and `OptVS` versions, a particular sparse matrix constructor is also needed (see Section 3) and these versions require that 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* and *Cusp*, C++ libraries for CUDA

This paper is organized as follows: in Section 2 we define two examples of finite element matrices. Then we introduce the notation associated to the mesh and to the algorithmic language used in this article. In Section 3 we give the classical and `OptV1` algorithms. In Section 4 we present the vectorized `OptV2` and `OptV` algorithms for a generic sparse matrix and $\mathbb{P}_1$ finite elements, with the application to the assemblies of the matrices of Section 2. A similar version called `OptVS` for symmetric matrices is also given. A first step towards finite elements of higher order is deferred to Appendix B. In Section 5 we consider the extension to the vector case with an application to linear elasticity. In Section 6, benchmark results illustrate the performance of the algorithms in the Matlab, Octave and Python languages. First, we show a comparison between the classical, `OptV1`, `OptV2`, `OptV` and `OptVS` versions. Then we compare the performances of the `OptVS` version to those obtained with a compiled language (using SuiteSparse [10] in C language), the latter being well-known to run at high speed and serving as a reference. A comparison is also given with FreeFEM++ [18] as a simple and reliable finite element software. We also show in Matlab and Octave a comparison of the `OptVS` algorithm and the codes given in [2,3,17,34].

All the computations are done on our reference computer[2] with the releases R2014b for Matlab, 3.8.1 for Octave, 3.4.0 for Python and 3.31 for FreeFEM++. The Matlab/Octave and Python codes may be found in [7].

[2] 2 x Intel Xeon E5-2630v2 (6 cores) at 2.60Ghz, 64Go RAM

2 Statement of the problem and notation

In this article we consider the assembly of the standard sparse matrices (e.g. weighted mass, stiffness and elastic stiffness matrices) arising from the $\mathbb{P}_1$ finite element discretization of partial differential equations (see e.g. [5,33]) in a bounded domain Ω of $\mathbb{R}^d$ ($d \geq 1$).

We suppose that Ω is equipped with a mesh $\mathcal{T}_h$ (locally conforming) as described in Table 2.1. We suppose that the elements belonging to the mesh are d-simplices . We introduce the finite dimensional space $X_h^1 = \{v \in \mathcal{C}^0(\overline{\Omega_h}),\ v_{|K} \in \mathbb{P}_1(K),\ \forall K \in \mathcal{T}_h\}$ where $\Omega_h = \bigcup_{K\in\mathcal{T}_h} K$ and $\mathbb{P}_1(K)$ denotes the space of all polynomials over K and of total degree less than or equal to 1. Let q^j, $j = 1, ..., \mathrm{n_q}$ be a vertex of Ω_h, with $\mathrm{n_q} = dim(X_h^1)$. The space X_h^1 is spanned by the $\mathbb{P}_1$ Lagrange basis functions $\{\varphi_i\}_{i\in\{1,...,\mathrm{n_q}\}}$ in $\mathbb{R}^d$, where $\varphi_i(\mathrm{q}^j) = \delta_{ij}$, with δ_{ij} the Kronecker delta.
We consider two examples of finite element matrices: the weighted mass matrix $\mathbb{M}^{[w]}$, with $w \in L^\infty(\Omega)$, defined by

$$\mathbb{M}^{[w]}_{i,j} = \int_{\Omega_h} w\varphi_j\varphi_i d\mathrm{q}, \quad \forall (i,j) \in \{1, ..., \mathrm{n_q}\}^2, \tag{2.1}$$

and the stiffness matrix $\mathbb{S}$ given by

$$\mathbb{S}_{i,j} = \int_{\Omega_h} \langle \nabla\,\varphi_j, \nabla\,\varphi_i\rangle\, d\mathrm{q}, \quad \forall (i,j) \in \{1, ..., \mathrm{n_q}\}^2. \tag{2.2}$$

Note that on the k-th element $K = T_k$ of $\mathcal{T}_h$ we have

$$\forall \alpha \in \{1, \ldots, d+1\}, \quad \varphi_{i|T_k} = \lambda_\alpha, \text{ with } i = \mathrm{me}(\alpha, k), \tag{2.3}$$

where $(\lambda_\alpha)_{\alpha\in\{1,...,d+1\}}$ are the barycentric coordinates (i.e the local $\mathbb{P}_1$ Lagrange basis functions) of K, and me is the connectivity array (see Table 2.1). The matrices $\mathbb{M}^{[w]}$ and $\mathbb{S}$ can be assembled efficiently with a vectorized algorithm proposed in Section 4, which uses the following formula (see e.g. [31])

$$\int_K \prod_{i=1}^{d+1} \lambda_i^{n_i} d\mathrm{q} = d!|K| \frac{\prod_{i=1}^{d+1} n_i!}{(d + \sum_{i=1}^{d+1} n_i)!} \tag{2.4}$$

where $|K|$ is the volume of K and $n_i \in \mathbb{N}$.

Remark 2.1 The (non-vectorized or vectorized) finite element assembly algorithms presented in this article may be adapted to compute matrices associated to the bilinear form (1.1).

Remark 2.2 These algorithms apply to finite element methods of higher order. Indeed, one can express the $\mathbb{P}_k$-Lagrange basis functions ($k \geq 2$) as polynomials in λ_i variable and then use formula (2.4). In Appendix D we give a first step to obtain a vectorized algorithm for $\mathbb{P}_k$ finite elements.

In the remainder of this article, we will use the following notations to describe the triangulation $\mathcal{T}_h$ of Ω:

name	type	dimension	description
d	integer	1	dimension of simplices of $\mathcal{T}_h$
$\mathrm{n_q}$	integer	1	number of vertices of $\mathcal{T}_h$
$\mathrm{n_{me}}$	integer	1	number of mesh elements in $\mathcal{T}_h$
q	double	$d \times \mathrm{n_q}$	array of vertex coordinates
me	integer	$(d+1) \times \mathrm{n_{me}}$	connectivity array
vols	double	$1 \times \mathrm{n_{me}}$	array of simplex volumes

Table 2.1: Data structure associated to the mesh $\mathcal{T}_h$

In Table 2.1, for $\nu \in \{1, \ldots, d\}$, $\mathrm{q}(\nu, j)$ represents the ν-th coordinate of the j-th vertex, $j \in \{1, \ldots, \mathrm{n_q}\}$. The j-th vertex will be also denoted by q^j. The term $\mathrm{me}(\beta, k)$ is the storage index of the β-th vertex of the k-th element, in the array q, for $\beta \in \{1, ..., d+1\}$ and $k \in \{1, \ldots, \mathrm{n_{me}}\}$.

We also provide below some common functions and operators of the vectorized algorithmic language used in this article which generalize the operations on scalars to higher dimensional arrays, matrices and vectors:

$\mathbb{A} \leftarrow \mathbb{B}$	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 $\mathbb{A}$,
SUM$(\mathbb{A}, dim)$	sums along the dimension dim,
$\mathbb{I}_n$	n-by-n identity matrix,
$\mathbb{1}_{m\times n}$ (or $\mathbb{1}_n$)	m-by-n (or n-by-n) matrix or sparse matrix of ones,
$\mathbb{O}_{m\times n}$ (or $\mathbb{O}_n$)	m-by-n (or n-by-n) matrix or sparse matrix of zeros,
ONES$(n_1, n_2, ..., n_\ell)$	ℓ dimensional array of ones,
ZEROS$(n_1, n_2, ..., n_\ell)$	ℓ dimensional array of zeros.

3 Standard finite element assemblies

In this section we consider the $\mathbb{P}_1$ finite element assembly of a generic $\mathrm{n_q}$-by-$\mathrm{n_q}$ sparse matrix $\mathbb{M}$ with its corresponding $(d+1)$-by-$(d+1)$ local matrix $\mathbb{E}$ (also denoted by $\mathbb{E}(K)$ when referring to an element $K \in \mathcal{T}_h$). For $K = T_k$, the (α, β)-th entry of $\mathbb{E}(T_k)$ is denoted by $e^k_{\alpha,\beta}$.

In Algorithm 3.1, we recall the classical finite element assembly method for calculating $\mathbb{M}$. In this algorithm, an $\mathrm{n_q}$-by-$\mathrm{n_q}$ sparse matrix $\mathbb{M}$ is first declared, then the contribution of each element $T_k \in \mathcal{T}_h$, given by a function `ElemMat`, is added to the matrix $\mathbb{M}$. These successive operations are very expensive due to a suboptimal use of the `sparse` function.

A first optimized, non-vectorized, version (called `OptV1`), suggested in [9], is based on the use of the `sparse` function:

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

This command returns an `m` $\times$`n` sparse matrix M such that

$$\texttt{M(Ig(k),Jg(k))} \leftarrow \texttt{M(Ig(k),Jg(k)) + Kg(k).}$$

The vectors `Ig`, `Jg` and `Kg` have the same length. The zero elements of `K` are not taken into account and the elements of `Kg` having the same indices in `Ig` and `Jg` are summed.
Examples of languages containing a sparse function are given below

- Python (*scipy.sparse* module) :
  `M=sparse.<format>_matrix((Kg,(Ig,Jg)),shape=(m,n))`
  where `<format>` is the sparse matrix format (e.g. `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.
- C with *SuiteSparse* [10]
- CUDA with *Thrust* [29] and *Cusp* [28] libraries

The `OptV1` version consists in computing and storing all elementary contributions first and then using them to generate the sparse matrix $\mathbb{M}$. The main idea is to create three global 1d-arrays $\boldsymbol{K}_g$, $\boldsymbol{I}_g$ and $\boldsymbol{J}_g$ of length $(d+1)^2 \mathrm{n_{me}}$, which store the local matrices as well as the position of their elements in the global matrix as shown on Figure 3.1. To create the arrays $\boldsymbol{K}_g$, $\boldsymbol{I}_g$ and $\boldsymbol{J}_g$, we define three local arrays $\boldsymbol{K}_k^e$, $\boldsymbol{I}_k^e$ and $\boldsymbol{J}_k^e$ of length $(d+1)^2$ obtained from the $(d+1)$-by-$(d+1)$ local matrix $\mathbb{E}(T_k)$ as follows:

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

Using $\boldsymbol{K}_k^e$, $\boldsymbol{I}_k^e$, $\boldsymbol{J}_k^e$ and a loop over the mesh elements T_k, one may calculate the global arrays $\boldsymbol{I}_g$, $\boldsymbol{J}_g$ and $\boldsymbol{K}_g$. The corresponding `OptV1` algorithm is given in Algorithm 3.2.

Numerical experiments in Section 6.1 and in Tables A.3 and A.4 show that the `OptV1` algorithm is more efficient than the classical one. The inefficiency of the classical (`base`) version compared to the `OptV1` version is mainly due to the repetition of element insertions into the sparse structure and to some dynamic reallocation troubles that may also occur.

However, the `OptV1` algorithm still uses a loop over the elements. To improve the efficiency of this algorithm, we propose in the next section other optimized versions, in a vectorized form: the main loop over the elements, which increases with the size of the mesh, is vectorized. The other loops (which are independent of the mesh size and with few iterations) will not necessarily be vectorized.

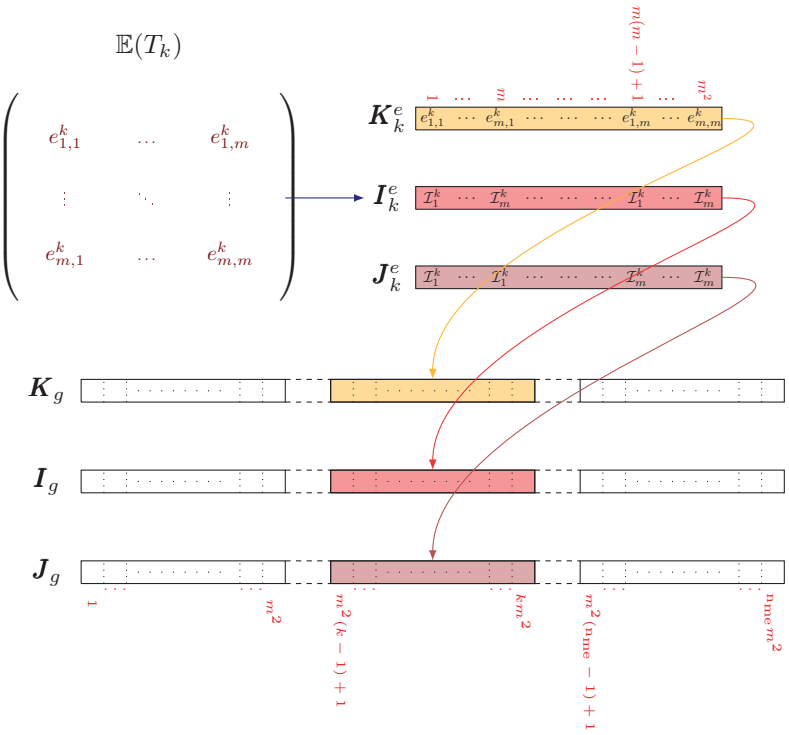

Fig. 3.1: Insertion of a local matrix into global 1d-arrays - `OptV1` version, where $m = d+1$, $\mathcal{I}^k_l = \mathrm{me}(l,k)$.

Algorithm 3.1 (`base`) - Classical assembly

$\mathbb{M} \leftarrow \mathbb{O}_{\mathrm{n_q}}$ ▷ Sparse matrix
for $k \leftarrow 1$ **to** $\mathrm{n_{me}}$ **do**
  $\mathbb{E} \leftarrow$ `ElemMat`$(\mathrm{vols}(k), \ldots)$
  for $\alpha \leftarrow 1$ **to** $d+1$ **do**
    $i \leftarrow \mathrm{me}(\alpha, k)$
    for $\beta \leftarrow 1$ **to** $d+1$ **do**
      $j \leftarrow \mathrm{me}(\beta, k)$
      $\mathbb{M}_{i,j} \leftarrow \mathbb{M}_{i,j} + \mathbb{E}_{\alpha,\beta}$
    end for
  end for
end for

Algorithm 3.2 (`OptV1`) - Optimized and non-vectorized assembly

$\boldsymbol{K}_g \leftarrow \boldsymbol{I}_g \leftarrow \boldsymbol{J}_g \leftarrow$ ZEROS$((d+1)^2 \mathrm{n_{me}}, 1)$
$l \leftarrow 1$
for $k \leftarrow 1$ **to** $\mathrm{n_{me}}$ **do**
  $\mathbb{E} \leftarrow$ `ElemMat`$(\mathrm{vols}(k), \ldots)$
  for $\beta \leftarrow 1$ **to** $d+1$ **do**
    for $\alpha \leftarrow 1$ **to** $d+1$ **do**
      $\boldsymbol{I}_g(l) \leftarrow \mathrm{me}(\alpha, k)$
      $\boldsymbol{J}_g(l) \leftarrow \mathrm{me}(\beta, k)$
      $\boldsymbol{K}_g(l) \leftarrow \mathbb{E}(\alpha, \beta)$
      $l \leftarrow l+1$
    end for
  end for
end for
$\mathbb{M} \leftarrow$`sparse`$(\boldsymbol{I}_g, \boldsymbol{J}_g, \boldsymbol{K}_g, \mathrm{n_q}, \mathrm{n_q})$

4 Optimized finite element assembly

In this section we present optimized algorithms, only available in vector languages. In the first algorithm, `OptV2`, the idea is to vectorize the main loop over the elements by defining the two-dimensional arrays $\mathbb{K}_g$, $\mathbb{I}_g$ and $\mathbb{J}_g$ of size $(d+1)^2$-by-$\mathrm{n_{me}}$ which 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:

$$\texttt{M} \leftarrow \texttt{sparse}(\mathbb{I}_g(:),\mathbb{J}_g(:),\mathbb{K}_g(:),\mathrm{n_q},\mathrm{n_q});$$

A non-vectorized approach inspired by `OptV1` is as follows: for each mesh element T_k, the k-th column of the global arrays $\mathbb{K}_g$, $\mathbb{I}_g$ and $\mathbb{J}_g$ is filled with the local arrays $\boldsymbol{K}_k^e$, $\boldsymbol{I}_k^e$, $\boldsymbol{J}_k^e$ respectively, as shown in Figure 4.1.

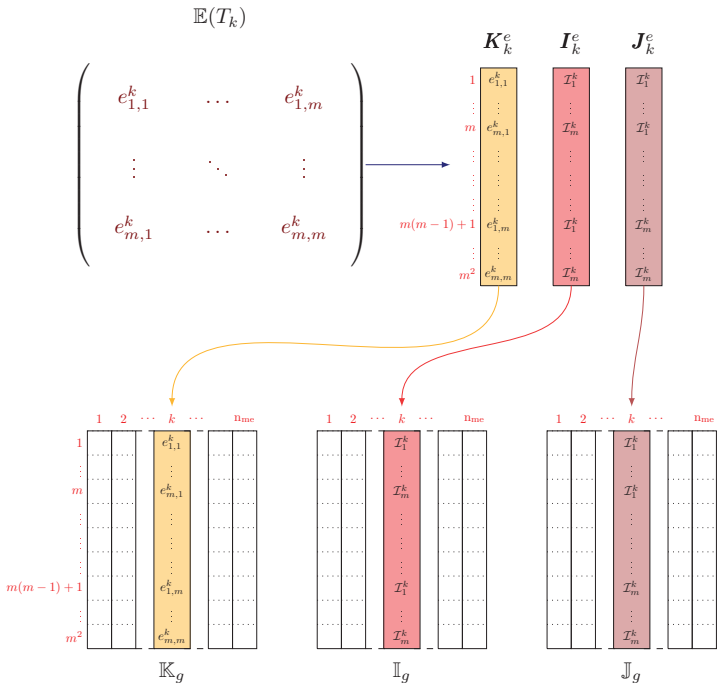

Fig. 4.1: Insertion of a local matrix into global 2D-arrays

Thus, $\mathbb{K}_g$, $\mathbb{I}_g$ and $\mathbb{J}_g$ are defined by: $\forall k \in \{1,\dots,\mathrm{n_{me}}\}$, $\forall l \in \left\{1,\dots,(d+1)^2\right\}$,

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

A natural way to calculate these three arrays is column-wise. In that case, for each array one needs to compute $\mathrm{n_{me}}$ columns.

The `OptV2` method consists in calculating these arrays row-wise. In that case, for each array one needs to calculate $(d+1)^2$ rows (where d is independent of the number of mesh elements). This vectorization method is represented in Figure 4.2.

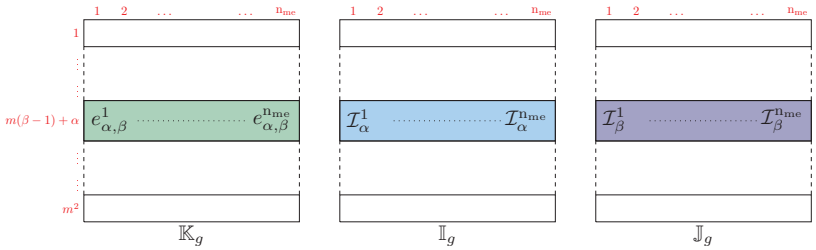

Fig. 4.2: Row-wise operations on global 2D-arrays

We first suppose that for α and β fixed, we can vectorize the computation of $e^k_{\alpha,\beta}$, for all $k \in \{1,\ldots,\mathrm{n_{me}}\}$. This vectorization procedure, denoted by vecElem$(\alpha,\beta,\ldots)$, returns a 1d-array containing these $\mathrm{n_{me}}$ values. We will describe it in detail for some examples in Sections 4.1 and 4.2. Then we obtain the following algorithm

Algorithm 4.1 (`OptV2`) - Optimized and vectorized assembly

Function $\mathbb{M} \leftarrow$ AssemblyGenP1OptV2$(\mathrm{me},\mathrm{n_q},\ldots)$
  $\mathbb{K}_g \leftarrow \mathbb{I}_g \leftarrow \mathbb{J}_g \leftarrow$ zeros$((d+1)^2,\mathrm{n_{me}})$ ▷ $(d+1)^2$-by-$\mathrm{n_{me}}$ 2d-arrays
  $l \leftarrow 1$
  for $\beta \leftarrow 1$ **to** $d+1$ **do**
    for $\alpha \leftarrow 1$ **to** $d+1$ **do**
      $\mathbb{K}_g(l,:) \leftarrow$ vecElem$(\alpha,\beta,\ldots)$
      $\mathbb{I}_g(l,:) \leftarrow \mathrm{me}(\alpha,:)$
      $\mathbb{J}_g(l,:) \leftarrow \mathrm{me}(\beta,:)$
      $l \leftarrow l+1$
    end for
  end for
  $\mathbb{M} \leftarrow$ Sparse$(\mathbb{I}_g(:),\mathbb{J}_g(:),\mathbb{K}_g(:),\mathrm{n_q},\mathrm{n_q})$
end Function

Algorithm 4.1 is efficient in terms of computation time (see Section 6.1). However it is memory consuming due to the size of the arrays $\mathbb{I}_g$, $\mathbb{J}_g$ and $\mathbb{K}_g$. Thus a variant (see [3,23] for dimension 2 or 3 in Matlab) consists in using the sparse command inside the loops (i.e. for each component of all element matrices). This method, called `OptV`, is given in Algorithm 4.2.

Algorithm 4.2 (`OptV`) - Optimized and vectorized assembly (less memory consuming)

1: **Function** $\mathbb{M} \leftarrow$AssemblyGenP1OptV$(\mathrm{me}, \mathrm{n_q}, \ldots)$
2: $\quad \mathbb{M} \leftarrow \mathbb{O}_{\mathrm{n_q}}$ ▷ $\mathrm{n_q}$-by-$\mathrm{n_q}$ sparse matrix
3: $\quad$ **for** $\beta \leftarrow 1$ **to** $d+1$ **do**
4: $\quad\quad$ **for** $\alpha \leftarrow 1$ **to** $d+1$ **do**
5: $\quad\quad\quad \boldsymbol{K}_g \leftarrow$ vecElem$(\alpha, \beta, \ldots)$
6: $\quad\quad\quad \mathbb{M} \leftarrow \mathbb{M} +$ Sparse$(\mathrm{me}(\alpha, :), \mathrm{me}(\beta, :), \boldsymbol{K}_g, \mathrm{n_q}, \mathrm{n_q})$
7: $\quad\quad$ **end for**
8: $\quad$ **end for**
9: **end Function**

For a symmetric matrix, the performance can be improved by using a symmetrized version of `OptV` (called `OptVS`), given in Algorithm 4.3. More precisely, in the lines 3-8 of this algorithm, we build a non-triangular sparse matrix which contains the contributions of the strictly upper parts of all the element matrices. In line 9 the strictly lower part contributions are added using the symmetry of the element matrices. Then in lines 10-13 the contributions of the diagonal parts of the element matrices are added.

Algorithm 4.3 (`OptVS`) - Symmetrized version of `OptV`

1: **Function** $\mathbb{M} \leftarrow$AssemblyGenP1OptVS$(\mathrm{me}, \mathrm{n_q}, \ldots)$
2: $\quad \mathbb{M} \leftarrow \mathbb{O}_{\mathrm{n_q}}$ ▷ $\mathrm{n_q}$-by-$\mathrm{n_q}$ sparse matrix
3: $\quad$ **for** $\alpha \leftarrow 1$ **to** $d+1$ **do**
4: $\quad\quad$ **for** $\beta \leftarrow \alpha+1$ **to** $d+1$ **do**
5: $\quad\quad\quad \boldsymbol{K}_g \leftarrow$ vecElem$(\alpha, \beta, \ldots)$
6: $\quad\quad\quad \mathbb{M} \leftarrow \mathbb{M} +$ Sparse$(\mathrm{me}(\alpha, :), \mathrm{me}(\beta, :), \boldsymbol{K}_g, \mathrm{n_q}, \mathrm{n_q})$
7: $\quad\quad$ **end for**
8: $\quad$ **end for**
9: $\quad \mathbb{M} \leftarrow \mathbb{M} + \mathbb{M}^t$
10: $\quad$ **for** $\alpha \leftarrow 1$ **to** $d+1$ **do**
11: $\quad\quad \boldsymbol{K}_g \leftarrow$ vecElem$(\alpha, \alpha, \ldots)$
12: $\quad\quad \mathbb{M} \leftarrow \mathbb{M} +$ Sparse$(\mathrm{me}(\alpha, :), \mathrm{me}(\alpha, :), \boldsymbol{K}_g, \mathrm{n_q}, \mathrm{n_q})$
13: $\quad$ **end for**
14: **end Function**

In the following, our objective is to show using examples how to vectorize the computation of $\boldsymbol{K}_g$ (i.e. how to obtain the vecElem function in algorithms `OptV2`, `OptV` and `OptVS`). More precisely for the examples derived from (1.1), the calculation of $\boldsymbol{K}_g$ only depends on the local basis functions and/or their gradients and one may need to calculate them on all mesh elements. For $\mathbb{P}_1$ finite elements, these gradients are constant on each d-simplex $K = T_k$. Let $\mathbb{G}$ be the 3D array of size $\mathrm{n_{me}}$-by-$(d+1)$-by-d defined by

$$\mathbb{G}(k, \alpha, :) = \nabla \varphi_\alpha^k(\mathrm{q}), \quad \forall \alpha \in \{1, \ldots, d+1\}, \ \forall k \in \{1, \ldots, \mathrm{n_{me}}\}. \tag{4.1}$$

In Appendix C, we give a vectorized function called GradientVec (see Algorithm C.1) which computes $\mathbb{G}$ in arbitrary dimension. Once the gradients are computed, the local matrices are calculated using the formula (2.4). For simplicity, in the following we consider the `OptV` version. The vectorization of the computation of $\boldsymbol{K}_g$ is shown using the two examples introduced in Section 2.

4.1 Weighted mass matrix assembly

The local weighted mass matrix $\mathbb{M}^{[w],e}(K)$ is given by

$$\mathbb{M}^{[w],e}_{\alpha,\beta}(K) = \int_K w\lambda_\beta\lambda_\alpha d\mathrm{q}, \quad \forall(\alpha,\beta) \in \{1,...,d+1\}^2, \tag{4.2}$$

with $w \in L^\infty(\Omega)$. Generally, this matrix cannot be computed exactly and one has to use a quadrature formula. In the following, we choose to approximate w by $w_h = \pi^1_K(w)$ where $\pi^1_K(w) = \sum_{\gamma=1}^{d+1} w(\mathrm{q}^\gamma)\lambda_\gamma$ is the $\mathbb{P}_1$ Lagrange interpolation of w. Then using (2.4), we have the quadrature formula for (4.2)

$$\int_K \pi^1_K(w)\lambda_\alpha\lambda_\beta d\mathrm{q} = \frac{d!}{(d+3)!}|K|(1+\delta_{\alpha,\beta})(w^s + w(\mathrm{q}^\alpha) + w(\mathrm{q}^\beta)), \tag{4.3}$$

where $w^s = \sum_{\gamma=1}^{d+1} w(\mathrm{q}^\gamma)$. Using (4.3) we vectorize the assembly of the approximate weighted mass matrix (2.1) as shown in Algorithm 4.4.

Algorithm 4.4 (`OptV`) - Weighted mass matrix assembly

```
1:  Function 𝕄 ←AssemblyMassWP1OptV(me, q, vols, w)
2:    w ← w(q)                                          ▷ 1d-array of size n_q
3:    𝕎 ← w(me)                                         ▷ (d+1)-by-n_me 2d-array
4:    w^s ← Sum(𝕎, 1)                                   ▷ 1d-array of size n_me
5:    𝕄 ← 𝕆_{n_q}                                       ▷ n_q-by-n_q sparse matrix
6:    for α ← 1 to d+1 do
7:      for β ← 1 to d+1 do
8:        K_g ← d!/(d+3)! (1+δ_{α,β}) * vols .* (w^s + 𝕎(α,:) + 𝕎(β,:))
9:        𝕄 ← 𝕄 + Sparse(me(α,:), me(β,:), K_g, n_q, n_q)
10:     end for
11:   end for
12: end Function
```

Line 8 of Algorithm 4.4 corresponds to the vectorization of formula (4.3) and is carried out as follows: first we set $\boldsymbol{w} \in \mathbb{R}^{\mathrm{n_q}}$ such that $\boldsymbol{w}(i) = w(\mathrm{q}^i)$, $1 \le i \le \mathrm{n_q}$, or in a vectorized form $\boldsymbol{w} \leftarrow w(\mathrm{q})$. Then we compute the array $\mathbb{W}$ of size $(d+1)$-by-$\mathrm{n_{me}}$ containing, for each d-simplex, the values of w at its vertices: $\mathbb{W}(\alpha,k) = w(\mathrm{q}^{\mathrm{me}(\alpha,k)})$ or in vectorized form $\mathbb{W} \leftarrow \boldsymbol{w}(\mathrm{me})$. We now calculate $\boldsymbol{w}^s \in \mathbb{R}^{\mathrm{n_{me}}}$ which contains, for each d-simplex, the sum of the values of w at its vertices, i.e. we sum $\mathbb{W}$ over the rows and obtain line 4 of Algorithm 4.4. Then, formula (4.3) may be vectorized to obtain line 8 in Algorithm 4.4.

Remark 4.1 Note that formula (4.3) is exact if w is a polynomial of degree 1 on K. Moreover, if w is constant, we get the mass matrix (up to the constant w). Other quadrature rules could be used to approximate the integral in (4.2) without changing the principle of Algorithm 4.4.

Remark 4.2 Algorithm 4.4 can be applied to meshes composed of n-simplices (for $n \le d$) and may be used to compute Neumann or Robin boundary terms.

4.2 Stiffness matrix assembly

The local stiffness matrix $\mathbb{S}^e(K)$ is given, for all $(\alpha,\beta) \in \{1,...,d+1\}^2$, by

$$\mathbb{S}^e_{\alpha,\beta}(K) = \int_K \langle \nabla \lambda_\beta, \nabla \lambda_\alpha \rangle \, d\mathrm{q} = |K| \, \langle \nabla \lambda_\beta, \nabla \lambda_\alpha \rangle \, . \tag{4.4}$$

To obtain the right-hand side of (4.4) we use the fact that the gradients of the local basis functions are constant on each d-simplex. The gradients are computed with the vectorized function GradientVec of Algorithm C.1. Then the vectorized assembly Algorithm 4.5 easily follows.

Algorithm 4.5 (`OptV`) - Stiffness matrix assembly

```
1: Function M ←AssemblyStiffP1OptV(me, q, vols)
2:   G ← GradientVec(q, me)
3:   M ← 0_{n_q}                                   ▷ n_q-by-n_q sparse matrix
4:   for α ← 1 to d + 1 do
5:     for β ← 1 to d + 1 do
6:       K_g ← zeros(1, n_me)
7:       for i ← 1 to d do
8:         K_g ← K_g + G(:, β, i) .* G(:, α, i)
9:       end for
10:      K_g ← K_g .* vols
11:      M ← M + Sparse(me(α, :), me(β, :), K_g, n_q, n_q)
12:    end for
13:  end for
14: end Function
```

We will now adapt these methods to the vector case with an application to the assembly of the elastic stiffness matrix in two and three dimensions.

5 Extension to the vector case

In this section we present an extension of Algorithms 4.1 and 4.2 to the vector case, i.e for a system of m ($m > 1$) partial differential equations such as in elasticity. First, we need to introduce some notation: the space $(X_h^1)^m$ (where X_h^1 is defined in Section 2), is of dimension $\mathrm{n_{dof}} = m\,\mathrm{n_q}$ and spanned by the vector basis functions $\{\boldsymbol{\psi}_{l,i}\}_{\substack{1\le i\le \mathrm{n_q}\\ 1\le l\le m}}$, given by

$$\boldsymbol{\psi}_{l,i} = \varphi_i \boldsymbol{e}_l, \tag{5.1}$$

where $\{\boldsymbol{e}_1, \cdots, \boldsymbol{e}_m\}$ is the standard basis of $\mathbb{R}^m$. The *alternate* numbering is chosen for the basis functions. We use either $\boldsymbol{\psi}_{l,i}$ or $\boldsymbol{\psi}_s$ with $s = (i-1)m + l$ to denote them. We will consider the assembly of a generic sparse matrix of dimension $\mathrm{n_{dof}}$-by-$\mathrm{n_{dof}}$ defined by

$$\mathbb{H}_{r,s} = \int_{\Omega_h} \mathcal{H}(\boldsymbol{\psi}_s, \boldsymbol{\psi}_r) d\mathrm{q},$$

where $\mathcal{H}$ is a bilinear differential operator of order one.

As in the scalar case, in order to vectorize the assembly of the matrix, one has to vectorize the computation of the local matrices. To define the local matrix, we introduce the following notation: on the k-th element $K = T_k$ of $\mathcal{T}_h$ we denote by $\{\boldsymbol{\lambda}_{l,\alpha}\}_{\substack{1\le l\le m\\ 1\le\alpha\le d+1}}$ the $\mathrm{n_{dfe}} = m(d+1)$ local basis functions defined by

$$\boldsymbol{\lambda}_{l,\alpha} = \lambda_\alpha \boldsymbol{e}_l. \tag{5.2}$$

We also use notation $\boldsymbol{\lambda}_i$ with $i = (\alpha-1)m+l$ to denote $\boldsymbol{\lambda}_{l,\alpha}$. By construction, we have $\forall l \in \{1,\dots,m\}$, $\forall \alpha \in \{1,\dots,d+1\}$

$$\boldsymbol{\psi}_{l,\mathrm{me}(\alpha,k)} = \boldsymbol{\lambda}_{l,\alpha} \quad \text{on } K = T_k.$$

Thus, the local matrix $\mathbb{H}^e$ on the d-simplex K is of size $\mathrm{n_{dfe}}$-by-$\mathrm{n_{dfe}}$, and is given by

$$\mathbb{H}^e_{i,j} = \int_K \mathcal{H}(\boldsymbol{\lambda}_j, \boldsymbol{\lambda}_i) d\mathrm{q}.$$

Then, a classical non-vectorized algorithm is given in Algorithm 5.1. The function ElemH is used to calculate the matrix $\mathbb{H}^e$ for a given d-simplex K. As in the scalar case, the vectorized assembly algorithm is based on the use of a function called vecHe which returns the values corresponding to the (i,j)-th entry (with $(i,j) = (m(\alpha-1)+l, m(\beta-1)+n)$) of the local matrices $\mathbb{H}^e(K)$, for all $K \in \mathcal{T}_h$ and for all l, α, n, β. We suppose that this function can be vectorized. Then we obtain the `OptV2` vectorized assembly of the matrix $\mathbb{H}$ given in Algorithm 5.2.

Algorithm 5.1 (`base`) - Classical assembly in vector case ($m > 1$)

```
1:  n_dof ← m * n_q
2:  H ← 0_{n_dof}                                   ▷ Sparse matrix
3:  for k ← 1 to n_me do
4:    H^e ← ElemH(vols(k), ...)
5:    for l ← 1 to m do
6:      for n ← 1 to m do
7:        for α ← 1 to d+1 do
8:          r ← m * (me(α, k) − 1) + l
9:          i ← m * (α − 1) + l
10:         for β ← 1 to d+1 do
11:           s ← m * (me(β, k) − 1) + n
12:           j ← m * (β − 1) + n
13:           H_{r,s} ← H_{r,s} + H^e_{i,j}
14:         end for
15:       end for
16:     end for
17:   end for
18: end for
```

Algorithm 5.2 (`OptV2`) - Optimized assembly in vector case ($m > 1$)

```
1:  n_dfe ← m * (d + 1)
2:  K_g ← I_g ← J_g ← zeros(n_dfe^2, n_me)
3:  p ← 1
4:  for l ← 1 to m do
5:    for n ← 1 to m do
6:      for β ← 1 to d+1 do
7:        for α ← 1 to d+1 do
8:          K_g(p, :) ← vecHe(l, α, n, β, ...)
9:          I_g(p, :) ← m * (me(α, :) − 1) + l
10:         J_g(p, :) ← m * (me(β, :) − 1) + n
11:         p ← p + 1
12:       end for
13:     end for
14:   end for
15: end for
16: n_dof ← m * n_q
17: H ← sparse(I_g(:), J_g(:), K_g(:), n_dof, n_dof)
```

As in Section 4, although Algorithm 5.2 is efficient in terms of computation time, it is memory consuming due to the size of the arrays $\mathbb{I}_g$, $\mathbb{J}_g$ and $\mathbb{K}_g$. Thus a variant consists in using the sparse command inside the loops, which leads to the extension of the `OptV` algorithm to the vector case, given in Algorithm 5.3.

Algorithm 5.3 (`OptV`) - Optimized assembly in vector case ($m > 1$)

Function $\mathbb{M} \leftarrow$ AssemblyVecGenP1OptV(me, $\mathrm{n_q}, \ldots$)
  $\mathrm{n_{dof}} \leftarrow m * \mathrm{n_q}$
  $\mathbb{M} \leftarrow \mathbb{O}_{\mathrm{n_{dof}}}$ ▷ $\mathrm{n_{dof}}$-by-$\mathrm{n_{dof}}$ sparse matrix
  for $l \leftarrow 1$ **to** m **do**
    for $\alpha \leftarrow 1$ **to** $d+1$ **do**
      $\boldsymbol{I}_g \leftarrow m * (\mathrm{me}(\alpha, :) - 1) + l$
      for $n \leftarrow 1$ **to** m **do**
        for $\beta \leftarrow 1$ **to** $d+1$ **do**
          $\boldsymbol{K}_g \leftarrow$ vecHe$(l, \alpha, n, \beta, \ldots)$
          $\boldsymbol{J}_g \leftarrow m * (\mathrm{me}(\beta, :) - 1) + n$
          $\mathbb{M} \leftarrow \mathbb{M} +$ Sparse$(\boldsymbol{I}_g, \boldsymbol{J}_g, \boldsymbol{K}_g, \mathrm{n_{dof}}, \mathrm{n_{dof}})$
        end for
      end for
    end for
  end for
end Function

For a symmetric matrix, the performance can be improved by using a symmetrized version of Algorithm 5.3 (as in Section 4), given in Algorithm C.2.

In the following the vectorized function vecHe is detailed for the elastic stiffness matrix in 2D and 3D.

5.1 Elastic stiffness matrix assembly

Here we consider sufficiently regular vector fields $\boldsymbol{u} = (u_1, \ldots, u_d) : \Omega \to \mathbb{R}^d$, with the associated discrete space $(X_h^1)^d$, $d = 2$ or 3 (i.e. $m = d$ in that case).

We consider 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]). This sparse matrix $\mathbb{K}$ is defined by

$$\mathbb{K}_{l,n} = \int_{\Omega_h} \underline{\boldsymbol{\epsilon}}^t(\boldsymbol{\psi}_n) \mathbb{C} \underline{\boldsymbol{\epsilon}}(\boldsymbol{\psi}_l) d\mathrm{q}, \quad \forall (l, n) \in \{1, \ldots, \mathrm{n_{dof}}\}^2, \tag{5.3}$$

where $\underline{\boldsymbol{\epsilon}}$ is the linearized strain tensor given by

$$\underline{\boldsymbol{\epsilon}}(\boldsymbol{u}) = \frac{1}{2} \left(\boldsymbol{\nabla}(\boldsymbol{u}) + \boldsymbol{\nabla}^t(\boldsymbol{u}) \right),$$

with $\underline{\boldsymbol{\epsilon}} = (\epsilon_{11}, \epsilon_{22}, 2\epsilon_{12})^t$ in 2D and $\underline{\boldsymbol{\epsilon}} = (\epsilon_{11}, \epsilon_{22}, \epsilon_{33}, 2\epsilon_{12}, 2\epsilon_{23}, 2\epsilon_{13})^t$ in 3D, with $\epsilon_{ij}(\boldsymbol{u}) = \frac{1}{2} \left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i} \right)$. The elasticity tensor $\mathbb{C}$ depends on the Lamé

parameters λ and μ satisfying $\lambda+\mu>0$, and possibly variable in Ω. For $d=2$ or $d=3$, the matrix $\mathbb{C}$ is given by

$$\mathbb{C}=\begin{pmatrix}\lambda\mathbb{1}_2+2\mu\mathbb{I}_2 & \mathbb{O}_{2\times1}\\ \mathbb{O}_{1\times2} & \mu\end{pmatrix}_{3\times3},\qquad \mathbb{C}=\begin{pmatrix}\lambda\mathbb{1}_3+2\mu\mathbb{I}_3 & \mathbb{O}_{3\times3}\\ \mathbb{O}_{3\times3} & \mu\mathbb{I}_3\end{pmatrix}_{6\times6}.$$

Formula (5.3) is related to the Hooke's law

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

where $\underline{\boldsymbol{\sigma}}$ is the elastic stress tensor.

The vectorization of the assembly of the elastic stiffness matrix (5.3) will be carried out as in Section 4, through the vectorization of the local elastic stiffness matrix $\mathbb{K}^e$ given for all $(i,j)\in\{1,\ldots,\mathrm{n_{dfe}}\}^2$ by

$$\mathbb{K}^e_{i,j}(K)=\int_K \underline{\boldsymbol{\epsilon}}^t(\boldsymbol{\lambda}_j)\mathbb{C}\underline{\boldsymbol{\epsilon}}(\boldsymbol{\lambda}_i)d\mathrm{q}, \tag{5.4}$$

or equivalently, using (5.2), we have for $1\le\alpha,\beta\le d+1$ and $1\le l,n\le m$

$$\mathbb{K}^e_{i,j}(K)=\int_K \underline{\boldsymbol{\epsilon}}^t(\boldsymbol{\lambda}_{n,\beta})\mathbb{C}\underline{\boldsymbol{\epsilon}}(\boldsymbol{\lambda}_{l,\alpha})d\mathrm{q}, \tag{5.5}$$

with $i=(\alpha-1)d+l$ and $j=(\beta-1)d+n$. The vectorization of $\mathbb{K}^e$ is based on the following result:

Lemma 5.1 *There exist two matrices $\mathbb{Q}^{n,l}$ and $\mathbb{S}^{n,l}$ of size d-by-d depending only on n and l such that*

$$\underline{\boldsymbol{\epsilon}}^t(\boldsymbol{\lambda}_{n,\beta})\mathbb{C}\underline{\boldsymbol{\epsilon}}(\boldsymbol{\lambda}_{l,\alpha})=\lambda\left\langle\nabla\lambda_\beta,\mathbb{Q}^{n,l}\nabla\lambda_\alpha\right\rangle+\mu\left\langle\nabla\lambda_\beta,\mathbb{S}^{n,l}\nabla\lambda_\alpha\right\rangle. \tag{5.6}$$

The proof of Lemma 5.1 is given in Appendix B.
Using (5.6) in (5.5), we have

$$\mathbb{K}^e_{i,j}(K)=\left\langle\nabla\lambda_\beta,\mathbb{Q}^{n,l}\nabla\lambda_\alpha\right\rangle\int_K\lambda d\mathrm{q}+\left\langle\nabla\lambda_\beta,\mathbb{S}^{n,l}\nabla\lambda_\alpha\right\rangle\int_K\mu d\mathrm{q}.$$

One possibility is to approximate the Lamé parameters λ and μ by their $\mathbb{P}_1$ finite element interpolation $\pi^1_K(\lambda)$ and $\pi^1_K(\mu)$, respectively (we consider $\mathbb{P}_1$ instead of $\mathbb{P}_0$ to illustrate better the vectorization, the latter being a special case of the former). Then we have

$$\mathbb{K}^e_{i,j}(K)\approx\frac{|K|}{d+1}\left(\left\langle\nabla\lambda_\beta,\mathbb{Q}^{n,l}\nabla\lambda_\alpha\right\rangle\lambda^s+\left\langle\nabla\lambda_\beta,\mathbb{S}^{n,l}\nabla\lambda_\alpha\right\rangle\mu^s\right), \tag{5.7}$$

with $\lambda^s=\sum_{\gamma=1}^{d+1}\lambda(\mathrm{q}^\gamma)$ and $\mu^s=\sum_{\gamma=1}^{d+1}\mu(\mathrm{q}^\gamma)$. The previous formula may now be vectorized as shown in Algorithm 5.4. This algorithm is based on the vectorization of the computation of the terms $\langle\nabla\lambda_\beta,\mathbb{A}\nabla\lambda_\alpha\rangle$, which is carried out with the function dotMatVecG in Algorithm 5.5, for any d-by-d matrix $\mathbb{A}$ independent of the d-simplices of the mesh. In this algorithm, $\mathbb{G}$ is the array

of gradients defined in (4.1), α and β are indices in $\{1,\dots,d+1\}$, and $\boldsymbol{X}$ is a 1-by-$\mathrm{n_{me}}$ array such that $\boldsymbol{X}(k) = \langle \nabla\lambda_\beta, \mathbb{A}\nabla\lambda_\alpha\rangle$ on $K = T_k$.

Algorithm 5.4 Elastic stiffness matrix assembly - `OptV` version

Function $\mathbb{M} \leftarrow$ AssemblyStiffElasP1OptV(me, q, vols, `lamb`, `mu`)
  $[\mathbb{Q},\mathbb{S}] \leftarrow$ MatQS(d) ▷ $\mathbb{Q}$,$\mathbb{S}$: 2d array of matrices with $\mathbb{Q}(l,n) = \mathbb{Q}^{l,n}$
  `Lambs` $\leftarrow$ sum(`lamb`(me), 1) `.*` vols$/(d+1)$
  `Mus` $\leftarrow$ sum(`mu`(me), 1) `.*` vols$/(d+1)$
  $\mathbb{G} \leftarrow$ GradientVec(q, me)
  $\mathrm{n_{dof}} \leftarrow m * \mathrm{n_q}$, $\mathbb{M} \leftarrow \mathbb{O}_{\mathrm{n_{dof}}}$ ▷ $\mathrm{n_{dof}}$-by-$\mathrm{n_{dof}}$ sparse matrix
  for $l \leftarrow 1$ **to** d **do**
    for $\alpha \leftarrow 1$ **to** $d+1$ **do**
      $\boldsymbol{I}_g \leftarrow m * (\mathrm{me}(\alpha,:) - 1) + l$
      for $n \leftarrow 1$ **to** d **do**
        for $\beta \leftarrow 1$ **to** $d+1$ **do**
          $\boldsymbol{K}_g \leftarrow$ `Lambs` `.*` dotMatVecG($\mathbb{Q}(l,n), \mathbb{G}, \alpha, \beta$)
            +`Mus` `.*` dotMatVecG($\mathbb{S}(l,n), \mathbb{G}, \alpha, \beta$)
          $\boldsymbol{J}_g \leftarrow m * (\mathrm{me}(\beta,:) - 1) + n$
          $\mathbb{M} \leftarrow \mathbb{M} +$ Sparse($\boldsymbol{I}_g, \boldsymbol{J}_g, \boldsymbol{K}_g, \mathrm{n_{dof}}, \mathrm{n_{dof}}$)
        end for
      end for
    end for
  end for
end Function

Algorithm 5.5 Vectorization of $\boldsymbol{X}$ in dimension d

Function $\boldsymbol{X} \leftarrow$ dotMatVecG($\mathbb{A}, \mathbb{G}, \alpha, \beta$)
  $\boldsymbol{X} \leftarrow$ zeros$(1, \mathrm{n_{me}})$
  for $i \leftarrow 1$ **to** d **do**
    for $j \leftarrow 1$ **to** d **do**
      $\boldsymbol{X} \leftarrow \boldsymbol{X} + \mathbb{A}(j,i) * (\mathbb{G}(:,\alpha,i)$ `.*` $\mathbb{G}(:,\beta,j))$
    end for
  end for
end Function

From Algorithm 5.4, it is straightforward to derive Algorithm C.2 which uses the symmetry when the assembly matrix is symmetric.

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

6 Benchmark results

We consider the assembly of the stiffness and elastic stiffness matrices in 2D and 3D, in the following vector languages

- Matlab (R2014b),
- Octave (3.8.1),
- Python 3.4.0 with *NumPy[1.8.2]* and *SciPy[0.13.3]*.

We first compare the computation times of the different codes (`base`, `OptV1`, `OptV2`, `OptV` and `OptVS`), for each language considered. Then we compare `OptVS` code with a C implementation of the assembly using the *SuiteSparse* library 4.2.1 [10] ("CXSparse") and with FreeFEM++. A comparison of the performance of the `OptVS` code with recent and efficient Matlab/Octave codes is also given. In every benchmark the domain Ω is the unit disk in 2D and the unit sphere in 3D. For each result we present the average computation time for at least five finite element assembly calculations.

6.1 Comparison of the `base`, `OptV1`, `OptV2`, `OptV` and `OptVS` assembly codes

We show in Figures 6.1 and 6.2, in logarithmic scales and for each vector language, the performance of the assembly codes versus the matrix dimension n_{dof}, for the 2D stiffness and 3D elastic stiffness matrices respectively. We observe that the `OptVS` version is the fastest one and its complexity is $\mathcal{O}(n_{dof})$.

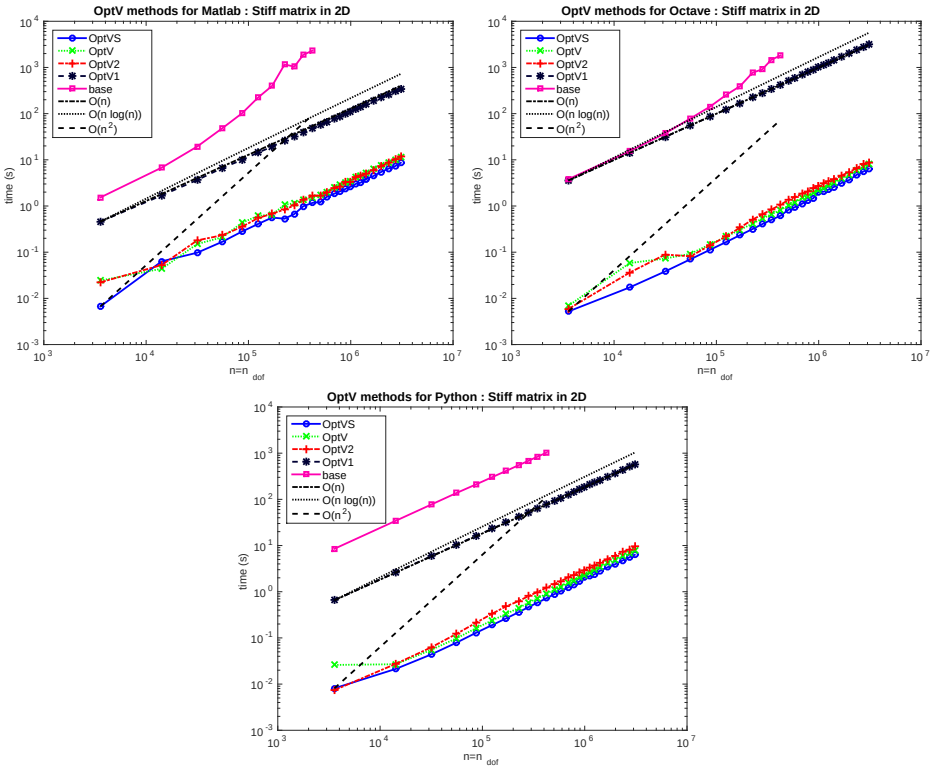

Fig. 6.1: Stiffness matrix (2D): comparison of `base`, `OptV1`, `OptV2`, `OptV` and `OptVS` codes in Matlab (top left), Octave (top right) and Python (bottom).

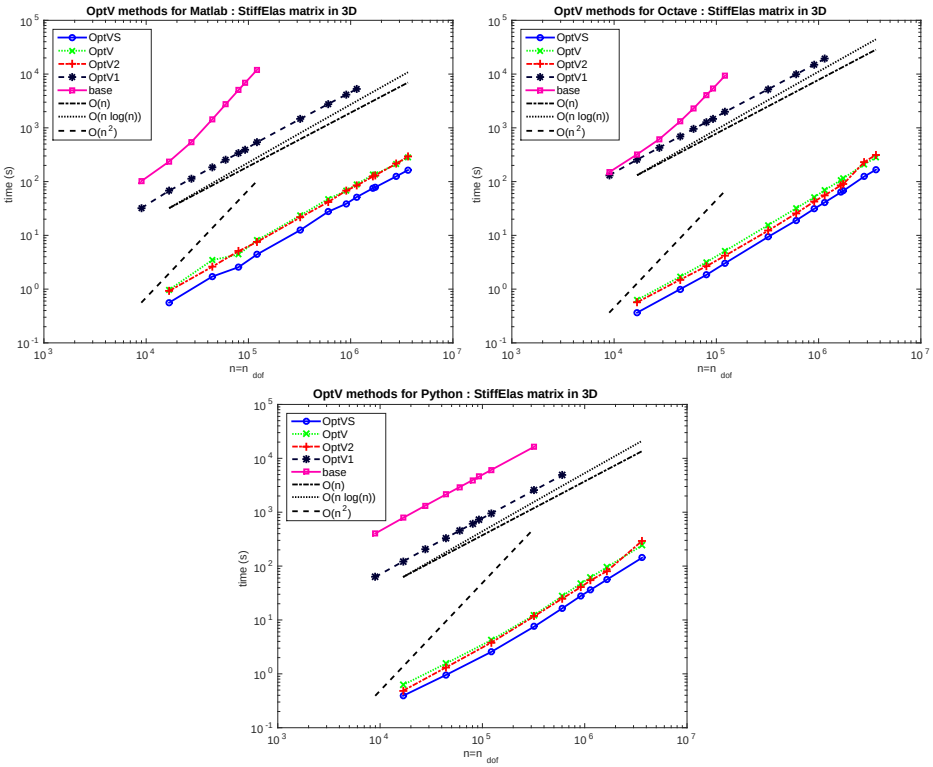

Fig. 6.2: Elastic stiffness matrix (3D): comparison of `base`, `OptV1`, `OptV2`, `OptV` and `OptVS` codes in Matlab (top left), Octave (top right) and Python (bottom).

For the stiffness matrix in 2D, the `OptV1` version is about 40, 95 and 550 times slower in Matlab, Python and Octave respectively. Its numerical complexity is $\mathcal{O}(n_{dof})$. The complexity of the less performing method, the `base` version, is $\mathcal{O}(n_{dof}^2)$ in Matlab and Octave, while it seems to be $\mathcal{O}(n_{dof})$ in Python. This is partly due to the use of the `LIL` format in the sparse matrix assembly in Python, the conversion to the `CSC` format being included in the computation time. We obtain similar results for the stiffness matrix in 3D and the elastic stiffness matrices in 2D and 3D. Computation times and `OptVS` speedup are given in Tables 6.1 and 6.2 for the 2D stiffness and 3D elastic stiffness matrices respectively. For the 3D stiffness and the 2D elastic stiffness matrices one can refer respectively to Tables A.3 and A.4. We observe that the performance differences of the stiffness and elastic stiffness matrix assemblies in 2D and 3D are partly 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. For matrices of the same size (i.e. for an equal n_{dof}), in comparison to the 2D stiffness matrix, the number of local values to be computed are 2, 4 and 16 times larger for the 2D elastic stiffness, the 3D stiffness and the 3D elastic stiffness matrices respectively.

StiffAssembling2DP1 - Matlab

n_{dof}	OptVS	OptV	OptV2	OptV1	base
14222	.063 (s) x 1	.044 (s) x .692	.053 (s) x .833	1.70 (s) x 26.8	6.79 (s) x 107
125010	.411 (s) x 1	.617 (s) x 1.5	.553 (s) x 1.35	14.4 (s) x 35.1	226 (s) x 550
343082	.985 (s) x 1	1.37 (s) x 1.39	1.36 (s) x 1.38	39.1 (s) x 39.7	1873 (s) x 1902
885521	2.34 (s) x 1	3.24 (s) x 1.39	3.29 (s) x 1.41	99.7 (s) x 42.7	-
1978602	5.45 (s) x 1	7.60 (s) x 1.40	7.28 (s) x 1.34	223 (s) x 40.9	-

StiffAssembling2DP1 - Octave

n_{dof}	OptVS	OptV	OptV2	OptV1	base
14222	.017 (s) x 1	.058 (s) x 3.36	.036 (s) x 2.09	14.3 (s) x 826	15.4 (s) x 888
125010	.167 (s) x 1	.218 (s) x 1.31	.221 (s) x 1.33	124 (s) x 742	255 (s) x 1533
343082	.499 (s) x 1	.656 (s) x 1.32	.835 (s) x 1.67	340 (s) x 681	1458 (s) x 2923
885521	1.47 (s) x 1	1.91 (s) x 1.30	2.43 (s) x 1.65	899 (s) x 613	-
1978602	3.64 (s) x 1	4.63 (s) x 1.27	5.44 (s) x 1.49	2007 (s) x 551	-

StiffAssembling2DP1 - Python

n_{dof}	OptVS	OptV	OptV2	OptV1	base
14222	.021 (s) x 1	.027 (s) x 1.26	.027 (s) x 1.29	2.64 (s) x 124	34.4 (s) x 1614
125010	.190 (s) x 1	.241 (s) x 1.26	.336 (s) x 1.77	23.2 (s) x 122	303 (s) x 1594
343082	.576 (s) x 1	.716 (s) x 1.24	.980 (s) x 1.70	63.5 (s) x 110	833 (s) x 1445
885521	1.66 (s) x 1	2.05 (s) x 1.23	2.62 (s) x 1.58	164 (s) x 98.9	-
1978602	3.92 (s) x 1	4.85 (s) x 1.24	6.04 (s) x 1.54	368 (s) x 93.9	-

Table 6.1: Stiffness matrix (2D) : comparison of `OptVS`, `OptV`, `OptV1` and `base` codes in Matlab (top left), Octave (top right) and Python (bottom) giving time in seconds (top value) and `OptVS` speedup (bottom value).

StiffElasAssembling3DP1 - Matlab

n_{dof}	OptVS	OptV	OptV2	OptV1	base
16773	.560 (s) x 1	.971 (s) x 1.73	.924 (s) x 1.65	67.6 (s) x 121	236 (s) x 422
44124	1.70 (s) x 1	3.45 (s) x 2.03	2.60 (s) x 1.52	184 (s) x 108	1427 (s) x 837
121710	4.43 (s) x 1	8.12 (s) x 1.83	7.55 (s) x 1.70	540 (s) x 122	1E+4 (s) x 2716
601272	27.5 (s) x 1	47.4 (s) x 1.72	41.5 (s) x 1.51	2765 (s) x 101	-
1144680	51.5 (s) x 1	89.4 (s) x 1.74	84.2 (s) x 1.64	5254 (s) x 102	-

StiffElasAssembling3DP1 - Octave

n_{dof}	OptVS	OptV	OptV2	OptV1	base
16773	.364 (s) x 1	.628 (s) x 1.73	.569 (s) x 1.56	255 (s) x 701	321 (s) x 882
44124	.993 (s) x 1	1.69 (s) x 1.71	1.49 (s) x 1.50	698 (s) x 703	1314 (s) x 1323
121710	3.03 (s) x 1	5.13 (s) x 1.69	4.19 (s) x 1.38	1976 (s) x 651	9338 (s) x 3078
601272	18.9 (s) x 1	31.7 (s) x 1.68	25.5 (s) x 1.35	9853 (s) x 521	-
1144680	40.9 (s) x 1	69.1 (s) x 1.69	55.6 (s) x 1.36	2E+4 (s) x 471	-

StiffElasAssembling3DP1 - Python

n_{dof}	OptVS	OptV	OptV2	OptV1	base
16773	.391 (s) x 1	.622 (s) x 1.59	.486 (s) x 1.24	122 (s) x 312	784 (s) x 2004
44124	.954 (s) x 1	1.56 (s) x 1.63	1.32 (s) x 1.38	333 (s) x 349	2141 (s) x 2243
121710	2.55 (s) x 1	4.21 (s) x 1.65	3.79 (s) x 1.49	946 (s) x 372	6071 (s) x 2384
601272	16.4 (s) x 1	27.6 (s) x 1.68	24.9 (s) x 1.52	4850 (s) x 296	-
1144680	36.4 (s) x 1	61.5 (s) x 1.69	54.2 (s) x 1.49	-	-

Table 6.2: Elastic stiffness matrix (3D) : comparison of `OptVS`, `OptV`, `OptV2`, `OptV1` and `base` codes in Matlab (top left), Octave (top right) and Python (bottom) giving time in seconds (top value) and `OptVS` speedup (bottom value).

In Figure 6.3 we compare the maximum of memory for `OptVS`, `OptV` and `OptV2` codes. The `OptV2` method is more consuming than `OptVS` and `OptV` respectively by a factor between 5 and 6.3 and between 6 and 8.9 depending on the language.

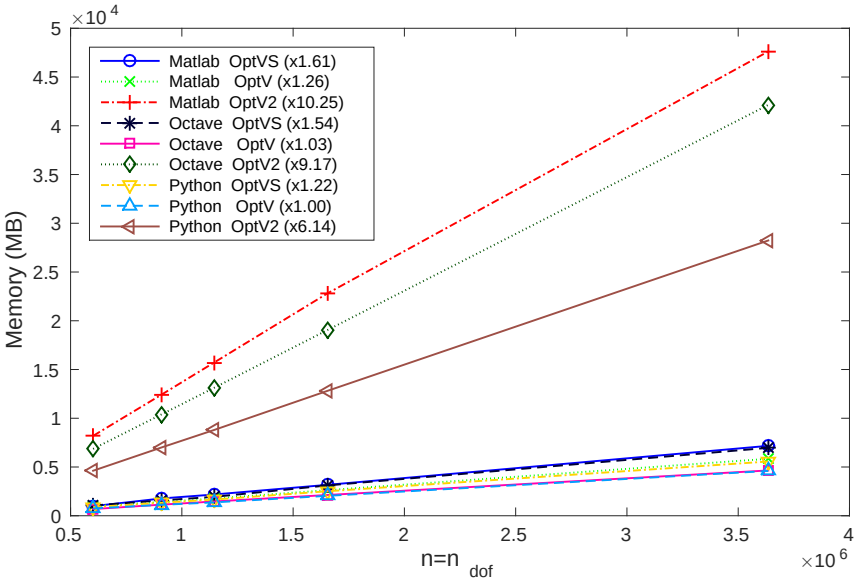

Fig. 6.3: Elastic stiffness matrix (3D): memory usage in MB and ratio between the slope of each method and `OptV` in Python (in the caption)

6.2 Comparison of the `OptVS` version with CXSparse and FreeFEM++

In Tables 6.3 the `OptVS` codes in Matlab/Octave/Python are compared with a C implementation of the assembly (`OptV1` version) using the *SuiteSparse* library [10] ("CXSparse") and with a FreeFEM++ code for the stiffness matrix in 2D and the elastic stiffness matrix in 3D.

The computation cost for the stiffness matrix in 3D and the elastic stiffness matrix in 2D are given in Tables A.1 and A.2. We observe that `OptVS` version is approximately 1.5 and 5.5 times in Matlab, 2 and 7.5 times in Octave, and 2.1 and 8.5 times in Python faster than FreeFEM++. Compared to C, computation times are multiplied by a factor between 2.5 and 4.7 in Matlab, 1.9 and 3.7 in Octave, and 1.8 and 3.2 in Python. Unlike what is commonly believed the performance is not radically worse than that of C.

n_{dof}	CXSparse (4.2.1)	Matlab (2014b)	Octave (3.8.1)	Python (3.4.0)	FreeFEM (3.31)
14222	0.014 (s) x 1.00	0.063 (s) x 4.64	0.017 (s) x 1.26	0.021 (s) x 1.56	0.071 (s) x 5.16
125010	0.073 (s) x 1.00	0.411 (s) x 5.62	0.167 (s) x 2.28	0.190 (s) x 2.60	0.501 (s) x 6.85
343082	0.221 (s) x 1.00	0.985 (s) x 4.46	0.499 (s) x 2.26	0.576 (s) x 2.61	1.421 (s) x 6.43
885521	0.606 (s) x 1.00	2.337 (s) x 3.86	1.467 (s) x 2.42	1.660 (s) x 2.74	3.692 (s) x 6.10
1978602	1.354 (s) x 1.00	5.446 (s) x 4.02	3.644 (s) x 2.69	3.920 (s) x 2.89	8.305 (s) x 6.13

n_{dof}	CXSparse (4.2.1)	Matlab (2014b)	Octave (3.8.1)	Python (3.4.0)	FreeFEM (3.31)
16773	0.137 (s) x 1.00	0.560 (s) x 4.09	0.364 (s) x 2.66	0.391 (s) x 2.86	3.827 (s) x 27.95
44124	0.398 (s) x 1.00	1.705 (s) x 4.29	0.993 (s) x 2.50	0.954 (s) x 2.40	10.440 (s) x 26.26
121710	1.193 (s) x 1.00	4.433 (s) x 3.72	3.034 (s) x 2.54	2.547 (s) x 2.13	29.914 (s) x 25.08
601272	6.386 (s) x 1.00	27.482 (s) x 4.30	18.894 (s) x 2.96	16.359 (s) x 2.56	152.553 (s) x 23.89
1144680	12.477 (s) x 1.00	51.465 (s) x 4.12	40.940 (s) x 3.28	36.392 (s) x 2.92	293.307 (s) x 23.51

Table 6.3: 2D Stiffness matrix (top table) and 3D elastic stiffness matrix (bottom table) : computational cost versus n_{dof}, with the `OptVS` Matlab/Octave/Python version ($2^{\text{nd}}/3^{\text{rd}}/4^{\text{th}}$ columns), with CXSparse (1^{st} column) and FreeFEM++ (5^{th} column); time in seconds (top value) and CXSparse speedup (bottom value).

6.3 Comparison with other matrix assemblies in Matlab and Octave

In Matlab/Octave other efficient algorithms have been proposed recently in [2, 3, 17, 34]. More precisely, in [17], 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 [34], the implementation is based on extending element operations on arrays into operations on arrays of matrices, calling them matrix-array operations. The array elements are matrices instead of 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 [3], for the assembly of the stiffness matrix in 2D associated to $\mathbb{P}_1$ finite elements, L. Chen constructs 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. The restriction to $d = 2$ or 3 of Algorithm 4.2 corresponds to the method in [3].

We compare these codes to `OptVS` for the assembly of the stiffness matrix in 2D. In Tables 6.4 and 6.5, using Matlab and Octave respectively, computation times versus the number of vertices are given for the different codes. `OptVS` speedup is between 1 and 2.5 in comparison with the other vectorized codes for sufficiently fine meshes.

n_{dof}	OptVs	Chen	iFEM	HanJun	RahVal
125010	0.411 (s) x 1.00	0.616 (s) x 1.50	0.693 (s) x 1.69	0.646 (s) x 1.57	0.664 (s) x 1.61
343082	0.985 (s) x 1.00	1.464 (s) x 1.49	1.257 (s) x 1.28	1.989 (s) x 2.02	2.096 (s) x 2.13
885521	2.337 (s) x 1.00	3.307 (s) x 1.41	2.966 (s) x 1.27	4.372 (s) x 1.87	4.721 (s) x 2.02
1978602	5.446 (s) x 1.00	9.286 (s) x 1.71	7.221 (s) x 1.33	9.813 (s) x 1.80	9.123 (s) x 1.68
3085628	8.644 (s) x 1.00	12.332 (s) x 1.43	11.444 (s) x 1.32	14.562 (s) x 1.68	14.841 (s) x 1.72

Table 6.4: Stiffness matrix (2D): computational cost in Matlab (R2014b) versus n_q, with the `OptVS` version (column 2) and with the codes in [2,3,17,34] (columns 3-6) : time in seconds (top value) and speedup (bottom value). The speedup reference is `OptVS` version.

n_{dof}	OptVs	Chen	iFEM	HanJun	RahVal
125010	0.167 (s) x 1.00	0.305 (s) x 1.83	0.288 (s) x 1.73	0.417 (s) x 2.50	0.486 (s) x 2.92
343082	0.499 (s) x 1.00	0.823 (s) x 1.65	0.644 (s) x 1.29	1.299 (s) x 2.60	1.245 (s) x 2.50
885521	1.467 (s) x 1.00	2.123 (s) x 1.45	1.663 (s) x 1.13	3.720 (s) x 2.54	3.221 (s) x 2.20
1978602	3.644 (s) x 1.00	4.674 (s) x 1.28	3.832 (s) x 1.05	8.279 (s) x 2.27	7.164 (s) x 1.97
3085628	6.457 (s) x 1.00	7.786 (s) x 1.21	6.642 (s) x 1.03	13.523 (s) x 2.09	11.583 (s) x 1.79

Table 6.5: Stiffness matrix (2D): computational cost in Octave (3.8.1) versus n_q, with the `OptVS` version (column 2) and with the codes in [2,3,17,34] (columns 3-6) : time in seconds (top value) and speedup (bottom value). The speedup reference is `OptVS` version.

In Figure 6.4 we compare the memory costs in Matlab of our assembly codes with the other ones. As expected the consumption of `OptVS` and `OptV` methods are observed to be close to that of `iFEM` and lower than that of the other codes.

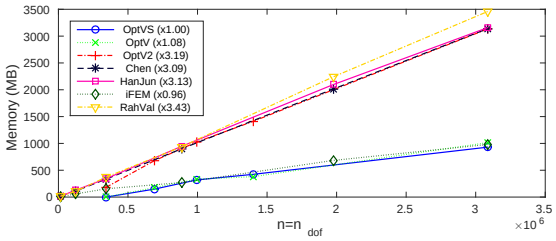

Fig. 6.4: Stiffness matrix (2D): memory usage in MB and ratio between the slope of each method and `OptVS` (in the caption)

7 Conclusion and work in progress

We presented vectorized algorithms for the assembly of $\mathbb{P}_1$ finite element matrices in arbitrary dimension. The implementation of these algorithms has been done in different vector languages such as Matlab, Octave and Python to calculate the stiffness and elastic stiffness matrices. Computation times of different versions (vectorized or not) have been compared in several interpreted languages and C. Numerical examples show the efficiency of the `OptV2`, `OptV` and `OptVS` algorithms. More precisely, for the `OptVS` method, the assembly of the stiffness matrix in 2D of size 10^6 is performed in 2.6, 1.75 and 2 seconds with Matlab, Octave and Python respectively and in 0.75 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 45, 35.8 and 31.8 seconds, with Matlab, Octave and Python respectively and in 10.9 seconds with C. Moreover we observed that `OptVS` is about 1.4 times faster than the non-symmetrized versions `OptV` and `OptV2`. `OptV` and `OptVS` methods are less memory consuming than `OptV2`. Preliminary results towards the extension to $\mathbb{P}_k$ finite elements are given in the Appendix. The algorithms in arbitrary dimension for piecewise polynomials of higher order, is the subject of a future paper. The `OptV2` algorithm has been also implemented with a NVIDIA GPU[3], using the *Thrust* and *Cusp* libraries. For the 2D elastic stiffness and 3D stiffness matrices, the `OptV2` code is respectively 3.5 and 7 times faster on GPU than the C code (the time for GPU/CPU data and matrix transfers is taken into account).

Vectorization gave good performance and the vectorized code can be used for other matrices or discretizations, the only part of the code that have to be reviewed (which is probably the most difficult part) is the vectorization of the element matrix computation. We have seen that it is possible to efficiently assemble matrices of large size in interpreted languages. In this framework Python showed some very good performance even though Octave seems to be more efficient in some cases. Moreover the performance of our vectorized codes was better in Octave than in Matlab. The Python and Matlab/Octave codes are available online (see [7]).

[3] GeForce GTX Titan Black, 2880 CUDA Core, 6Go Memory

A Additional benchmark results

In this section, we consider the assembly of the 3D stiffness and 2D elastic stiffness matrices. In Tables A.1 and A.2 we compare the `OptVS` 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. In Tables A.3 and A.4 the computation times of `OptVS`, `OptV`, `OptV2`, `OptV1` and `base` versions are compared in Matlab, Octave and Python. We observe similar results as in Section 6.

n_{dof}	CXSparse (4.2.1)	Matlab (2014b)	Octave (3.8.1)	Python (3.4.0)	FreeFEM (3.31)
14708	0.087 (s) x 1.00	0.135 (s) x 1.55	0.143 (s) x 1.64	0.191 (s) x 2.19	0.545 (s) x 6.26
40570	0.155 (s) x 1.00	0.461 (s) x 2.97	0.293 (s) x 1.89	0.377 (s) x 2.43	1.571 (s) x 10.12
200424	1.014 (s) x 1.00	3.457 (s) x 3.41	1.911 (s) x 1.88	1.935 (s) x 1.91	8.742 (s) x 8.62
580975	3.844 (s) x 1.00	9.767 (s) x 2.54	7.193 (s) x 1.87	6.804 (s) x 1.77	26.970 (s) x 7.02
1747861	10.752 (s) x 1.00	31.203 (s) x 2.90	31.008 (s) x 2.88	26.069 (s) x 2.42	84.698 (s) x 7.88

Table A.1: Stiffness matrix (3D) : computational cost versus n_{dof}, with the `OptVS` Matlab/Octave/Python version ($2^{\text{nd}}/3^{\text{rd}}/4^{\text{th}}$ columns), with CXSparse (1^{st} column) and FreeFEM++ (5^{th} column) : time in seconds (top value) and speedup (bottom value). The speedup reference is CXSparse code.

n_{dof}	CXSparse (4.2.1)	Matlab (2014b)	Octave (3.8.1)	Python (3.4.0)	FreeFEM (3.31)
28444	0.023 (s) x 1.00	0.139 (s) x 5.92	0.115 (s) x 4.88	0.076 (s) x 3.25	0.762 (s) x 32.47
111838	0.107 (s) x 1.00	0.412 (s) x 3.84	0.321 (s) x 2.99	0.288 (s) x 2.69	2.865 (s) x 26.75
250020	0.267 (s) x 1.00	1.046 (s) x 3.92	0.727 (s) x 2.73	0.662 (s) x 2.48	6.432 (s) x 24.11
1013412	1.133 (s) x 1.00	5.000 (s) x 4.41	4.238 (s) x 3.74	3.377 (s) x 2.98	26.301 (s) x 23.20
2802258	3.142 (s) x 1.00	14.867 (s) x 4.73	11.483 (s) x 3.65	10.036 (s) x 3.19	72.561 (s) x 23.10

Table A.2: Elastic stiffness matrix (2D) : computational cost versus n_{dof}, with the `OptVS` Matlab/Octave/Python version ($2^{\text{nd}}/3^{\text{rd}}/4^{\text{th}}$ columns), with CXSparse (1^{st} column) and FreeFEM++ (5^{th} column) : time in seconds (top value) and speedup (bottom value). The speedup reference is CXSparse code.

StiffAssembling3DP1 - Matlab

n_{dof}	OptVS	OptV	OptV2	OptV1	base
14708	.135 (s) x 1	.409 (s) x 3.03	.442 (s) x 3.27	5.07 (s) x 37.5	25.6 (s) x 189
40570	.461 (s) x 1	.827 (s) x 1.80	.775 (s) x 1.68	14.0 (s) x 30.3	112 (s) x 244
200424	3.46 (s) x 1	4.69 (s) x 1.36	4.74 (s) x 1.37	69.6 (s) x 20.1	3255 (s) x 942
580975	9.77 (s) x 1	13.2 (s) x 1.35	14.1 (s) x 1.44	204 (s) x 20.9	-
1747861	31.2 (s) x 1	40.4 (s) x 1.29	44.5 (s) x 1.42	623 (s) x 20.0	-

StiffAssembling3DP1 - Octave

n_{dof}	OptVS	OptV	OptV2	OptV1	base
14708	.143 (s) x 1	.177 (s) x 1.24	.146 (s) x 1.02	76.0 (s) x 531	77.3 (s) x 540
40570	.293 (s) x 1	.446 (s) x 1.52	.488 (s) x 1.66	216 (s) x 737	248 (s) x 844
200424	1.91 (s) x 1	2.37 (s) x 1.24	3.15 (s) x 1.65	1120 (s) x 586	3041 (s) x 1592
580975	7.19 (s) x 1	9.15 (s) x 1.27	10.6 (s) x 1.47	3264 (s) x 454	-
1747861	31.0 (s) x 1	38.0 (s) x 1.22	40.5 (s) x 1.31	-	-

StiffAssembling3DP1 - Python

n_{dof}	OptVS	OptV	OptV2	OptV1	base
14708	.191 (s) x 1	.207 (s) x 1.08	.258 (s) x 1.35	12.8 (s) x 67.0	172 (s) x 903
40570	.377 (s) x 1	.530 (s) x 1.41	.823 (s) x 2.19	36.0 (s) x 95.5	488 (s) x 1295
200424	1.93 (s) x 1	2.50 (s) x 1.29	4.15 (s) x 2.14	182 (s) x 94.0	2480 (s) x 1282
580975	6.80 (s) x 1	8.89 (s) x 1.31	12.4 (s) x 1.83	541 (s) x 79.5	-
1747861	26.1 (s) x 1	34.2 (s) x 1.31	40.3 (s) x 1.55	-	-

Table A.3: Stiffness matrix (3D) : comparison of `OptVS`, `OptV`, `OptV2`, `OptV1` and `base` codes in Matlab (top left), Octave (top right) and Python (bottom) giving time in seconds (top value) and `OptVS` speedup (bottom value).

StiffElasAssembling2DP1 - Matlab

n_{dof}	OptVS	OptV	OptV2	OptV1	base
28444	.114 (s) x 1	.272 (s) x 2.39	.185 (s) x 1.63	16.9 (s) x 148	67.8 (s) x 594
250020	1.01 (s) x 1	1.68 (s) x 1.65	1.91 (s) x 1.88	150 (s) x 148	6156 (s) x 6073
686164	3.20 (s) x 1	5.28 (s) x 1.65	5.42 (s) x 1.70	414 (s) x 129	-
1771042	8.97 (s) x 1	14.7 (s) x 1.64	15.2 (s) x 1.69	1090 (s) x 122	-
3957204	21.6 (s) x 1	34.1 (s) x 1.58	33.1 (s) x 1.53	-	-

StiffElasAssembling2DP1 - Octave

n_{dof}	OptVS	OptV	OptV2	OptV1	base
28444	.082 (s) x 1	.129 (s) x 1.57	.091 (s) x 1.11	63.7 (s) x 777	88.6 (s) x 1080
250020	.726 (s) x 1	1.26 (s) x 1.74	.915 (s) x 1.26	564 (s) x 777	4485 (s) x 6176
686164	2.21 (s) x 1	3.57 (s) x 1.61	3.36 (s) x 1.52	1550 (s) x 701	-
1771042	6.85 (s) x 1	10.7 (s) x 1.56	9.39 (s) x 1.37	-	-
3957204	16.2 (s) x 1	25.8 (s) x 1.59	22.0 (s) x 1.36	-	-

StiffElasAssembling2DP1 - Python

n_{dof}	OptVS	OptV	OptV2	OptV1	base
28444	.136 (s) x 1	.207 (s) x 1.53	.202 (s) x 1.49	30.0 (s) x 221	183 (s) x 1348
250020	.721 (s) x 1	1.08 (s) x 1.50	1.22 (s) x 1.7	277 (s) x 384	1639 (s) x 2274
686164	2.05 (s) x 1	3.06 (s) x 1.50	3.55 (s) x 1.73	761 (s) x 372	-
1771042	6.06 (s) x 1	9.12 (s) x 1.50	9.58 (s) x 1.58	-	-
3957204	14.0 (s) x 1	21.2 (s) x 1.52	21.5 (s) x 1.54	-	-

Table A.4: Elastic stiffness matrix (2D): comparison of `OptVS`, `OptV`, `OptV2` , `OptV1` and `base` codes in Matlab (top left), Octave (top right) and Python (bottom) giving time in seconds (top value) and `OptVS` speedup (bottom value).

B Proof of Lemma 5.1

To prove Lemma 5.1, we introduce the following matrix $\mathbb{B}_l$:

$$\mathbb{B}_l = \begin{pmatrix} \delta_{l,1} & 0 \\ 0 & \delta_{l,2} \\ \delta_{l,2} & \delta_{l,1} \end{pmatrix} \text{ if } d=2, \text{ and } \mathbb{B}_l = \begin{pmatrix} \delta_{l,1} & 0 & 0 \\ 0 & \delta_{l,2} & 0 \\ 0 & 0 & \delta_{l,3} \\ \delta_{l,2} & \delta_{l,1} & 0 \\ 0 & \delta_{l,3} & \delta_{l,2} \\ \delta_{l,3} & 0 & \delta_{l,1} \end{pmatrix} \text{ if } d=3.$$

Thus we have $\underline{\boldsymbol{\epsilon}}(\boldsymbol{\lambda}_{l,\alpha}) = \mathbb{B}_l \nabla \lambda_\alpha$ and then

$$\underline{\boldsymbol{\epsilon}}^t(\boldsymbol{\lambda}_{n,\beta})\mathbb{C}\underline{\boldsymbol{\epsilon}}(\boldsymbol{\lambda}_{l,\alpha}) = \nabla \lambda_\beta^t \mathbb{B}_n^t \mathbb{C} \mathbb{B}_l \nabla \lambda_\alpha.$$

Moreover we have $\mathbb{C} = \lambda \mathbb{C}_0 + \mu \mathbb{C}_1$ with

$$\mathbb{C}_0 = \begin{pmatrix} \mathbb{1}_d & \mathbb{O}_{d,2d-3} \\ \mathbb{O}_{2d-3,d} & \mathbb{O}_{2d-3} \end{pmatrix}_{3(d-1)\times 3(d-1)} \text{ and } \mathbb{C}_1 = \begin{pmatrix} 2\mathbb{I}_d & \mathbb{O}_{d,2d-3} \\ \mathbb{O}_{2d-3,d} & \mathbb{I}_{2d-3} \end{pmatrix}_{3(d-1)\times 3(d-1)}$$

Thus we obtain

$$\underline{\boldsymbol{\epsilon}}^t(\boldsymbol{\lambda}_{n,\beta})\mathbb{C}\underline{\boldsymbol{\epsilon}}(\boldsymbol{\lambda}_{l,\alpha}) = \lambda \nabla \lambda_\beta^t \mathbb{B}_n^t \mathbb{C}_0 \mathbb{B}_l \nabla \lambda_\alpha + \mu \nabla \lambda_\beta^t \mathbb{B}_n^t \mathbb{C}_1 \mathbb{B}_l \nabla \lambda_\alpha.$$

Denoting $\mathbb{Q}^{n,l} = \mathbb{B}_n^t \mathbb{C}_0 \mathbb{B}_l$ and $\mathbb{S}^{n,l} = \mathbb{B}_n^t \mathbb{C}_1 \mathbb{B}_l$ we obtain (5.6) which ends the proof of Lemma 5.1.

C Remaining routines

C.1 Gradients of the barycentric coordinates

Let T_k be a d-simplex of $\mathbb{R}^d$ with vertices $\mathrm{q}^0, \ldots, \mathrm{q}^d$, and $\hat{T}$ be the reference d-simplex with vertices $\hat{\mathrm{q}}^0, \ldots, \hat{\mathrm{q}}^d$ where $\hat{\mathrm{q}}^0 = \mathbf{0}_d$ and $\hat{\mathrm{q}}^i = \boldsymbol{e}_i$, $\forall i \in \{1, \ldots, d\}$.

Let $\mathcal{F}_k$ be the bijection from $\hat{T}$ to T_k defined by $\mathrm{q} = \mathcal{F}_k(\hat{\mathrm{q}}) = \mathbb{B}_k \hat{\mathrm{q}} + \mathrm{q}^0$ where $\mathbb{B}_k \in \mathcal{M}_d(\mathbb{R})$ is such that its i-th column is equal to $\mathrm{q}^i - \mathrm{q}^0$, for all $i \in \{1, \ldots, d\}$.

The barycentric coordinates of $\hat{\mathrm{q}} = (\hat{x}_1, \ldots, \hat{x}_d) \in \hat{T}$ are given by $\hat{\lambda}_0 = 1 - \sum_{i=1}^d \hat{x}_i$, and $\hat{\lambda}_i = \hat{x}_i$, $\forall i \in \{1, \ldots, d\}$. The barycentric coordinates of $\mathrm{q} = (x_1, \ldots, x_d) \in T_k$ are given by $\lambda_{k,i}(\mathrm{q}) = \hat{\lambda}_i \circ \mathcal{F}_k^{-1}(\mathrm{q})$ and we have

$$\nabla \lambda_{k,i}(\mathrm{q}) = \mathbb{B}_k^{-t} \hat{\nabla} \hat{\lambda}_i(\hat{\mathrm{q}}), \ \forall i \in \{0, \ldots, d\}, \tag{C.1}$$

with $\hat{\nabla}\hat{\lambda}_0(\hat{\mathrm{q}}) = \begin{pmatrix} -1 \\ \cdots \\ -1 \end{pmatrix}$, $\hat{\nabla}\hat{\lambda}_i = \boldsymbol{e}_i$, $\forall i \in \{1, \ldots, d\}$. Note that gradients are constant. Let

$$\hat{\mathbb{G}} = \left(\hat{\nabla}\hat{\lambda}_0, \ \ldots, \ \hat{\nabla}\hat{\lambda}_d\right) = \begin{pmatrix} -1 & 1 & 0 & \ldots & 0 \\ -1 & 0 & 1 & \ddots & \vdots \\ \vdots & \vdots & \ddots & \ddots & 0 \\ -1 & 0 & \ldots & 0 & 1 \end{pmatrix}.$$

Then computing the gradients of the barycentric coordinates is equivalent to solve $(d+1)$ linear systems, written in matrix form as follows:

$$\mathbb{B}_k^t \mathbb{G}_k = \hat{\mathbb{G}}, \tag{C.2}$$

where $\mathbb{G}_k = \big(\nabla\lambda_{k,0}(\mathrm{q}),\ \ldots,\ \nabla\lambda_{k,d}(\mathrm{q})\big) \in \mathcal{M}_{d,d+1}(\mathbb{R})$.

For each d-simplex one has to calculate $(d+1)$ gradients and thus to determine $(d+1)\mathrm{n_{me}}$ vectors of dimension d.

The vectorization of the calculation of the gradients is done by rewriting the equations (C.2), for $k = 1, ..., \mathrm{n_{me}}$, under an equivalent form of a large block diagonal sparse system of size $N = d \times \mathrm{n_{me}}$, with d-by-d diagonal blocks given by:

$$\begin{pmatrix} \mathbb{B}_1^t & \mathbb{O} & \ldots & \mathbb{O} \\ \mathbb{O} & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \mathbb{O} \\ \mathbb{O} & \ldots & \mathbb{O} & \mathbb{B}_{\mathrm{n_{me}}}^t \end{pmatrix}_{N\times N} \begin{pmatrix} \mathbb{G}_1 \\ \mathbb{G}_2 \\ \vdots \\ \mathbb{G}_{\mathrm{n_{me}}} \end{pmatrix}_{N\times(d+1)} = \begin{pmatrix} \hat{\mathbb{G}} \\ \hat{\mathbb{G}} \\ \vdots \\ \hat{\mathbb{G}} \end{pmatrix}_{N\times(d+1)} \tag{C.3}$$

Algorithm C.1 Vectorized computation of gradients of the basis functions in dimension d

Function $\boldsymbol{G} \leftarrow$ GradientVec(q, me)
  $\mathbb{K} \leftarrow \mathbb{I} \leftarrow \mathbb{J} \leftarrow$ zeros$(d, d, \mathrm{n_{me}})$
  ii $\leftarrow d * [0 : (\mathrm{n_{me}} - 1)]$
  for $i \leftarrow 1$ **to** d **do**
    for $j \leftarrow 1$ **to** d **do**
      $\mathbb{K}(i, j, :) \leftarrow \mathrm{q}(i, \mathrm{me}(j + 1, :)) - \mathrm{q}(i, \mathrm{me}(1, :))$
      $\mathbb{I}(i, j, :) \leftarrow$ ii $+ j$, $\mathbb{J}(i, j, :) \leftarrow$ ii $+ i$
    end for
  end for
  $\mathbb{S} \leftarrow$ sparse$(\mathbb{I}(:), \mathbb{J}(:), \mathbb{K}(:), d * \mathrm{n_{me}}, d * \mathrm{n_{me}})$
  $\mathbb{R} \leftarrow$ zeros$(d * \mathrm{n_{me}}, d + 1)$ ▷ Build RHS
  $\hat{\mathbb{G}} \leftarrow [-\mathbb{1}_{d\times 1}, \mathbb{I}_d]$
  $\mathbb{R} \leftarrow$ copymat$(\hat{\mathbb{G}}, \mathrm{n_{me}}, 1)$
  $\mathbb{G} \leftarrow$ solve$(\mathbb{S}, \mathbb{R})$ ▷ $\mathbb{G}(d(k-1)+i, \alpha) = \frac{\partial\lambda_\alpha}{\partial x_i}_{|T_k}$
  $\mathbb{G} \leftarrow$ transform$(\mathbb{G}, ...)$ ▷ such that $\mathbb{G}(k, \alpha, i) = \frac{\partial\lambda_\alpha}{\partial x_i}_{|T_k}$
end Function

The performance of this algorithm may be improved by writing specific algorithms in each dimension $d = 1, 2$ or 3 (see Appendix A in [6]).

C.2 Elastic stiffness matrix assembly : algorithm using the symmetry

When the assembly matrix is symmetric, one may improve the performance of Algorithm 5.3 by using the symmetry of the element matrices (see Section 4), which leads to the following algorithm:

Algorithm C.2 (`OptVS`) - Optimized assembly in vector case ($m > 1$)

Function $\mathbb{M} \leftarrow$AssemblyVecGenP1OptVS(me, $\mathrm{n_q}, \ldots$)
  $\mathrm{n_{dof}} \leftarrow m * \mathrm{n_q}$
  $\mathbb{M} \leftarrow \mathbb{O}_{\mathrm{n_{dof}}}$ ▷ $\mathrm{n_{dof}}$-by-$\mathrm{n_{dof}}$ sparse matrix
  for $l \leftarrow 1$ **to** m **do**
    for $\alpha \leftarrow 1$ **to** $d+1$ **do**
      $\boldsymbol{I}_g \leftarrow m * (\mathrm{me}(\alpha, :) - 1) + l$
      $ii \leftarrow m(\alpha - 1) + l$
      for $n \leftarrow 1$ **to** m **do**
        for $\beta \leftarrow 1$ **to** $d+1$ **do**
          $jj \leftarrow m(\beta - 1) + n$
          if $ii > jj$ **then**
            $\boldsymbol{K}_g \leftarrow$ vecHe$(l, \alpha, n, \beta, \ldots)$
            $\boldsymbol{J}_g \leftarrow m * (\mathrm{me}(\beta, :) - 1) + n$
            $\mathbb{M} \leftarrow \mathbb{M} +$ Sparse$(\boldsymbol{I}_g, \boldsymbol{J}_g, \boldsymbol{K}_g, \mathrm{n_{dof}}, \mathrm{n_{dof}})$
          end if
        end for
      end for
    end for
  end for
  $\mathbb{M} \leftarrow \mathbb{M} + \mathbb{M}'$
  for $l \leftarrow 1$ **to** m **do**
    for $\alpha \leftarrow 1$ **to** $d+1$ **do**
      $\boldsymbol{I}_g \leftarrow m * (\mathrm{me}(\alpha, :) - 1) + l$
      $\boldsymbol{K}_g \leftarrow$ vecHe$(l, \alpha, l, \alpha, \ldots)$
      $\mathbb{M} \leftarrow \mathbb{M} +$ Sparse$(\boldsymbol{I}_g, \boldsymbol{I}_g, \boldsymbol{K}_g, \mathrm{n_{dof}}, \mathrm{n_{dof}})$
    end for
  end for
end Function

D Extension to $\mathbb{P}_k$-Lagrange finite elements

In this section we adapt the optimized algorithm of Section 4 to the case of finite elements of higher order. For simplicity, we consider the assembly algorithm on the example of the mass matrix.

The mesh used is adapted to $\mathbb{P}_k$ finite elements and is called a"$\mathbb{P}_k$-mesh". Only arrays q and me differ between the usual mesh and the $\mathbb{P}_k$-mesh. In the $\mathbb{P}_k$-mesh, q contains the coordinates of the nodal points associated to the $\mathbb{P}_k$ finite elements and me is of dimension $\mathrm{n_{dfe}}$-by-$\mathrm{n_{me}}$, where $\mathrm{n_{dfe}}$ is the local number of $\mathbb{P}_k$-nodes in a d-simplex K : $\mathrm{n_{dfe}} = \frac{(d+k)!}{d!k!}$, as shown in the table below.

name	type	dimension	description
$\mathrm{n_{dfe}}$	integer	1	local number of $\mathbb{P}_k$-nodes in a d-simplex
$\mathrm{n_q}$	integer	1	number of $\mathbb{P}_k$-nodes
q	double	$\mathrm{d} \times \mathrm{n_q}$	array of $\mathbb{P}_k$-node coordinates
me	integer	$\mathrm{n_{dfe}} \times \mathrm{n_{me}}$	($\mathbb{P}_k$) connectivity array

By construction, the total number of degrees of freedom of a $\mathbb{P}_k$-mesh is its number of nodal points. One may use for example `gmsh` [16] to generate a $\mathbb{P}_k$-mesh in 2D or in 3D.

First, we need to introduce some notations: let $\mathcal{S}_d^k$ be the set of multi-indices given by

$$\mathcal{S}_d^k = \left\{ \boldsymbol{\alpha} = (\alpha_1, \ldots, \alpha_{d+1}) \in \mathbb{N}^{d+1} \text{ such that } |\boldsymbol{\alpha}| := \sum_{i=1}^{d+1} \alpha_i = k \right\}, \tag{D.1}$$

with $\#\mathcal{S}_d = N$. Then the $\mathbb{P}_k$ basis functions $\varphi_{\boldsymbol{\alpha}}$ on a d-simplex K may be deduced from the barycentric coordinates $\{\lambda_j\}_{j=1}^{d+1}$

$$\varphi_{\boldsymbol{\alpha}} = \prod_{l=1}^{d+1} \prod_{j=0}^{\alpha_l - 1} \frac{k\lambda_l - j}{j+1}, \quad \forall \boldsymbol{\alpha} \in \mathcal{S}_d^k, \tag{D.2}$$

or equivalently, noticing that $\varphi_{\boldsymbol{\alpha}}$ is a polynomial in the variable $(\lambda_1, \ldots, \lambda_{d+1})$ and introducing a multi-index $\boldsymbol{\mu} = (\mu_1, \ldots, \mu_{d+1}) \in \mathbb{N}^{d+1}$, we have

$$\varphi_{\boldsymbol{\alpha}} = \sum_{|\boldsymbol{\mu}| \leq k} a_{\boldsymbol{\mu}}(\boldsymbol{\alpha}) \left(\prod_{j=1}^{d+1} \lambda_j^{\mu_j} \right). \tag{D.3}$$

All the non-zero $a_{\boldsymbol{\mu}}(\boldsymbol{\alpha})$ values can be computed from (D.2) and depend only on $\boldsymbol{\alpha}$, d and k.

As in the previous sections, the assembly algorithm of the mass matrix is based on the vectorization of the local mass matrix $\mathbb{M}^e$ on K, which is an N-by-N matrix given by

$$\mathbb{M}^e_{I(\boldsymbol{\alpha}), I(\boldsymbol{\beta})}(K) = \int_K \varphi_{\boldsymbol{\alpha}} \varphi_{\boldsymbol{\beta}} d\mathrm{q}, \quad \forall (\boldsymbol{\alpha}, \boldsymbol{\beta}) \in \mathcal{S}_d^k \times \mathcal{S}_d^k,$$

where $I : \mathcal{S}_d^k \longrightarrow \{1, \ldots, N\}$ is the local numbering choice.

We then introduce a formula of the same type as (4.3) to vectorize the computation of $\mathbb{M}^e$. Using (D.3), we have for all $(\boldsymbol{\alpha}, \boldsymbol{\beta}) \in \mathcal{S}_d^k \times \mathcal{S}_d^k$

$$\int_K \varphi_{\boldsymbol{\alpha}} \varphi_{\boldsymbol{\beta}} d\mathrm{q} = \sum_{|\boldsymbol{\mu}| \leq k} \sum_{|\boldsymbol{\nu}| \leq k} a_{\boldsymbol{\mu}}(\boldsymbol{\alpha}) a_{\boldsymbol{\nu}}(\boldsymbol{\beta}) \int_K \prod_{j=1}^{d+1} \lambda_j^{\mu_j + \nu_j} d\mathrm{q}.$$

Then, using formula (2.4) we obtain

$$\int_K \varphi_{\boldsymbol{\alpha}} \varphi_{\boldsymbol{\beta}} d\mathrm{q} = d! |K| C_{\boldsymbol{\alpha}, \boldsymbol{\beta}}, \tag{D.4}$$

where the constant $C_{\boldsymbol{\alpha}, \boldsymbol{\beta}}$ does not depend on K and is given by

$$C_{\boldsymbol{\alpha}, \boldsymbol{\beta}} = \sum_{|\boldsymbol{\mu}| \leq k} \sum_{|\boldsymbol{\nu}| \leq k} a_{\boldsymbol{\mu}}(\boldsymbol{\alpha}) a_{\boldsymbol{\nu}}(\boldsymbol{\beta}) \frac{\prod_{i=1}^{d+1} (\mu_i + \nu_i)!}{(d + |\boldsymbol{\mu}| + |\boldsymbol{\nu}|)!}. \tag{D.5}$$

Using (D.4), we can now extend Algorithm 4.4 (with $w = 1$) to the $\mathbb{P}_k$ finite element case. This leads to the vectorized algorithm of the mass matrix given in Algorithm D.1.

Remark D.1 We have considered the extension of the `OptV2` algorithm to finite elements of higher order. The main idea is that all the steps of Section 4 remain valid for $\mathbb{P}_k$ finite elements, if one replaces $(d+1)$ by $\mathrm{n_{dfe}}$, and with q and me defined above. Then, one may derive from Algorithm D.1 the other optimized versions `OptV` and `OptVS` for the $\mathbb{P}_k$ case, as in Section 4.

Algorithm D.1 (`OptV2`) - Mass matrix in $\mathbb{P}_k$ case

Function $\mathbb{M} \leftarrow$ AssemblyMassPk(me, vols, n_q, d, k)
  $\mathbb{C} \leftarrow$ coeffMass(d, k) ▷ Get coefficients $C_{\boldsymbol{\alpha},\boldsymbol{\beta}}$
  $\mathbb{K}_g \leftarrow \mathbb{I}_g \leftarrow \mathbb{J}_g \leftarrow$ zeros(n_{dfe}^2, n_{me}) ▷ n_{dfe}^2-by-n_{me} 2d-arrays
  $l \leftarrow 1$
  for $\beta \leftarrow 1$ **to** n_{dfe} **do**
    for $\alpha \leftarrow 1$ **to** n_{dfe} **do**
      $\mathbb{K}_g(l,:) \leftarrow d! * \mathbb{C}(\alpha,\beta) * \text{vols}$
      $\mathbb{I}_g(l,:) \leftarrow \text{me}(\alpha,:)$
      $\mathbb{J}_g(l,:) \leftarrow \text{me}(\beta,:)$
      $l \leftarrow l + 1$
    end for
  end for
  $\mathbb{M} \leftarrow$ Sparse($\mathbb{I}_g, \mathbb{J}_g, \mathbb{K}_g, n_q, n_q$)
end Function

In Table D.1, using Matlab, we show the computation times versus the number of $\mathbb{P}_k$ nodes, for Algorithm 4.4 (with $w = 1$), and for Algorithm D.1 with $k = 1, 2, 3, 4, 5, 6$. We observe that the computation times are almost the same for Algorithm 4.4 and Algorithm D.1 with $k = 1$. Moreover, for a fixed number of nodes, the computation times increase slowly with the degree of the polynomials: for a million of nodes, the computation time with $\mathbb{P}_5$ finite elements is twice the one for $\mathbb{P}_1$ finite elements.

n_{dof}	P1OptV2	Pk(k=1)	Pk(k=2)	Pk(k=3)	Pk(k=4)	Pk(k=5)	Pk(k=6)
3.10^4	0.535	0.543	0.459	0.544	0.707	0.982	1.350
1.210^5	2.322	2.500	2.025	2.407	3.184	4.389	5.696
5.10^5	10.885	13.203	9.811	11.684	15.184	19.766	25.340
10^6	22.744	28.362	22.635	25.656	33.314	42.812	54.782

Table D.1: 3D Mass matrix : computational cost versus n_{dof}, with Matlab, for `OptV2` code : Algorithm 4.4 with $w = 1$) (column 1), and with Algorithm D.1 for $k = 1, 2, 3, 4, 5, 6$ (columns 2 to 7).

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. I. Anjam and J. Valdman. Fast MATLAB assembly of FEM matrices in 2D and 3D: Edge elements. arXiv:1409.4618v1, 2014.
2. L. Chen. Programming of Finite Element Methods in Matlab. `http://math.uci.edu/~chenlong/226/Ch3FEMCode.pdf`, 2011.
3. L. Chen. `iFEM`, a Matlab software package. `http://math.uci.edu/~chenlong/programming.html`, 2013.
4. Z. Chen. Finite Element Methods and their Applications. Springer, scientific computation edition, 2005.
5. P. G. Ciarlet. The Finite Element Method for Elliptic Problems. SIAM, Philadelphia, 2002.

6. F. Cuvelier, C. Japhet, and G. Scarella. An efficient way to perform the assembly of finite element matrices in vector languages. http://hal.archives-ouvertes.fr/hal-00931066, 2014.
7. F. Cuvelier, C. Japhet, and G. Scarella. OptFEM packages. http://www.math.univ-paris13.fr/~cuvelier/software, 2015.
8. M. Dabrowski, M. Krotkiewski, and D. W. Schmid. Milamin: Matlab-based finite element method solver for large problems. Geochem. Geophys. Geosyst., 9, 2008.
9. T. A. Davis. Direct Methods for Sparse Linear Systems. Society for Industrial and Applied Mathematics, 2006.
10. T. A. Davis. SuiteSparse packages, released 4.2.1. http://faculty.cse.tamu.edu/davis/suitesparse.html, 2013.
11. G. Dhatt, E. Lefrançois, and G. Touzot. Finite Element Method. Wiley, 2012.
12. Scilab Enterprises. Scilab. http://www.scilab.org/, 2015.
13. Python Software Foundation. Python. http://www.python.org/, 2013.
14. R Foundation. The R Project for Statistical Computing. http://www.r-project.org/, 2015.
15. 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.
16. C. Geuzaine and J.-F. Remacle. Gmsh: a three-dimensional finite element mesh generator with built-in pre- and post-processing facilities. International Journal for Numerical Methods in Engineering, 79(11):1309–1331, 2009.
17. A. Hannukainen and M. Juntunen. Implementing the Finite Element Assembly in Interpreted Languages, 2012. Preprint, Aalto University.
18. F. Hecht. New development in freefem++. J. Numer. Math., 20 (3-4):251–265, 2012.
19. F. Hecht, O. Pironneau, J. Morice, A. Le Hyaric, and K. Ohtsuka. Freefem++. http://www.freefem.org/ff++.
20. J. S. Hesthaven and T. Warburton. Nodal Discontinuous Galerkin Methods, Algorithms, Analysis, and Applications, volume 54. Springer, texts in Applied Mathematics edition, 2008.
21. C. Johnson. Numerical Solution of Partial Differential Equations by the Finite Element Method. Dover Publications, Inc, 2009.
22. Julia. http://julialang.org/, 2015.
23. J. Koko. Vectorized Matlab codes for linear two-dimensional elasticity. Scientific Programming, 15(3):157–172, 2007.
24. 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 Modeling, Simulation and Optimization of Complex Processes, pages 337–358. Springer, 2008.
25. A. Logg, K.-A. Mardal, and G. N. Wells et al. Matlab Implementation of the Finite Element Method in Elasticity. Automated Solution of Differential Equations by the Finite Element Method, 2012.
26. B. Lucquin and O. Pironneau. Introduction to Scientific Computing. John Wiley & Sons Ltd, 1998.
27. Mathworks. Matlab. http://www.mathworks.com, 2014.
28. NVIDIA. Cusp, a C++ Templated Library for sparse linear algebra on CUDA. https://developer.nvidia.com/cusp, 2013.
29. NVIDIA. Thrust, a C++ template library for CUDA based on the Standard Template Library (STL). https://developer.nvidia.com/thrust, 2013.
30. Octave community. GNU Octave 3.8.1, 2014.
31. A. Quarteroni. Numerical Models for Differential Problems. Springer, 2014.
32. A. Quarteroni, F. Saleri, and P. Gervasio. Scientific Computing with MATLAB and Octave. Springer, texts in Computational Science and Engineering edition, 2013.
33. A. Quarteroni and A. Valli. Numerical Approximation of Partial Differential Equations. Springer, 2008.
34. T. Rahman and J. Valdman. Fast MATLAB assembly of FEM matrices in 2D and 3D: Nodal elements. Appl. Math. Comput., 219 (13):7151–7158, 2013.
35. V. Thomée. Galerkin Finite Element Method for Parabolic Problems. Springer, 1994.