An efficient way to perform the assembly of finite element matrices in Matlab and Octave

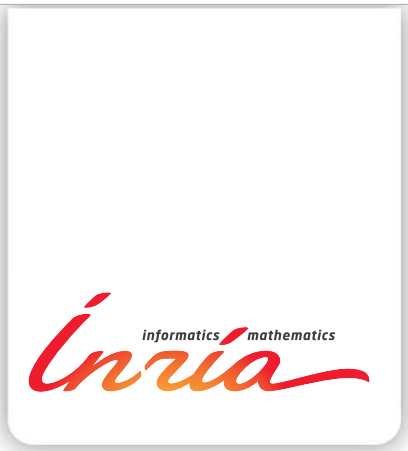
François Cuvelier, Caroline Japhet, Gilles Scarella

An efficient way to perform the assembly of finite element matrices in Matlab and Octave

François Cuvelier, Caroline Japhet , Gilles Scarella

RESEARCH REPORT
N° 8305
May 2013
Project-Teams Pomdapi

ISSN 0249-6399 ISRN INRIA/RR--8305--FR+ENG



An efficient way to perform the assembly of finite element matrices in Matlab and Octave

François Cuvelier *, Caroline Japhet * §, Gilles Scarella*

Project-Teams Pomdapi

Research Report n° 8305 — May 2013 — 37 pages

Abstract: We describe different optimization techniques to perform the assembly of finite element matrices in Matlab and Octave, from the standard approach to recent vectorized ones, without any low level language used. We finally obtain a simple and efficient vectorized algorithm able to compete in performance with dedicated software such as FreeFEM++. The principle of this assembly algorithm is general, we present it for different matrices in the P_1 finite elements case and in linear elasticity. We present numerical results which illustrate the computational costs of the different approaches.

Key-words: P_1 finite elements, matrix assembly, vectorization, linear elasticity, Matlab, Octave

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

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

§ INRIA Paris-Rocquencourt, project-team Pomdapi, 78153 Le Chesnay Cedex, France Email:Caroline.Japhet@inria.fr

RESEARCH CENTRE
PARIS – ROCQUENCOURT

Domaine de Voluceau, - Rocquencourt
B.P. 105 - 78153 Le Chesnay Cedex

Optimisation de l'assemblage de matrices éléments finis sous Matlab et Octave

Résumé : L'objectif est de décrire différentes techniques d'optimisation, sous Matlab/Octave, de routines d'assemblage de matrices éléments finis, en partant de l'approche classique jusqu'aux plus récentes vectorisées, sans utiliser de langage de bas niveau. On aboutit au final à une version vectorisée rivalisant, en terme de performance, avec des logiciels dédiés tels que FreeFEM++. Les descriptions des différentes méthodes d'assemblage étant génériques, on les présente pour différentes matrices dans le cadre des éléments finis P_1-Lagrange en dimension 2 et en élasticité linéaire. Des résultats numériques sont donnés pour illustrer les temps calculs des méthodes proposées.

Mots-clés : éléments finis P_1, assemblage de matrices, vectorisation, élasticité linéaire, Matlab, Octave

1. Introduction. Usually, finite elements methods [4, 14] are used to solve partial differential equations (PDEs) occurring in many applications such as mechanics, fluid dynamics and computational electromagnetics. These methods are based on a discretization of a weak formulation of the PDEs and need the assembly of large sparse matrices (e.g. mass or stiffness matrices). They enable complex geometries and various boundary conditions and they may be coupled with other discretizations, using a weak coupling between different subdomains with nonconforming meshes [1]. Solving accurately these problems requires meshes containing a large number of elements and thus the assembly of large sparse matrices.

Matlab [16] and GNU Octave [11] are efficient numerical computing softwares using matrix-based language for teaching or industry calculations. However, the classical assembly algorithms (see for example [5, 15]) basically implemented in Matlab/Octave are much less efficient than when implemented with other languages.

In [8] Section 10, T. Davis describes different assembly techniques applied to random matrices of finite element type, while the classical matrices are not treated. A first vectorization technique is proposed in [8]. Other more efficient algorithms have been proposed recently in [2, 3, 12, 17]. More precisely, in [12], a vectorization is proposed, based on the permutation of two local loops with the one through the elements. This more formal technique allows to easily assemble different matrices, from a reference element by affine transformation and by using a numerical integration. In [17], 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 [3], L. Chen builds vectorially the nine sparse matrices corresponding to the nine elements of the element matrix and adds them to obtain the global matrix.

In this paper we present an optimization approach, in Matlab/Octave, using a vectorization of the algorithm. This finite element assembly code is entirely vectorized (without loop) and without any quadrature formula. Our vectorization is close to the one proposed in [2], with a full vectorization of the arrays of indices.

Due to the length of the paper, we restrict ourselves to P_1 Lagrange finite elements in 2D with an extension to linear elasticity. Our method extends easily to the P_k finite elements case, $k \geqslant 2$, and in 3D, see [7]. We compare the performances of this code with the ones obtained with the standard algorithm and with those proposed in [2, 3, 12, 17]. We also show that this implementation is able to compete in performance with dedicated software such as FreeFEM++ [13]. All the computations are done on our reference computer[1] with the releases R2012b for Matlab, 3.6.3 for Octave and 3.20 for FreeFEM++. The entire Matlab/Octave code may be found in [6]. The Matlab codes are fully compatible with Octave.

The remainder of this paper is organized as follows: in Section 2 we give the notations associated to the mesh and we define three finite element matrices. Then, in Section 3 we recall the classical algorithm to perform the assembly of these matrices and show its inefficiency compared to FreeFEM++. This is due to the storage of sparse

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

§INRIA Paris-Rocquencourt, project-team Pomdapi, 78153 Le Chesnay Cedex, France

[1] 2 x Intel Xeon E5645 (6 cores) at 2.40Ghz, 32Go RAM, supported by GNR MoMaS

matrices in Matlab/Octave as explained in Section 4. In Section 5 we give a method to best use Matlab/Octave `sparse` function, the "optimized version 1", suggested in [8]. Then, in Section 6 we present a new vectorization approach, the "optimized version 2", and compare its performances to those obtained with FreeFEM++ and the codes given in [2, 3, 12, 17]. Finally, in Section 7, we present an extension to linear elasticity. The full listings of the routines used in the paper are given in Appendix B (see also [6]).

2. Notations. Let Ω be an open bounded subset of $\mathbb{R}^2$. It is provided with its mesh $\mathcal{T}_h$ (classical and locally conforming). We use a triangulation $\Omega_h = \bigcup_{T_k \in \mathcal{T}_h} T_k$ of Ω (see Figure 2.1) described by :

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

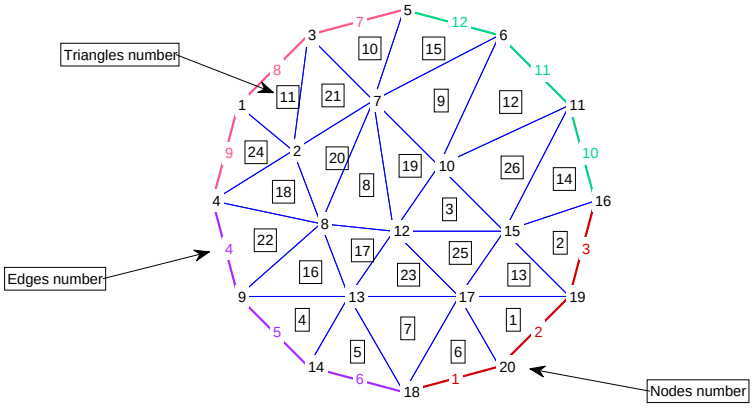


FIG. 2.1. *Description of the mesh.*

In this paper we will consider the assembly of the mass, weighted mass and stiffness matrices denoted by $\mathbb{M}$, $\mathbb{M}^{[w]}$ and $\mathbb{S}$ respectively. These matrices of size $\mathrm{n_q}$ are sparse, and their coefficients are defined by

$$\mathbb{M}_{i,j} = \int_{\Omega_h} \varphi_i(\mathrm{q})\varphi_j(\mathrm{q})d\mathrm{q}, \quad \mathbb{M}^{[w]}_{i,j} = \int_{\Omega_h} w(\mathrm{q})\varphi_i(\mathrm{q})\varphi_j(\mathrm{q})d\mathrm{q}, \quad \mathbb{S}_{i,j} = \int_{\Omega_h} \langle \nabla \varphi_i(\mathrm{q}), \nabla \varphi_j(\mathrm{q}) \rangle d\mathrm{q},$$

where φ_i are the usual P_1 Lagrange basis functions, w is a function defined on Ω and $\langle \cdot, \cdot \rangle$ is the usual scalar product in $\mathbb{R}^2$. More details are given in [5]. To assemble

these matrices, one needs to compute its associated element matrix. On a triangle T with local vertices $\tilde{q}^1$, $\tilde{q}^2$, $\tilde{q}^3$ and area $|T|$, the element mass matrix is given by

$$\mathbb{M}^e(T) = \frac{|T|}{12}\begin{pmatrix} 2 & 1 & 1 \\ 1 & 2 & 1 \\ 1 & 1 & 2 \end{pmatrix}. \tag{2.1}$$

Let $\tilde{w}_\alpha = w(\tilde{q}^\alpha)$, $\forall \alpha \in \{1, \ldots, 3\}$. The element weighted mass matrix is approximated by

$$\mathbb{M}^{e,[\tilde{w}]}(T) = \frac{|T|}{30}\begin{pmatrix} 3\tilde{w}_1 + \tilde{w}_2 + \tilde{w}_3 & \tilde{w}_1 + \tilde{w}_2 + \frac{\tilde{w}_3}{2} & \tilde{w}_1 + \frac{\tilde{w}_2}{2} + \tilde{w}_3 \\ \tilde{w}_1 + \tilde{w}_2 + \frac{\tilde{w}_3}{2} & \tilde{w}_1 + 3\tilde{w}_2 + \tilde{w}_3 & \frac{\tilde{w}_1}{2} + \tilde{w}_2 + \tilde{w}_3 \\ \tilde{w}_1 + \frac{\tilde{w}_2}{2} + \tilde{w}_3 & \frac{\tilde{w}_1}{2} + \tilde{w}_2 + \tilde{w}_3 & \tilde{w}_1 + \tilde{w}_2 + 3\tilde{w}_3 \end{pmatrix}. \tag{2.2}$$

Denoting $\boldsymbol{u} = \tilde{q}^2 - \tilde{q}^3$, $\boldsymbol{v} = \tilde{q}^3 - \tilde{q}^1$ and $\boldsymbol{w} = \tilde{q}^1 - \tilde{q}^2$, the element stiffness matrix is

$$\mathbb{S}^e(T) = \frac{1}{4|T|}\begin{pmatrix} \langle \boldsymbol{u}, \boldsymbol{u} \rangle & \langle \boldsymbol{u}, \boldsymbol{v} \rangle & \langle \boldsymbol{u}, \boldsymbol{w} \rangle \\ \langle \boldsymbol{v}, \boldsymbol{u} \rangle & \langle \boldsymbol{v}, \boldsymbol{v} \rangle & \langle \boldsymbol{v}, \boldsymbol{w} \rangle \\ \langle \boldsymbol{w}, \boldsymbol{u} \rangle & \langle \boldsymbol{w}, \boldsymbol{v} \rangle & \langle \boldsymbol{w}, \boldsymbol{w} \rangle \end{pmatrix}. \tag{2.3}$$

The listings of the routines to compute the previous element matrices are given in Appendix B.1 We now give the classical assembly algorithm using these element matrices with a loop through the triangles.

3. The classical algorithm. We describe the assembly of a given $n_q \times n_q$ matrix M from its associated 3×3 element matrix E. We denote by "ElemMat" the routine which computes the element matrix E.

Listing 1
Classical matrix assembly code in Matlab/Octave

```
M=sparse(nq,nq);
for k=1:nme
    E=ElemMat(areas(k),...);
    for il=1:3
        i=me(il,k);
        for jl=1:3
            j=me(jl,k);
            M(i,j)=M(i,j)+E(il,jl);
        end
    end
end
```

We aim to compare the performances of this code (see Appendix B.2 for the complete listings) with those obtained with FreeFEM++ [13]. The FreeFEM++ commands to build the mass, weighted mass and stiffness matrices are given in Listing 2. On Figure 3.1, we show the computation times (in seconds) versus the number of vertices n_q of the mesh (unit disk), for the classical assembly and FreeFEM++ codes. The values of the computation times are given in Appendix A.1. We observe that the complexity is $\mathcal{O}(n_q^2)$ (quadratic) for the Matlab/Octave codes, while the complexity seems to be $\mathcal{O}(n_q)$ (linear) for FreeFEM++.

LISTING 2
Matrix assembly code in FreeFEM++

```
mesh Th(...);
fespace Vh(Th,P1); // P1 FE-space
varf vMass (u,v)= int2d(Th)( u*v);
varf vMassW (u,v)= int2d(Th)( w*u*v);
varf vStiff (u,v)= int2d(Th)( dx(u)*dx(v)
                              + dy(u)*dy(v) );

matrix M= vMass(Vh,Vh);      // Mass matrix assembly
matrix Mw = vMassW(Vh,Vh);   // Weighted mass matrix assembly
matrix S = vStiff(Vh,Vh);    // Stiffness matrix assembly
```

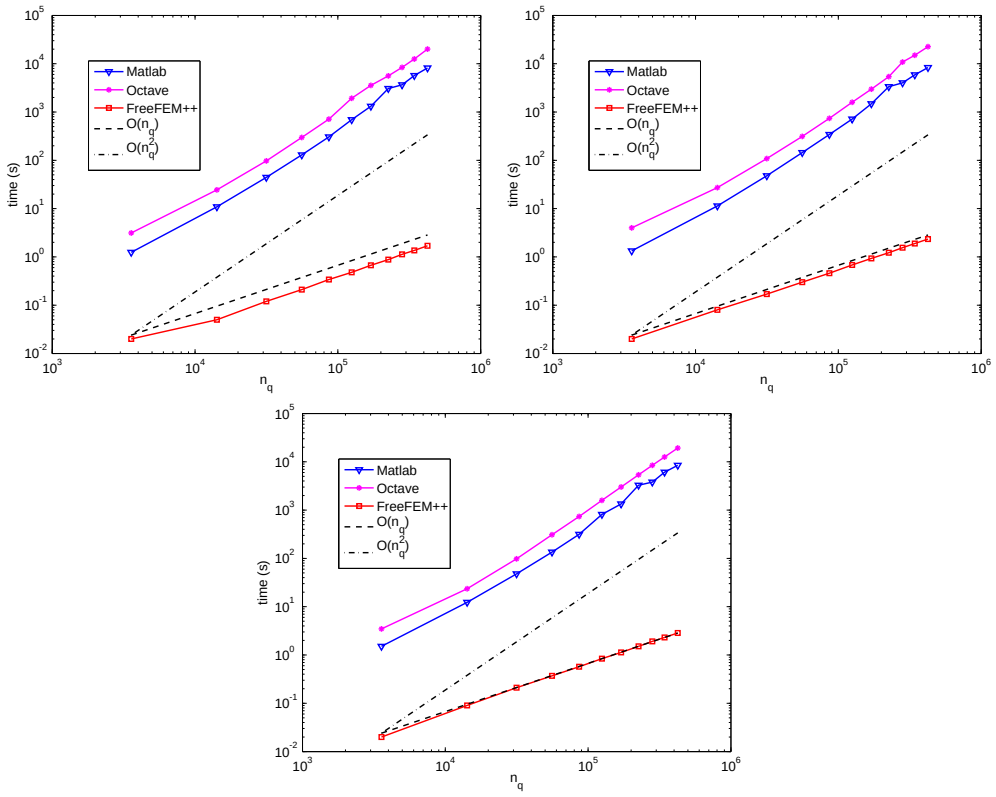


FIG. 3.1. *Comparison of the classical matrix assembly code in Matlab/Octave with FreeFEM++, for the mass (top left), weighted mass (top right) and stiffness (bottom) matrices.*

We have surprisingly observed that the Matlab performances may be improved using an older Matlab release (see Appendix C)

Our objective is to propose optimizations of the classical code that lead to more efficient codes with computational costs comparable to those obtained with FreeFEM++. A first improvement of the classical algorithm (Listing 1) is to vectorize the two local loops, see Listing 3 (the complete listings are given in Appendix B.3).

LISTING 3
Optimized matrix assembly code - version 0

```
M=sparse(nq,nq);
for k=1:nme
  I=me(:,k);
  M(I,I)=M(I,I)+ElemMat(areas(k),...);
end
```

However the complexity of this algorithm is still quadratic (i.e. $\mathcal{O}(\mathrm{n_q^2})$).

In the next section, we explain the storage of sparse matrices in Matlab/Octave in order to justify this lack of efficiency.

4. Sparse matrices storage. In Matlab/Octave, a sparse matrix $\mathbb{A} \in \mathcal{M}_{M,N}(\mathbb{R})$, with nnz non-zeros elements, is stored with CSC (Compressed Sparse Column) format using the following three arrays:

- $aa(1:nnz)$: which contains the nnz non-zeros elements of $\mathbb{A}$ stored column-wise,
- $ia(1:nnz)$: which contains the row numbers of the elements stored in aa,
- $ja(1:N+1)$: which allows to find the elements of a column of $\mathbb{A}$, with the information that the first non-zero element of the column k of $\mathbb{A}$ is in the $ja(k)$-th position in the array aa. We have $ja(1) = 1$ and $ja(N+1) = nnz + 1$.

For example, with the matrix

$$\mathbb{A} = \begin{pmatrix} 1. & 0. & 0. & 6. \\ 0. & 5. & 0. & 4. \\ 0. & 1. & 2. & 0. \end{pmatrix},$$

we have $M = 3$, $N = 4$, $nnz = 6$ and

aa	1.	5.	1.	2.	6.	4.
ia	1	2	3	3	1	2
ja	1	2	4	5	7	

The first non-zero element in column $k = 3$ of $\mathbb{A}$ is 2, the position of this number in aa is 4, thus $ja(3) = 4$.

We now describe the operations to be done on the arrays aa, ia and ja if we modify the matrix $\mathbb{A}$ by taking $\mathbb{A}(1,2) = 8$. It becomes

$$\mathbb{A} = \begin{pmatrix} 1. & 8. & 0. & 6. \\ 0. & 5. & 0. & 4. \\ 0. & 1. & 2. & 0. \end{pmatrix}.$$

In this case, a zero element of $\mathbb{A}$ has been replaced by the non-zero value 8 which must be stored in the arrays while no space is provided. We suppose that the arrays are sufficiently large (to avoid memory space problems), we must then shift one cell all the values in the arrays aa and ia from the third position and then copy the value 8 in $aa(3)$ and the value 1 (row number) in $ia(3)$:

aa	1.	8.	5.	1.	2.	6.	4.
ia	1	1	2	3	3	1	2

For the array ja, we increment of 1 the values after the position 2 :

ja	1	2	5	6	8

The repetition of these operations is expensive upon assembly of the matrix in the previous codes. Moreover, we haven't considered dynamic reallocation problems that may also occur.

We now present the optimized version 1 of the code that will allow to improve the performance of the classical code.

5. Optimized matrix assembly - version 1 (OptV1). We will use the following call of the `sparse` Matlab function:

M = **sparse**(I,J,K,m,n);

This command returns a sparse matrix M of size m × n such that M(I(k),J(k)) = K(k). The vectors I, J and K have the same length. The zero elements of K are not taken into account and the elements of K having the same indices in I and J are summed.

The idea is to create three global 1d-arrays $\boldsymbol{I}_g$, $\boldsymbol{J}_g$ and $\boldsymbol{K}_g$ allowing the storage of the element matrices as well as the position of their elements in the global matrix. The length of each array is $9\mathrm{n_{me}}$. Once these arrays are created, the matrix assembly is obtained with the command

M = **sparse**(Ig,Jg,Kg,nq,nq);

To create these three arrays, we first define three local arrays $\boldsymbol{K}_k^e$, $\boldsymbol{I}_k^e$ and $\boldsymbol{J}_k^e$ of nine elements obtained from a generic element matrix $\mathbb{E}(T_k)$ of dimension 3 :

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

We have chosen a column-wise numbering for 1d-arrays in Matlab/Octave implementation, but for representation convenience we draw them in line format,

$$\mathbb{E}(T_k) = \begin{pmatrix} e_{1,1}^k & e_{1,2}^k & e_{1,3}^k \\ e_{2,1}^k & e_{2,2}^k & e_{2,3}^k \\ e_{3,1}^k & e_{3,2}^k & e_{3,3}^k \end{pmatrix} \Longrightarrow$$

$\boldsymbol{K}_k^e$:	$e_{1,1}^k$	$e_{2,1}^k$	$e_{3,1}^k$	$e_{1,2}^k$	$e_{2,2}^k$	$e_{3,2}^k$	$e_{1,3}^k$	$e_{2,3}^k$	$e_{3,3}^k$
$\boldsymbol{I}_k^e$:	i_1^k	i_2^k	i_3^k	i_1^k	i_2^k	i_3^k	i_1^k	i_2^k	i_3^k
$\boldsymbol{J}_k^e$:	i_1^k	i_1^k	i_1^k	i_2^k	i_2^k	i_2^k	i_3^k	i_3^k	i_3^k

with $i_1^k = \mathrm{me}(1,k)$, $i_2^k = \mathrm{me}(2,k)$, $i_3^k = \mathrm{me}(3,k)$.

To create the three arrays $\boldsymbol{K}_k^e$, $\boldsymbol{I}_k^e$ and $\boldsymbol{J}_k^e$, in Matlab/Octave, one can use the following commands :

```
E  = ElemMat(areas(k), ...);        % E  : 3-by-3 matrix
Ke = E(:);                          % Ke : 9-by-1 matrix
Ie = me([1 2 3 1 2 3 1 2 3],k);     % Ie : 9-by-1 matrix
Je = me([1 1 1 2 2 2 3 3 3],k);     % Je : 9-by-1 matrix
```

From these arrays, it is then possible to build the three global arrays $\boldsymbol{I}_g$, $\boldsymbol{J}_g$ and $\boldsymbol{K}_g$, of size $9\mathrm{n_{me}} \times 1$ defined by : $\forall k \in \{1,\ldots,\mathrm{n_{me}}\}$, $\forall il \in \{1,\ldots,9\}$,

$$\begin{aligned} \boldsymbol{K}_g(9(k-1)+il) &= \boldsymbol{K}_k^e(il), \\ \boldsymbol{I}_g(9(k-1)+il) &= \boldsymbol{I}_k^e(il), \\ \boldsymbol{J}_g(9(k-1)+il) &= \boldsymbol{J}_k^e(il). \end{aligned}$$

On Figure 5.1, we show the insertion of the local array $\boldsymbol{K}_k^e$ into the global 1d-array $\boldsymbol{K}_g$, and, for representation convenience, we draw them in line format. We make the same operation for the two other arrays.

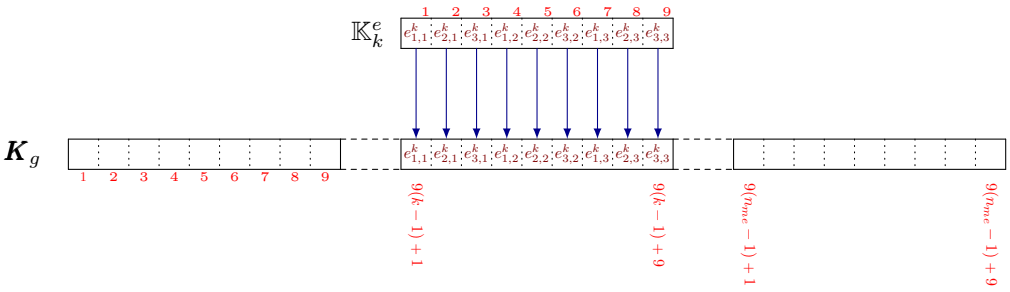


FIG. 5.1. *Insertion of an element matrix in the global array - Version 1*

We give in Listing 4 the Matlab/Octave associated code where the global vectors $\boldsymbol{I}_g$, $\boldsymbol{J}_g$ and $\boldsymbol{K}_g$ are stored column-wise. The complete listings and the values of the computation times are given in Appendices B.4 and A.3 respectively. On Figure 5.2, we show the computation times of the Matlab, Octave and FreeFEM++ codes versus the number of vertices of the mesh (unit disk). The complexity of the Matlab/Octave codes seems now linear (i.e. $\mathcal{O}(\mathrm{n_q})$) as for FreeFEM++. However, FreeFEM++ is still much more faster than Matlab/Octave (about a factor 5 for the mass matrix, 6.5 for the weighted mass matrix and 12.5 for the stiffness matrix, in Matlab).

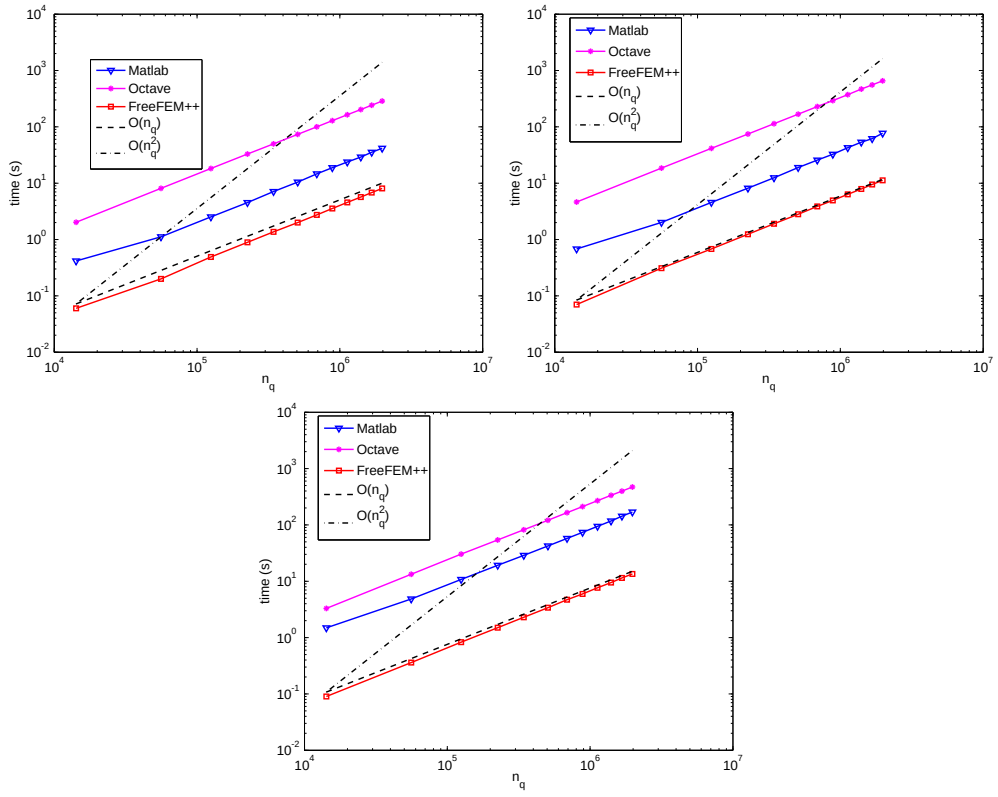


FIG. 5.2. *Comparison of the matrix assembly codes :* `OptV1` *in Matlab/Octave and FreeFEM++, for the mass (top left), weighted mass (top right) and stiffness (bottom) matrices.*

LISTING 4
Optimized matrix assembly code - version 1

```
Ig=zeros(9*nme,1); Jg=zeros(9*nme,1); Kg=zeros(9*nme,1);
ii=[1 2 3 1 2 3 1 2 3]; jj=[1 1 1 2 2 2 3 3 3];
kk=1:9;
for k=1:nme
  E=ElemMat(areas(k), ...);
  Ig(kk)=me(ii,k);
  Jg(kk)=me(jj,k);
  Kg(kk)=E(:);
  kk=kk+9;
end
M=sparse(Ig,Jg,Kg,nq,nq);
```

To further improve the efficiency of the codes, we introduce now a second optimized version of the assembly algorithm.

6. Optimized matrix assembly - version 2 (OptV2). We present the optimized version 2 of the algorithm where no loop is used.

We define three 2d-arrays that allow to store all the element matrices as well as their positions in the global matrix. We denote by $\mathbb{K}_g$, $\mathbb{I}_g$ and $\mathbb{J}_g$ these 9-by-$\mathrm{n_{me}}$ arrays, defined $\forall k \in \{1,\dots,\mathrm{n_{me}}\}$, $\forall il \in \{1,\dots,9\}$ by

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

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

A natural way to build these three arrays consists in using a loop through the triangles T_k in which we insert the local arrays column-wise, see Figure 6.1. Once these arrays are determined, the matrix assembly is obtained with the Matlab/Octave command

M = **sparse**(Ig(:),Jg(:),Kg(:),nq,nq);

We remark that the matrices containing global indices $\mathbb{I}_g$ and $\mathbb{J}_g$ may be computed, in Matlab/Octave, without any loop. For the computation of these two matrices, on the left we give the usual code and on the right the vectorized code :

```
Ig=zeros(9,nme); Jg=zeros(9,nme);
for k=1:nme
  Ig(:,k)=me([1 2 3 1 2 3 1 2 3],k);
  Jg(:,k)=me([1 1 1 2 2 2 3 3 3],k);
end
```

```
Ig=me([1 2 3 1 2 3 1 2 3],:);
Jg=me([1 1 1 2 2 2 3 3 3],:);
```

Another way to present this computation, used and adapted in Section 7, is given by

REMARK 6.1. *Denoting* $\mathcal{I}_k = [\mathrm{me}(1,k),\ \mathrm{me}(2,k),\ \mathrm{me}(3,k)]$ *and*

$$\mathbb{T} = \begin{pmatrix} \mathcal{I}_1(1) & \dots & \mathcal{I}_k(1) & \dots & \mathcal{I}_{\mathrm{n_{me}}}(1) \\ \mathcal{I}_1(2) & \dots & \mathcal{I}_k(2) & \dots & \mathcal{I}_{\mathrm{n_{me}}}(2) \\ \mathcal{I}_1(3) & \dots & \mathcal{I}_k(3) & \dots & \mathcal{I}_{\mathrm{n_{me}}}(3) \end{pmatrix},$$

then, in that case $\mathbb{T} = \mathrm{me}$, *and* $\mathbb{I}_g$ *and* $\mathbb{J}_g$ *may be computed from* $\mathbb{T}$ *as follows:*

```
ii=[1 1 1; 2 2 2; 3 3 3]; jj=ii';
Ig=T(ii(:),:); Jg=T(jj(:),:);
```

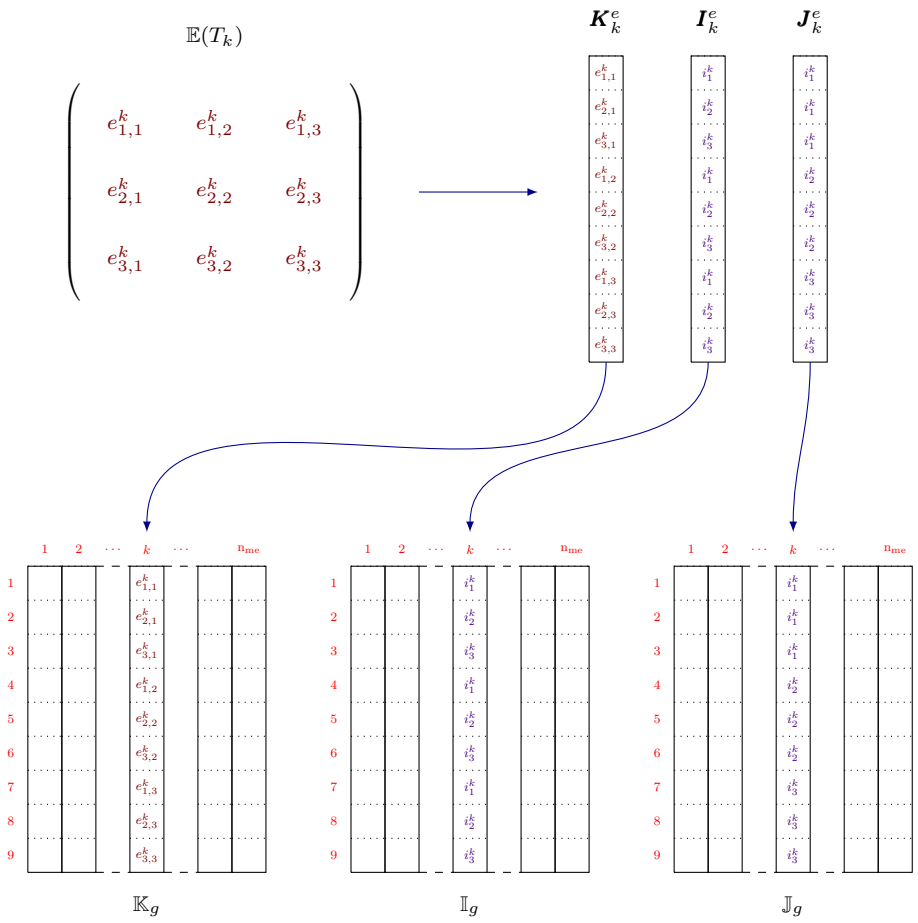


FIG. 6.1. *Insertion of an element matrix in the global array - Version 2*

It remains to vectorize the computation of the 2d-array $\mathbb{K}_g$. The usual code, corresponding to a column-wise computation, is :

LISTING 5
Usual assembly (column-wise computation)

```
Kg=zeros(9,nme);
for k=1:nme
  E=ElemMat(areas(k), ...);
  Kg(:,k)=E(:);
end
```

The vectorization of this code is done by the computation of the array $\mathbb{K}_g$ row-wise, for each matrix assembly. This corresponds to the permutation of the loop through the elements with the local loops, in the classical matrix assembly code (see Listing 1). This vectorization differs from the one proposed in [12] as it doesn't use any quadrature formula and from the one in [2] by the full vectorization of the arrays $\mathbb{I}_g$ and $\mathbb{J}_g$.

We describe below this method for each matrix defined in Section 2.

6.1. Mass matrix assembly. The element mass matrix $\mathbb{M}^e(T_k)$ associated to the triangle T_k is given by (2.1). The array $\mathbb{K}_g$ is defined by : $\forall k \in \{1, \ldots, \mathrm{n_{me}}\}$,

$$\mathbb{K}_g(\alpha, k) = \frac{|T_k|}{6}, \ \forall \alpha \in \{1, 5, 9\},$$
$$\mathbb{K}_g(\alpha, k) = \frac{|T_k|}{12}, \ \forall \alpha \in \{2, 3, 4, 6, 7, 8\}.$$

Then we build two arrays A_6 and A_{12} of size $1 \times \mathrm{n_{me}}$ such that $\forall k \in \{1, \ldots, \mathrm{n_{me}}\}$:

$$A_6(k) = \frac{|T_k|}{6}, \qquad A_{12}(k) = \frac{|T_k|}{12}.$$

The rows $\{1, 5, 9\}$ in the array $\mathbb{K}_g$ correspond to A_6 and the rows $\{2, 3, 4, 6, 7, 8\}$ to A_{12}, see Figure 6.2. The Matlab/Octave code associated to this technique is :

LISTING 6
Optimized matrix assembly code - version 2 (Mass matrix)

```
function [M]=MassAssemblingP1OptV2(nq,nme,me,areas)
Ig = me([1 2 3 1 2 3 1 2 3],:);
Jg = me([1 1 1 2 2 2 3 3 3],:);
A6=areas/6;
A12=areas/12;
Kg = [A6;A12;A12;A12;A6;A12;A12;A12;A6];
M = sparse(Ig(:),Jg(:),Kg(:),nq,nq);
```

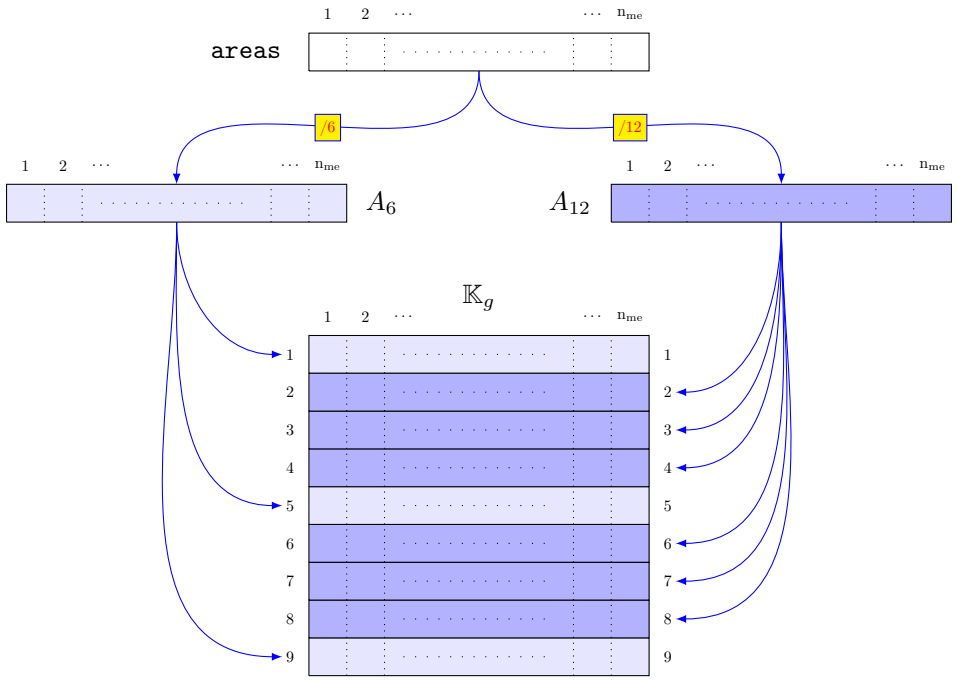


FIG. 6.2. *Mass matrix assembly - Version 2*

6.2. Weighted mass matrix assembly. The element weighted mass matrices $\mathbb{M}^{e,[\tilde{w}]}(T_k)$ are given by (2.2). We introduce the array $\boldsymbol{T_w}$ of length $\mathrm{n_q}$ defined by $\boldsymbol{T_w}(i) = w(\mathrm{q}^i)$, for all $i \in \{1, \ldots, \mathrm{n_q}\}$ and the three arrays $\boldsymbol{W}_\alpha$, $1 \leqslant \alpha \leqslant 3$, of length $\mathrm{n_{me}}$, defined for all $k \in \{1, \ldots, \mathrm{n_{me}}\}$ by $\boldsymbol{W}_\alpha(k) = \frac{|T_k|}{30}\boldsymbol{T_w}(\mathrm{me}(\alpha, k))$.

The code for computing these three arrays is given below, in a non-vectorized form (on the left) and in a vectorized form (on the right):

```
W1=zeros(1,nme);
W2=zeros(1,nme);
W3=zeros(1,nme);
for k=1:nme
  W1(k)=Tw(me(1,k))*areas(k)/30;
  W2(k)=Tw(me(2,k))*areas(k)/30;
  W3(k)=Tw(me(3,k))*areas(k)/30;
end
```

```
W1=Tw(me(1,:)).*areas/30;
W2=Tw(me(2,:)).*areas/30;
W3=Tw(me(3,:)).*areas/30;
```

We follow the method described on Figure 6.1. We have to vectorize the computation of $\mathbb{K}_g$ (Listing 5). Let $\boldsymbol{K}_1$, $\boldsymbol{K}_2$, $\boldsymbol{K}_3$, $\boldsymbol{K}_5$, $\boldsymbol{K}_6$, $\boldsymbol{K}_9$ be six arrays of length $\mathrm{n_{me}}$ defined by

$$\boldsymbol{K}_1 = 3\boldsymbol{W}_1 + \boldsymbol{W}_2 + \boldsymbol{W}_3, \quad \boldsymbol{K}_2 = \boldsymbol{W}_1 + \boldsymbol{W}_2 + \frac{\boldsymbol{W}_3}{2}, \quad \boldsymbol{K}_3 = \boldsymbol{W}_1 + \frac{\boldsymbol{W}_2}{2} + \boldsymbol{W}_3,$$
$$\boldsymbol{K}_5 = \boldsymbol{W}_1 + 3\boldsymbol{W}_2 + \boldsymbol{W}_3, \quad \boldsymbol{K}_6 = \frac{\boldsymbol{W}_1}{2} + \boldsymbol{W}_2 + \boldsymbol{W}_3, \quad \boldsymbol{K}_9 = \boldsymbol{W}_1 + \boldsymbol{W}_2 + 3\boldsymbol{W}_3.$$

The element weighted mass matrix and the k-th column of $\mathbb{K}_g$ are respectively :

$$\mathbb{M}^{e,[\tilde{w}]}(T_k) = \begin{pmatrix} \boldsymbol{K}_1(k) & \boldsymbol{K}_2(k) & \boldsymbol{K}_3(k) \\ \boldsymbol{K}_2(k) & \boldsymbol{K}_5(k) & \boldsymbol{K}_6(k) \\ \boldsymbol{K}_3(k) & \boldsymbol{K}_6(k) & \boldsymbol{K}_9(k) \end{pmatrix}, \qquad \mathbb{K}_g(:,k) = \begin{pmatrix} \boldsymbol{K}_1(k) \\ \boldsymbol{K}_2(k) \\ \boldsymbol{K}_3(k) \\ \boldsymbol{K}_2(k) \\ \boldsymbol{K}_5(k) \\ \boldsymbol{K}_6(k) \\ \boldsymbol{K}_3(k) \\ \boldsymbol{K}_6(k) \\ \boldsymbol{K}_9(k) \end{pmatrix}.$$

Thus we obtain the following vectorized code for $\mathbb{K}_g$:

```
K1 = 3*W1+W2+W3;
K2 = W1+W2+W3/2;
K3 = W1+W2/2+W3;
K5 = W1+3*W2+W3;
K6 = W1/2+W2+W3;
K9 = W1+W2+3*W3;
Kg = [K1;K2;K3;K2;K5;K6;K3;K6;K9];
```

We represent this technique on Figure 6.3.

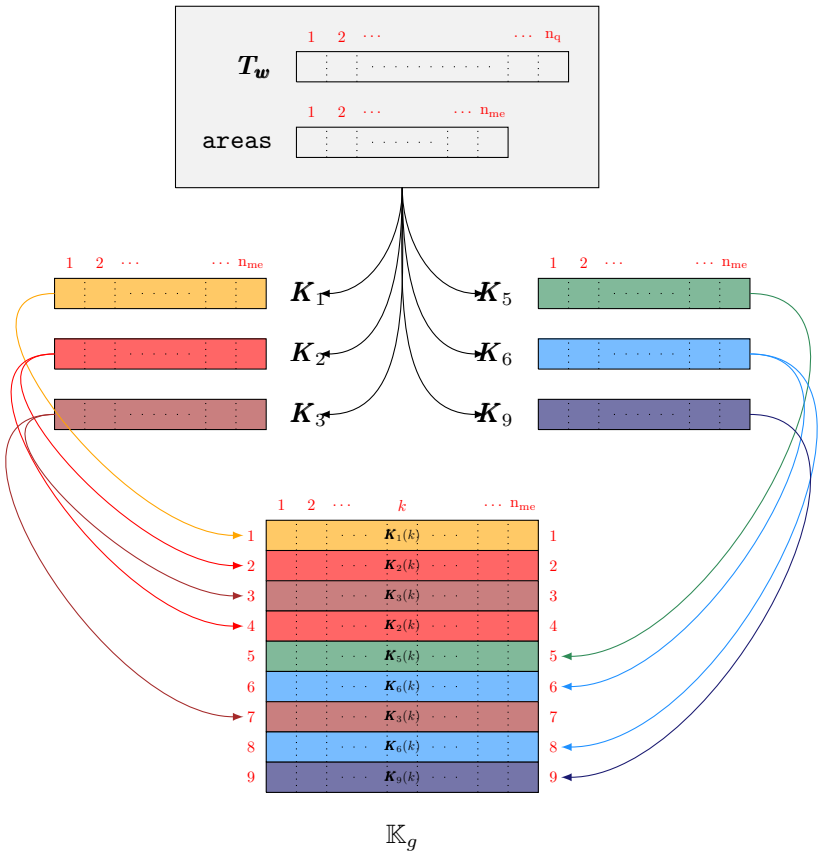


FIG. 6.3. *Weighted mass matrix assembly - Version 2*

Finally, the complete vectorized code using element matrix symmetry is :

LISTING 7
Optimized assembly - version 2 (Weighted mass matrix)

```
function M=MassWAssemblingP1OptV2(nq,nme,me,areas,Tw)
W1=Tw(me(1,:)).*areas/30;
W2=Tw(me(2,:)).*areas/30;
W3=Tw(me(3,:)).*areas/30;
Kg=zeros(9,nme);
Kg(1,:) = 3*W1+W2+W3;
Kg(2,:) = W1+W2+W3/2;
Kg(3,:) = W1+W2/2+W3;
Kg(5,:) = W1+3*W2+W3;
Kg(6,:) = W1/2+W2+W3;
Kg(9,:) = W1+W2+3*W3;
Kg([4, 7, 8],:)=Kg([2, 3, 6],:);
clear W1 W2 W3
Ig = me([1 2 3 1 2 3 1 2 3],:);
Jg = me([1 1 1 2 2 2 3 3 3],:);
M = sparse(Ig(:),Jg(:),Kg(:),nq,nq);
```

6.3. Stiffness matrix assembly. The vertices of the triangle T_k are $\mathrm{q}^{\mathrm{me}(\alpha,k)}$, $1 \leqslant \alpha \leqslant 3$. We define $\boldsymbol{u}^k = \mathrm{q}^{\mathrm{me}(2,k)} - \mathrm{q}^{\mathrm{me}(3,k)}$, $\boldsymbol{v}^k = \mathrm{q}^{\mathrm{me}(3,k)} - \mathrm{q}^{\mathrm{me}(1,k)}$ and $\boldsymbol{w}^k = \mathrm{q}^{\mathrm{me}(1,k)} - \mathrm{q}^{\mathrm{me}(2,k)}$. Then, the element stiffness matrix $\mathbb{S}^e(T_k)$ associated to T_k is defined by (2.3) with $\boldsymbol{u} = \boldsymbol{u}^k$, $\boldsymbol{v} = \boldsymbol{v}^k$, $\boldsymbol{w} = \boldsymbol{w}^k$ and $T = T_k$. Let $\boldsymbol{K}_1$, $\boldsymbol{K}_2$, $\boldsymbol{K}_3$, $\boldsymbol{K}_5$, $\boldsymbol{K}_6$ and $\boldsymbol{K}_9$ be six arrays of length $\mathrm{n_{me}}$ such that, for all $k \in \{1, \ldots, \mathrm{n_{me}}\}$,

$$\boldsymbol{K}_1(k) = \frac{\langle \boldsymbol{u}^k, \boldsymbol{u}^k \rangle}{4|T_k|}, \quad \boldsymbol{K}_2(k) = \frac{\langle \boldsymbol{u}^k, \boldsymbol{v}^k \rangle}{4|T_k|}, \quad \boldsymbol{K}_3(k) = \frac{\langle \boldsymbol{u}^k, \boldsymbol{w}^k \rangle}{4|T_k|},$$

$$\boldsymbol{K}_5(k) = \frac{\langle \boldsymbol{v}^k, \boldsymbol{v}^k \rangle}{4|T_k|}, \quad \boldsymbol{K}_6(k) = \frac{\langle \boldsymbol{v}^k, \boldsymbol{w}^k \rangle}{4|T_k|}, \quad \boldsymbol{K}_9(k) = \frac{\langle \boldsymbol{w}^k, \boldsymbol{w}^k \rangle}{4|T_k|}.$$

With these arrays, the vectorized assembly method is similar to the one shown in Figure 6.3 and the corresponding code is :

```
Kg = [K1;K2;K3;K2;K5;K6;K3;K6;K9];
S = sparse(Ig(:),Jg(:),Kg(:),nq,nq);
```

We now describe the vectorized computation of these six arrays. We introduce the 2-by-$\mathrm{n_{me}}$ arrays $\boldsymbol{q}_\alpha$, $\alpha \in \{1, \ldots, 3\}$, containing the coordinates of the three vertices of the triangle T_k :

$$\boldsymbol{q}_\alpha(1,k) = \mathrm{q}(1, \mathrm{me}(\alpha,k)), \quad \boldsymbol{q}_\alpha(2,k) = \mathrm{q}(2, \mathrm{me}(\alpha,k)).$$

We give below the code to compute these arrays, in a non-vectorized form (on the left) and in a vectorized form (on the right) :

```
q1=zeros(2,nme);q2=zeros(2,nme);q3=zeros(2,nme);
for k=1:nme
  q1(:,k)=q(:,me(1,k));
  q2(:,k)=q(:,me(2,k));
  q3(:,k)=q(:,me(3,k));
end
```

```
q1=q(:,me(1,:));
q2=q(:,me(2,:));
q3=q(:,me(3,:));
```

We trivially obtain the 2-by-$\mathrm{n_{me}}$ arrays $\boldsymbol{u}$, $\boldsymbol{v}$ and $\boldsymbol{w}$ whose k-th column is $\boldsymbol{u}^k$, $\boldsymbol{v}^k$ and $\boldsymbol{w}^k$ respectively.

The associated code is :

```
u=q2-q3;
v=q3-q1;
w=q1-q2;
```

The operators .*, ./ (element-wise arrays multiplication and division) and the function **sum**(.,1) (row-wise sums) allow to compute all arrays. For example, $\boldsymbol{K}_2$ is computed using the following vectorized code :

```
K2=sum(u.*v,1)./(4*areas);
```

Then, the complete vectorized function using element matrix symmetry is :

LISTING 8
Optimized matrix assembly code - version 2 (Stiffness matrix)

```
function S=StiffAssemblingP1OptV2(nq,nme,q,me,areas)
q1 =q(:,me(1,:)); q2 =q(:,me(2,:)); q3 =q(:,me(3,:));
u = q2-q3; v=q3-q1; w=q1-q2;
areas4=4*areas;
Kg=zeros(9,nme);
Kg(1,:)=sum(u.*u,1)./areas4; % K1
Kg(2,:)=sum(v.*u,1)./areas4; % K2
Kg(3,:)=sum(w.*u,1)./areas4; % K3
Kg(5,:)=sum(v.*v,1)./areas4; % K5
Kg(6,:)=sum(w.*v,1)./areas4; % K6
Kg(9,:)=sum(w.*w,1)./areas4; % K9
Kg([4, 7, 8],:)=Kg([2, 3, 6],:);
clear q1 q2 q3 areas4 u v w
Ig = me([1 2 3 1 2 3 1 2 3],:);
Jg = me([1 1 1 2 2 2 3 3 3],:);
S = sparse(Ig(:),Jg(:),Kg(:),nq,nq);
```

6.4. Comparison with FreeFEM++. On Figure 6.4, we show the computation times of the FreeFEM++ and `OptV2` Matlab/Octave codes, versus n_q.

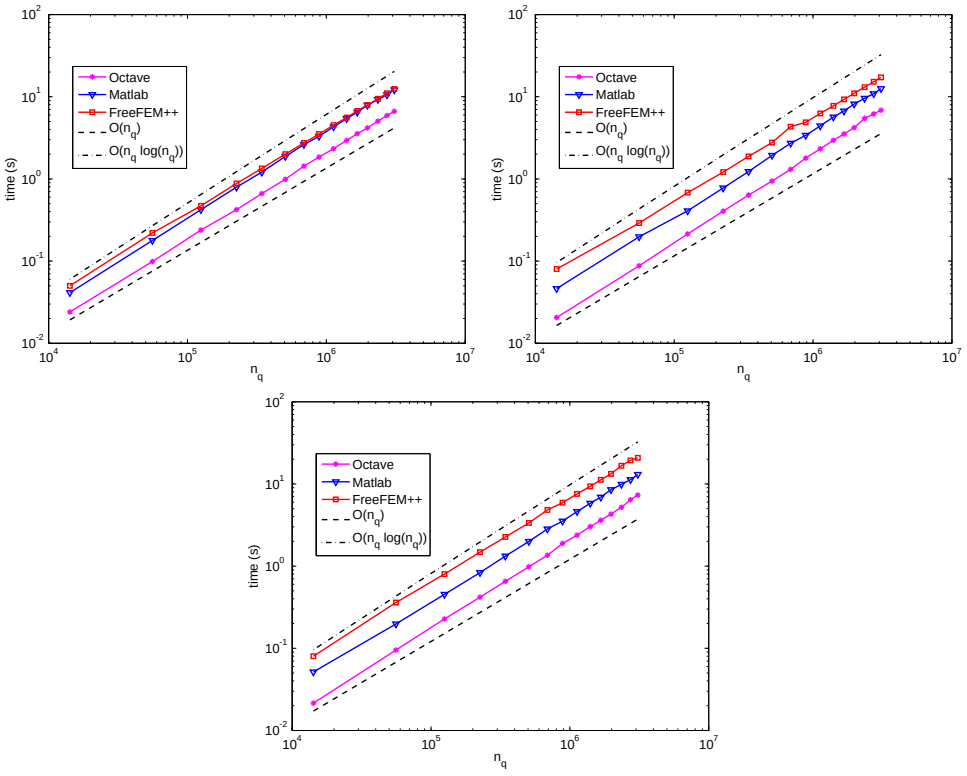


FIG. 6.4. *Comparison of the matrix assembly codes : `OptV2` in Matlab/Octave and FreeFEM++, for the mass (top left), weighted mass (top right) and stiffness (bottom) matrices.*

The computation times values are given in Appendix A.4. The complexity of the Matlab/Octave codes is still linear ($\mathcal{O}(n_q)$) and slightly better than the one of FreeFEM++.

REMARK 6.2. *We observed that only with the* `OptV2` *codes, Octave gives better results than Matlab. For the other versions of the codes, not fully vectorized, the JIT-Accelerator (Just-In-Time) of Matlab allows significantly better performances than Octave (JIT compiler for GNU Octave is under development).*

Furthermore, we can improve Matlab performances using SuiteSparse packages from T. Davis [9], which is originally used in Octave. In our codes, using `cs_sparse` *function from SuiteSparse instead of Matlab* `sparse` *function is approximately 1.1 times faster for* `OptV1` *version and 2.5 times for* `OptV2` *version (see also Section 7).*

6.5. Comparison with other matrix assembly codes. We compare the matrix assembly codes proposed by L. Chen [2, 3], A. Hannukainen and M. Juntunen [12] and T. Rahman and J. Valdman [17] to the `OptV2` version developed in this paper, for the mass and stiffness matrices. The domain Ω is the unit disk. The computations have been done on our reference computer. On Figure 6.5, with Matlab (top) and Octave (bottom), we show the computation times versus the number of vertices of the mesh, for these different codes. The associated values are given in Tables 6.1 to 6.4. For large sparse matrices, our `OptV2` version allows gains in computational time of 5% to 20%, compared to the other vectorized codes (for sufficiently large meshes).

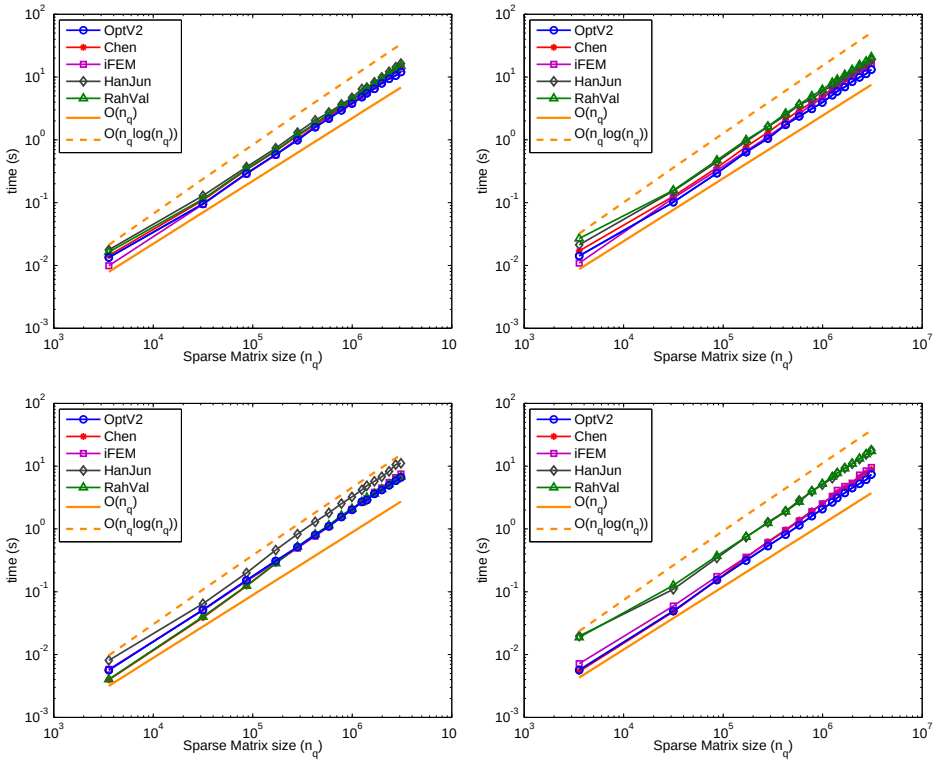


FIG. 6.5. *Comparison of the assembly codes in Matlab R2012b (top) and Octave 3.6.3 (bottom):* `OptV2` *and [2, 3, 12, 17], for the mass (left) and stiffness (right) matrices.*

n_q	OptV2	Chen	iFEM	HanJun	RahVal
86488	0.291 (s) x 1.00	0.333 (s) x 0.87	0.288 (s) x 1.01	0.368 (s) x 0.79	0.344 (s) x 0.85
170355	0.582 (s) x 1.00	0.661 (s) x 0.88	0.575 (s) x 1.01	0.736 (s) x 0.79	0.673 (s) x 0.86
281769	0.986 (s) x 1.00	1.162 (s) x 0.85	1.041 (s) x 0.95	1.303 (s) x 0.76	1.195 (s) x 0.83
424178	1.589 (s) x 1.00	1.735 (s) x 0.92	1.605 (s) x 0.99	2.045 (s) x 0.78	1.825 (s) x 0.87
582024	2.179 (s) x 1.00	2.438 (s) x 0.89	2.267 (s) x 0.96	2.724 (s) x 0.80	2.588 (s) x 0.84
778415	2.955 (s) x 1.00	3.240 (s) x 0.91	3.177 (s) x 0.93	3.660 (s) x 0.81	3.457 (s) x 0.85
992675	3.774 (s) x 1.00	4.146 (s) x 0.91	3.868 (s) x 0.98	4.682 (s) x 0.81	4.422 (s) x 0.85
1251480	4.788 (s) x 1.00	5.590 (s) x 0.86	5.040 (s) x 0.95	6.443 (s) x 0.74	5.673 (s) x 0.84
1401129	5.526 (s) x 1.00	5.962 (s) x 0.93	5.753 (s) x 0.96	6.790 (s) x 0.81	6.412 (s) x 0.86
1671052	6.507 (s) x 1.00	7.377 (s) x 0.88	7.269 (s) x 0.90	8.239 (s) x 0.79	7.759 (s) x 0.84
1978602	7.921 (s) x 1.00	8.807 (s) x 0.90	8.720 (s) x 0.91	9.893 (s) x 0.80	9.364 (s) x 0.85
2349573	9.386 (s) x 1.00	10.969 (s) x 0.86	10.388 (s) x 0.90	12.123 (s) x 0.77	11.160 (s) x 0.84
2732448	10.554 (s) x 1.00	12.680 (s) x 0.83	11.842 (s) x 0.89	14.343 (s) x 0.74	13.087 (s) x 0.81
3085628	12.034 (s) x 1.00	14.514 (s) x 0.83	13.672 (s) x 0.88	16.401 (s) x 0.73	14.950 (s) x 0.80

TABLE 6.1

Computational cost, in Matlab (R2012b), of the **`Mass`** *matrix assembly versus* n_q, *with the* **`OptV2`** *version (column 2) and with the codes in [2, 3, 12, 17] (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
86488	0.294 (s) x 1.00	0.360 (s) x 0.82	0.326 (s) x 0.90	0.444 (s) x 0.66	0.474 (s) x 0.62
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
281769	1.048 (s) x 1.00	1.316 (s) x 0.80	1.119 (s) x 0.94	1.616 (s) x 0.65	1.621 (s) x 0.65
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
582024	2.369 (s) x 1.00	2.932 (s) x 0.81	2.565 (s) x 0.92	3.620 (s) x 0.65	3.648 (s) x 0.65
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
992675	3.933 (s) x 1.00	4.862 (s) x 0.81	4.525 (s) x 0.87	5.948 (s) x 0.66	6.270 (s) x 0.63
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
1401129	5.901 (s) x 1.00	7.590 (s) x 0.78	7.148 (s) x 0.83	8.510 (s) x 0.69	9.132 (s) x 0.65
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
1978602	8.410 (s) x 1.00	10.845 (s) x 0.78	10.153 (s) x 0.83	12.315 (s) x 0.68	13.006 (s) x 0.65
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
2732448	11.255 (s) x 1.00	14.259 (s) x 0.79	13.977 (s) x 0.81	17.035 (s) x 0.66	17.774 (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 6.2

Computational cost, in Matlab (R2012b), of the **`Stiffness`** *matrix assembly versus* n_q, *with the* **`OptV2`** *version (column 2) and with the codes in [2, 3, 12, 17] (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
86488	0.152 (s) x 1.00	0.123 (s) x 1.24	0.148 (s) x 1.03	0.199 (s) x 0.76	0.125 (s) x 1.22
170355	0.309 (s) x 1.00	0.282 (s) x 1.10	0.294 (s) x 1.05	0.462 (s) x 0.67	0.284 (s) x 1.09
281769	0.515 (s) x 1.00	0.518 (s) x 1.00	0.497 (s) x 1.04	0.828 (s) x 0.62	0.523 (s) x 0.99
424178	0.799 (s) x 1.00	0.800 (s) x 1.00	0.769 (s) x 1.04	1.297 (s) x 0.62	0.820 (s) x 0.97
582024	1.101 (s) x 1.00	1.127 (s) x 0.98	1.091 (s) x 1.01	1.801 (s) x 0.61	1.145 (s) x 0.96
778415	1.549 (s) x 1.00	1.617 (s) x 0.96	1.570 (s) x 0.99	2.530 (s) x 0.61	1.633 (s) x 0.95
992675	2.020 (s) x 1.00	2.075 (s) x 0.97	2.049 (s) x 0.99	3.237 (s) x 0.62	2.095 (s) x 0.96
1251480	2.697 (s) x 1.00	2.682 (s) x 1.01	2.666 (s) x 1.01	4.190 (s) x 0.64	2.684 (s) x 1.01
1401129	2.887 (s) x 1.00	2.989 (s) x 0.97	3.025 (s) x 0.95	4.874 (s) x 0.59	3.161 (s) x 0.91
1671052	3.622 (s) x 1.00	3.630 (s) x 1.00	3.829 (s) x 0.95	5.750 (s) x 0.63	3.646 (s) x 0.99
1978602	4.176 (s) x 1.00	4.277 (s) x 0.98	4.478 (s) x 0.93	6.766 (s) x 0.62	4.293 (s) x 0.97
2349573	4.966 (s) x 1.00	5.125 (s) x 0.97	5.499 (s) x 0.90	8.267 (s) x 0.60	5.155 (s) x 0.96
2732448	5.862 (s) x 1.00	6.078 (s) x 0.96	6.575 (s) x 0.89	10.556 (s) x 0.56	6.080 (s) x 0.96
3085628	6.634 (s) x 1.00	6.793 (s) x 0.98	7.500 (s) x 0.88	11.109 (s) x 0.60	6.833 (s) x 0.97

TABLE 6.3

Computational cost, in Octave (3.6.3), of the **`Mass`** *matrix assembly versus* n_q, *with the* **`OptV2`** *version (column* 2*) and with the codes in [2, 3, 12, 17] (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
86488	0.154 (s) x 1.00	0.152 (s) x 1.01	0.175 (s) x 0.88	0.345 (s) x 0.44	0.371 (s) x 0.41
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
281769	0.536 (s) x 1.00	0.624 (s) x 0.86	0.609 (s) x 0.88	1.280 (s) x 0.42	1.243 (s) x 0.43
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
582024	1.148 (s) x 1.00	1.391 (s) x 0.83	1.336 (s) x 0.86	2.846 (s) x 0.40	2.707 (s) x 0.42
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
992675	2.077 (s) x 1.00	2.512 (s) x 0.83	2.514 (s) x 0.83	5.076 (s) x 0.41	5.236 (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
1401129	3.128 (s) x 1.00	3.761 (s) x 0.83	4.120 (s) x 0.76	7.766 (s) x 0.40	7.748 (s) x 0.40
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
1978602	4.482 (s) x 1.00	5.268 (s) x 0.85	5.361 (s) x 0.84	10.939 (s) x 0.41	10.935 (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
2732448	6.082 (s) x 1.00	7.782 (s) x 0.78	8.376 (s) x 0.73	15.339 (s) x 0.40	15.485 (s) x 0.39
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 6.4

Computational cost, in Octave (3.6.3), of the **`Stiffness`** *matrix assembly versus* n_q, *with the* **`OptV2`** *version (column* 2*) and with the codes in [2, 3, 12, 17] (columns* 3-6*) : time in seconds (top value) and speedup (bottom value). The speedup reference is* **`OptV2`** *version.*

7. Extension to linear elasticity. In this part we extend the codes of the previous sections to a linear elasticity matrix assembly.

Let $H_h^1(\Omega_h)$ be the finite dimensional space spanned by the P_1 Lagrange basis functions $\{\varphi_i\}_{i\in\{1,\dots,\mathrm{n_q}\}}$. Then, the space $(H_h^1(\Omega_h))^2$ is spanned by $\mathcal{B} = \{\boldsymbol{\psi}_l\}_{1\leqslant l\leqslant 2\mathrm{n_q}}$, with $\boldsymbol{\psi}_{2i-1} = \begin{pmatrix}\varphi_i\\0\end{pmatrix}$, $\boldsymbol{\psi}_{2i} = \begin{pmatrix}0\\\varphi_i\end{pmatrix}$, $1\leqslant i\leqslant \mathrm{n_q}$.

The example we consider is the elastic stiffness matrix $\mathbb{K}$, defined by

$$\mathbb{K}_{m,l} = \int_{\Omega_h} \underline{\boldsymbol{\epsilon}}^t(\boldsymbol{\psi}_m)\,\underline{\boldsymbol{\sigma}}(\boldsymbol{\psi}_l)dT,\ \forall (m,l)\in\{1,\dots,2\,\mathrm{n_q}\}^2,$$

where $\underline{\boldsymbol{\sigma}} = (\sigma_{xx},\sigma_{yy},\sigma_{xy})^t$ and $\underline{\boldsymbol{\epsilon}} = (\epsilon_{xx},\epsilon_{yy},2\epsilon_{xy})^t$ are the elastic stress and strain tensors respectively. We consider here linearized elasticity with small strain hypothesis (see for example [10]). Consequently, let $\mathcal{D}$ be the differential operator which links displacements $\boldsymbol{u}$ to strains:

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

This gives, in vectorial form and after reduction to the plane,

$$\mathcal{D} = \begin{pmatrix}\frac{\partial}{\partial x} & 0\\ 0 & \frac{\partial}{\partial y}\\ \frac{\partial}{\partial y} & \frac{\partial}{\partial x}\end{pmatrix}.$$

For the constitutive equation, Hooke's law is used and the material is supposed to be isotropic. Thus, the elasticity tensor denoted by $\mathbb{C}$ becomes a 3-by-3 matrix and can be defined by the Lamé parameters λ and μ, which are supposed constant on Ω and satisfying $\lambda+\mu>0$. Thus, the constitutive equation writes

$$\underline{\boldsymbol{\sigma}} = \mathbb{C}\underline{\boldsymbol{\epsilon}} = \begin{pmatrix}\lambda+2\mu & \lambda & 0\\ \lambda & \lambda+2\mu & 0\\ 0 & 0 & \mu\end{pmatrix}\underline{\boldsymbol{\epsilon}}.$$

Using the triangulation Ω_h of Ω, we have

$$\mathbb{K}_{m,l} = \sum_{k=1}^{\mathrm{n_{me}}} \mathbb{K}_{m,l}(T_k),\quad \text{with}\quad \mathbb{K}_{m,l}(T_k) = \int_{T_k} \underline{\boldsymbol{\epsilon}}^t(\boldsymbol{\psi}_m)\,\underline{\boldsymbol{\sigma}}(\boldsymbol{\psi}_l)dT,\ \forall (m,l)\in\{1,\dots,2\mathrm{n_q}\}^2.$$

Let $\mathcal{I}_k = [2\,\mathrm{me}(1,k)-1, 2\,\mathrm{me}(1,k),\ 2\,\mathrm{me}(2,k)-1, 2\,\mathrm{me}(2,k),\ 2\,\mathrm{me}(3,k)-1, 2\,\mathrm{me}(3,k)]$. Due to the support of functions $\boldsymbol{\psi}_l$, we have $\forall (l,m)\in(\{1,\dots,\mathrm{n_q}\}\setminus\mathcal{I}_k)^2$, $\mathbb{K}_{m,l}(T_k)=0$. Thus, we only have to compute $\mathbb{K}_{m,l}(T_k)$, $\forall (m,l)\in\mathcal{I}_k\times\mathcal{I}_k$, the other terms being zeros. We denote by $\mathbb{K}^e_{\alpha,\beta}(T_k) = \mathbb{K}_{\mathcal{I}_k(\alpha),\mathcal{I}_k(\beta)}(T_k)$, $\forall(\alpha,\beta)\in\{1,\dots,6\}^2$. Therefore, we introduce $\tilde{\mathcal{B}}(T_k) = \{\tilde{\boldsymbol{\psi}}_\alpha\}_{1\leqslant\alpha\leqslant 6}$ the local basis associated to a triangle T_k with $\tilde{\boldsymbol{\psi}}_\alpha = \boldsymbol{\psi}_{\mathcal{I}_k(\alpha)}$, $1\leqslant\alpha\leqslant 6$. We thus have $\tilde{\boldsymbol{\psi}}_{2\gamma-1} = \begin{pmatrix}\tilde{\varphi}_\gamma\\0\end{pmatrix}$, $\tilde{\boldsymbol{\psi}}_{2\gamma} = \begin{pmatrix}0\\\tilde{\varphi}_\gamma\end{pmatrix}$ $1\leqslant\gamma\leqslant 3$.

The element stiffness matrix $\mathbb{K}^e$ is given by

$$\mathbb{K}^e_{\alpha,\beta}(T_k) = \int_{T_k} \underline{\boldsymbol{\epsilon}}^t(\tilde{\boldsymbol{\psi}}_\alpha)\mathbb{C}\underline{\boldsymbol{\epsilon}}(\tilde{\boldsymbol{\psi}}_\beta)dT,\quad \forall(\alpha,\beta)\in\{1,\dots,6\}^2.$$

Denoting, as in Section 6.3,

$$\boldsymbol{u}^k = \mathrm{q}^{\mathrm{me}(2,k)} - \mathrm{q}^{\mathrm{me}(3,k)},\quad \boldsymbol{v}^k = \mathrm{q}^{\mathrm{me}(3,k)} - \mathrm{q}^{\mathrm{me}(1,k)},\quad \text{and}\quad \boldsymbol{w}^k = \mathrm{q}^{\mathrm{me}(1,k)} - \mathrm{q}^{\mathrm{me}(2,k)},$$

with $\mathrm{q}^{\mathrm{me}(\alpha,k)}$, $1 \leqslant \alpha \leqslant 3$, the three vertices of T_k, then the gradients of the local functions $\tilde{\varphi}_\alpha^k = \varphi_{\mathrm{me}(\alpha,k)|_{T_k}}$, $1 \leqslant \alpha \leqslant 3$, associated to T_k, are constants and given respectively by

$$\nabla\tilde{\varphi}_1^k = \frac{1}{2|T_k|}\begin{pmatrix} u_2^k \\ -u_1^k \end{pmatrix},\ \nabla\tilde{\varphi}_2^k = \frac{1}{2|T_k|}\begin{pmatrix} v_2^k \\ -v_1^k \end{pmatrix},\ \nabla\tilde{\varphi}_3^k = \frac{1}{2|T_k|}\begin{pmatrix} w_2^k \\ -w_1^k \end{pmatrix}. \tag{7.1}$$

So, we can rewrite the matrix $\mathbb{K}^e(T_k)$ in the form

$$\mathbb{K}^e(T_k) = |T_k|\mathbb{B}_k^{\mathtt{t}}\mathbb{C}\mathbb{B}_k,$$

where

$$\mathbb{B}_k = \begin{pmatrix} \underline{\boldsymbol{\epsilon}}(\tilde{\boldsymbol{\psi}}_1) & | & \dots & | & \underline{\boldsymbol{\epsilon}}(\tilde{\boldsymbol{\psi}}_6) \end{pmatrix} = \frac{1}{2|T_k|}\begin{pmatrix} u_2^k & 0 & v_2^k & 0 & w_2^k & 0 \\ 0 & -u_1^k & 0 & -v_1^k & 0 & -w_1^k \\ -u_1^k & u_2^k & -v_1^k & v_2^k & -w_1^k & w_2^k \end{pmatrix}.$$

We give the Matlab/Octave code for computing $\mathbb{K}^e(T_k)$:

Listing 9
Element matrix code (elastic stiffness matrix)

```
function Ke=ElemStiffElasMatP1(qm,area,C)
% qm=[q1,q2,q3]
u=qm(:,2)-qm(:,3);
v=qm(:,3)-qm(:,1);
w=qm(:,1)-qm(:,2);
B=[u(2),0,v(2),0,w(2),0; ...
    0,-u(1),0,-v(1),0,-w(1); ...
    -u(1),u(2),-v(1),v(2),-w(1),w(2)];
Ke=B'*C*B/(4*area);
```

Then, the classical matrix assembly code using the element matrix $\mathbb{K}^e(T_k)$ with a loop through the triangles is

Listing 10
Classical matrix assembly code (elastic stiffness matrix)

```
function K=StiffElasAssemblingP1(nq,nme,q,me,areas,lam,mu)
K=sparse(2*nq,2*nq);
C=[lam+2*mu,lam,0;lam,lam+2*mu,0;0,0,mu];
for k=1:nme
    MatElem=ElemStiffElasMatP1(q(:,me(:,k)),areas(k),C);
    I=[2*me(1,k)-1, 2*me(1,k), 2*me(2,k)-1, ...
       2*me(2,k), 2*me(3,k)-1, 2*me(3,k)];
    for il=1:6
        for jl=1:6
            K(I(il),I(jl))=K(I(il),I(jl))+MatElem(il,jl);
        end
    end
end
```

On Figure 7.1 on the left, we show the computation times (in seconds) versus the matrix size $\mathrm{n_{df}} = 2\mathrm{n_q}$, for the classical matrix assembly code and the FreeFEM++ code given in Listing 12. We observe that the complexity is $\mathcal{O}(\mathrm{n_{df}^2})$ for the Matlab/Octave codes, while the complexity seems to be $\mathcal{O}(\mathrm{n_{df}})$ for FreeFEM++.

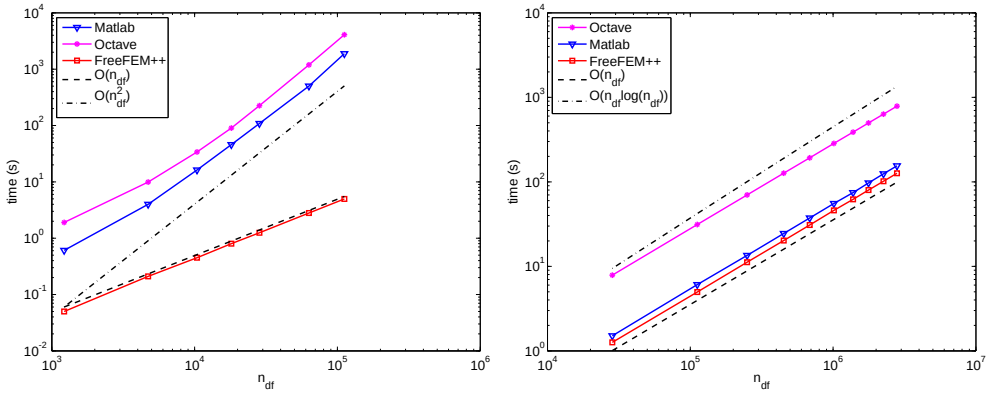


FIG. 7.1. *Comparison of the matrix assembly codes : usual assembly (left) and* `OptV1` *(right) in Matlab/Octave and FreeFEM++, for stiffness elasticity matrix.*

7.1. Optimized matrix assembly - version 1 (OptV1). We define the three local arrays $\boldsymbol{K}_k^e$, $\boldsymbol{I}_k^e$ and $\boldsymbol{J}_k^e$ of 36 elements by

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

Using the definition of $\mathcal{I}_k$ in the introduction of Section 7, we have

$$\forall(\alpha,\beta)\in\{1,\dots,6\},\ \begin{cases} \boldsymbol{K}_k^e\big(6(\beta-1)+\alpha\big) &=& \mathbb{K}^e_{\alpha,\beta}(T_k),\\ \boldsymbol{I}_k^e\big(6(\beta-1)+\alpha\big) &=& \mathcal{I}_k(\alpha),\\ \boldsymbol{J}_k^e\big(6(\beta-1)+\alpha\big) &=& \mathcal{I}_k(\beta). \end{cases}$$

Thus, from the matrix $\mathbb{K}^e(T_k)=(K^k_{i,j})_{1\leqslant i,j\leqslant 6}$, we obtain

$$\begin{array}{rcl} \boldsymbol{K}_k^e &=& \big(K^k_{1,1} \ \dots\ K^k_{6,1}\ ,\ K^k_{1,2}\ \dots\ K^k_{6,2}\ ,\ \dots\ ,\ K^k_{1,6}\ \dots\ K^k_{6,6}\big)\\ \boldsymbol{I}_k^e &=& \big(\mathcal{I}_k(1) \ \dots\ \mathcal{I}_k(6)\ ,\ \mathcal{I}_k(1)\ \dots\ \mathcal{I}_k(6)\ ,\ \dots\ ,\ \mathcal{I}_k(1)\ \dots\ \mathcal{I}_k(6)\big)\\ \boldsymbol{J}_k^e &=& \big(\mathcal{I}_k(1) \ \dots\ \mathcal{I}_k(1)\ ,\ \mathcal{I}_k(2)\ \dots\ \mathcal{I}_k(2)\ ,\ \dots\ ,\ \mathcal{I}_k(6)\ \dots\ \mathcal{I}_k(6)\big) \end{array}$$

We give below the associated Matlab/Octave code :

LISTING 11
Optimized matrix assembly code - version 1 (elastic stiffness matrix)

```
function K=StiffElasAssemblingP1OptV1(nq,nme,q,me,areas,lam,mu)
Ig=zeros(36*Th.nme,1); Jg=zeros(36*Th.nme,1);
Kg=zeros(36*Th.nme,1);
kk=1:36;
C=[lam+2*mu,lam,0;lam,lam+2*mu,0;0,0,mu];
for k=1:nme
  Me=ElemStiffElasMatP1(q(:,me(:,k)),areas(k),C);
  I=[2*me(1,k)-1, 2*me(1,k), 2*me(2,k)-1, ...
     2*me(2,k), 2*me(3,k)-1, 2*me(3,k)];
  je=ones(6,1)*I; ie=je';
  Ig(kk)=ie(:); Jg(kk)=je(:);
  Kg(kk)=Me(:);
  kk=kk+36;
end
K=sparse(Ig,Jg,Kg,2*nq,2*nq);
```

On Figure 7.1 on the right, we show the computation times of the `OptV1` codes in Matlab/Octave and of the FreeFEM++ codes versus the number of degrees of freedom on the mesh (unit disk). The complexity of the Matlab/Octave codes seems now linear (i.e. $\mathcal{O}(\mathrm{n_{df}})$) as for FreeFEM++. Also, FreeFEM++ is slightly faster than Matlab, while much more faster than Octave (about a factor 10).

LISTING 12
Matrix assembly code in FreeFEM++ (elastic stiffness matrix)

```
mesh Th(...);
fespace Wh(Th,[P1,P1]);
Wh [u1,u2],[v1,v2];
real lam=...,mu=...;
func C=[[lam+2*mu,lam,0],[lam,lam+2*mu,0],[0,0,mu]];
macro epsilon(ux,uy) [dx(ux),dy(uy),(dy(ux)+dx(uy))]
macro sigma(ux,uy) ( C*epsilon(ux,uy) )
varf vStiffElas([u1,u2],[v1,v2])=
                    int2d(Th)(epsilon(u1,u2)'*sigma(v1,v2));
matrix K = vStiffElas(Wh,Wh);
```

To further improve the efficiency of the matrix assembly code, we introduce now the optimized version 2.

7.2. Optimized matrix assembly - version 2 (OptV2). In this version, no loop is used. As in Section 6, we define three 2d-arrays that allow to store all the element matrices as well as their positions in the global matrix. We denote by $\mathbb{K}_g$, $\mathbb{I}_g$ and $\mathbb{J}_g$ these 36-by-$\mathrm{n_{me}}$ arrays, defined $\forall k \in \{1,\dots,\mathrm{n_{me}}\}$, $\forall il \in \{1,\dots,36\}$ by

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

Thus, the local arrays $\boldsymbol{K}_k^e$, $\boldsymbol{I}_k^e$ and $\boldsymbol{J}_k^e$ are stored in the k-th column of the global arrays $\mathbb{K}_g$, $\mathbb{I}_g$ and $\mathbb{J}_g$ respectively. Once these arrays are determined, the assembly matrix is obtained with the Matlab/Octave command

M = **sparse**(Ig(:),Jg(:),Kg(:),2*nq,2*nq);

In order to vectorize the computation of $\mathbb{I}_g$ and $\mathbb{J}_g$, we generalize the technique introduced in the Remark 6.1 and denote by $\mathbb{T}$ the $6 \times \mathrm{n_{me}}$ array defined by

$$\mathbb{T} = \begin{pmatrix} \mathcal{I}_1(1) & \dots & \mathcal{I}_k(1) & \dots & \mathcal{I}_{\mathrm{n_{me}}}(1) \\ \mathcal{I}_1(2) & \dots & \mathcal{I}_k(2) & \dots & \mathcal{I}_{\mathrm{n_{me}}}(2) \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \mathcal{I}_1(6) & \dots & \mathcal{I}_k(6) & \dots & \mathcal{I}_{\mathrm{n_{me}}}(6) \end{pmatrix}.$$

Then $\mathbb{I}_g$ is computed by duplicating $\mathbb{T}$ six times, column-wise. The array $\mathbb{J}_g$ is computed from $\mathbb{T}$ by duplicating each line, six times, successively. We give in Listing 13, the Matlab/Octave vectorized function which enables to compute $\mathbb{I}_g$ and $\mathbb{J}_g$.

It remains to vectorize the computation of the 2d-array $\mathbb{K}_g$. Using formulas (7.1), for $1 \leqslant \alpha \leqslant 3$, we define the 2-by-$\mathrm{n_{me}}$ array $\boldsymbol{G}_\alpha$, the k-th column of which contains $\nabla\tilde{\varphi}_\alpha^k$.

Listing 13
Vectorized code for computing $\mathbb{I}_g$ and $\mathbb{J}_g$

```
function [Ig,Jg]=BuildIgJgP1VF(me)
T= [2*me(1,:)-1; 2*me(1,:); ...
    2*me(2,:)-1; 2*me(2,:); ...
    2*me(3,:)-1; 2*me(3,:)];

ii=[1 1 1 1 1 1; ...
    2 2 2 2 2 2; ...
    3 3 3 3 3 3; ...
    4 4 4 4 4 4; ...
    5 5 5 5 5 5; ...
    6 6 6 6 6 6];

jj=ii';

Ig=T(ii(:),:);
Jg=T(jj(:),:);
```

Let us focus on the first column of $\mathbb{K}^e(T_k)$. It is given by

$$\mathbb{K}^e_{1,1}(T_k) = |T_k|\left((\lambda+2\,\mu)\frac{\partial\tilde{\varphi}_1}{\partial x}^2+\mu\frac{\partial\tilde{\varphi}_1}{\partial y}^2\right),\qquad \mathbb{K}^e_{2,1}(T_k) = |T_k|\left(\lambda\frac{\partial\tilde{\varphi}_1}{\partial x}\frac{\partial\tilde{\varphi}_1}{\partial y}+\mu\frac{\partial\tilde{\varphi}_1}{\partial x}\frac{\partial\tilde{\varphi}_1}{\partial y}\right)$$
$$\mathbb{K}^e_{3,1}(T_k) = |T_k|\left((\lambda+2\,\mu)\frac{\partial\tilde{\varphi}_1}{\partial x}\frac{\partial\tilde{\varphi}_2}{\partial x}+\mu\frac{\partial\tilde{\varphi}_1}{\partial y}\frac{\partial\tilde{\varphi}_2}{\partial y}\right),\mathbb{K}^e_{4,1}(T_k) = |T_k|\left(\lambda\frac{\partial\tilde{\varphi}_1}{\partial x}\frac{\partial\tilde{\varphi}_2}{\partial y}+\mu\frac{\partial\tilde{\varphi}_1}{\partial y}\frac{\partial\tilde{\varphi}_2}{\partial x}\right)$$
$$\mathbb{K}^e_{5,1}(T_k) = |T_k|\left((\lambda+2\,\mu)\frac{\partial\tilde{\varphi}_1}{\partial x}\frac{\partial\tilde{\varphi}_3}{\partial x}+\mu\frac{\partial\tilde{\varphi}_1}{\partial y}\frac{\partial\tilde{\varphi}_3}{\partial y}\right),\mathbb{K}^e_{6,1}(T_k) = |T_k|\left(\lambda\frac{\partial\tilde{\varphi}_1}{\partial x}\frac{\partial\tilde{\varphi}_3}{\partial y}+\mu\frac{\partial\tilde{\varphi}_1}{\partial y}\frac{\partial\tilde{\varphi}_3}{\partial x}\right)$$

This gives, on the triangle T_k,

$$\begin{aligned}
\mathbb{K}^e_{1,1}(T_k) &= |T_k|\left((\lambda+2\,\mu)G_1(1,k)^2+\mu G_1(2,k)^2\right)\\
\mathbb{K}^e_{2,1}(T_k) &= |T_k|\left(\lambda G_1(1,k)G_1(2,k)+\mu G_1(1,k)G_1(2,k)\right)\\
\mathbb{K}^e_{3,1}(T_k) &= |T_k|\left((\lambda+2\,\mu)G_1(1,k)G_2(1,k)+\mu G_1(2,k)G_2(2,k)\right)\\
\mathbb{K}^e_{4,1}(T_k) &= |T_k|\left(\lambda G_1(1,k)G_2(2,k)+\mu G_1(2,k)G_2(1,k)\right)\\
\mathbb{K}^e_{5,1}(T_k) &= |T_k|\left((\lambda+2\,\mu)G_1(1,k)G_3(1,k)+\mu G_1(2,k)G_3(2,k)\right)\\
\mathbb{K}^e_{6,1}(T_k) &= |T_k|\left(\lambda G_1(1,k)G_3(2,k)+\mu G_1(2,k)G_3(1,k)\right)
\end{aligned}$$

Thus, the computation of the first six lines of $\mathbb{K}_g$ may be vectorized under the form:

```
Kg(1,:)=((lam+2*mu)*G1(1,:).^2 + mu*G1(2,:).^2).*area;
Kg(2,:)=(lam*G1(1,:).*G1(2,:) + mu*G1(1,:).*G1(2,:)).*area;
Kg(3,:)=((lam+2*mu)*G1(1,:).*G2(1,:) + mu*G1(2,:).*G2(2,:)).*area;
Kg(4,:)=(lam*G1(1,:).*G2(2,:) + mu*G1(2,:).*G2(1,:)).*area;
Kg(5,:)=((lam+2*mu)*G1(1,:).*G3(1,:) + mu*G1(2,:).*G3(2,:)).*area;
Kg(6,:)=(lam*G1(1,:).*G3(2,:) + mu*G1(2,:).*G3(1,:)).*area;
```

The other columns of $\mathbb{K}_g$ are computed on the same principle, using the symmetry of the matrix. We give in the Listings 14 and 15 the complete vectorized Matlab/Octave functions for computing $\mathbb{K}_g$ and the elastic stiffness matrix assembly respectively.

LISTING 14
Vectorized code for computing $\mathbb{K}_g$

```
function [Kg]=ElemStiffElasMatVecP1(q,me,areas,lam,mu)
u=q(:,me(2,:))-q(:,me(3,:));   % q2-q3
G1=[u(2,:);-u(1,:)];
u=q(:,me(3,:))-q(:,me(1,:));   % q3-q1
G2=[u(2,:);-u(1,:)];
u=q(:,me(1,:))-q(:,me(2,:));   % q1-q2
G3=[u(2,:);-u(1,:)];
clear u
coef=ones(2,1)*(0.5./sqrt(areas));
G1=G1.*coef;
G2=G2.*coef;
G3=G3.*coef;
clear coef
Kg=zeros(36,size(me,2));
Kg(1,:)=(lam + 2*mu)*G1(1,:).^2 + mu*G1(2,:).^2;
Kg(2,:)=lam.*G1(1,:).*G1(2,:) + mu*G1(1,:).*G1(2,:);
Kg(3,:)=(lam + 2*mu)*G1(1,:).*G2(1,:) + mu*G1(2,:).*G2(2,:);
Kg(4,:)=lam.*G1(1,:).*G2(2,:) + mu*G1(2,:).*G2(1,:);
Kg(5,:)=(lam + 2*mu)*G1(1,:).*G3(1,:) + mu*G1(2,:).*G3(2,:);
Kg(6,:)=lam.*G1(1,:).*G3(2,:) + mu*G1(2,:).*G3(1,:);
Kg(8,:)=(lam + 2*mu)*G1(2,:).^2 + mu*G1(1,:).^2;
Kg(9,:)=lam.*G1(2,:).*G2(1,:) + mu*G1(1,:).*G2(2,:);
Kg(10,:)=(lam + 2*mu)*G1(2,:).*G2(2,:) + mu*G1(1,:).*G2(1,:);
Kg(11,:)=lam.*G1(2,:).*G3(1,:) + mu*G1(1,:).*G3(2,:);
Kg(12,:)=(lam + 2*mu)*G1(2,:).*G3(2,:) + mu*G1(1,:).*G3(1,:);
Kg(15,:)=(lam + 2*mu)*G2(1,:).^2 + mu*G2(2,:).^2;
Kg(16,:)=lam.*G2(1,:).*G2(2,:) + mu*G2(1,:).*G2(2,:);
Kg(17,:)=(lam + 2*mu)*G2(1,:).*G3(1,:) + mu*G2(2,:).*G3(2,:);
Kg(18,:)=lam.*G2(1,:).*G3(2,:) + mu*G2(2,:).*G3(1,:);
Kg(22,:)=(lam + 2*mu)*G2(2,:).^2 + mu*G2(1,:).^2;
Kg(23,:)=lam.*G2(2,:).*G3(1,:) + mu*G2(1,:).*G3(2,:);
Kg(24,:)=(lam + 2*mu)*G2(2,:).*G3(2,:) + mu*G2(1,:).*G3(1,:);
Kg(29,:)=(lam + 2*mu)*G3(1,:).^2 + mu*G3(2,:).^2;
Kg(30,:)=lam.*G3(1,:).*G3(2,:) + mu*G3(1,:).*G3(2,:);
Kg(36,:)=(lam + 2*mu)*G3(2,:).^2 + mu*G3(1,:).^2;
Kg([7,13,14,19,20,21,25,26,27,28,31,32,33,34,35],:)= ...
  Kg([2,3,9,4,10,16,5,11,17,23,6,12,18,24,30],:);
```

LISTING 15
Optimized matrix assembly code - version 2 (elastic stiffness matrix)

```
function [K]=StiffElasAssemblingP1OptV2(nq,nme,q,me,areas,lam,mu)
[Ig,Jg]=BuildIgJgP1VF(me);
Kg=ElemStiffElasMatVecP1(q,me,areas,lam,mu);
K = sparse(Ig(:),Jg(:),Kg(:),2*nq,2*nq);
```

On Figure 7.2, we show the computation times of the `OptV2` (in Matlab/Octave) and FreeFEM++ codes, versus the number of degrees of freedom on the mesh. The computation times values are given in Table 7.1. The complexity of the Matlab/Octave codes is still linear ($\mathcal{O}(\mathrm{n_{df}})$). Moreover, the computation times are 10 (resp. 5) times faster with Octave (resp. Matlab) than those obtained with FreeFEM++.

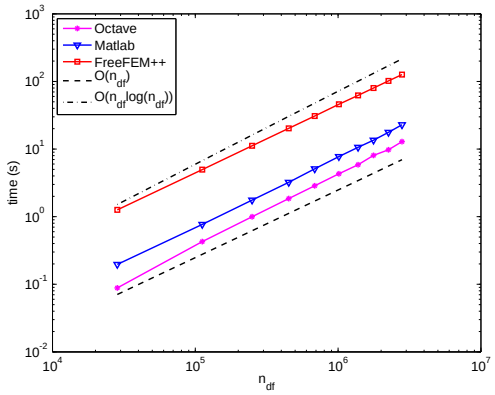


FIG. 7.2. *Comparison of the assembly code :* `OptV2` *in Matlab/Octave and FreeFEM++, for elastic stiffness matrix.*

n_q	n_{df}	Octave (3.6.3)	Matlab (R2012b)	FreeFEM++ (3.20)
14222	28444	0.088 (s) x 1.00	0.197 (s) x 0.45	1.260 (s) x 0.07
55919	111838	0.428 (s) x 1.00	0.769 (s) x 0.56	4.970 (s) x 0.09
125010	250020	0.997 (s) x 1.00	1.757 (s) x 0.57	11.190 (s) x 0.09
225547	451094	1.849 (s) x 1.00	3.221 (s) x 0.57	20.230 (s) x 0.09
343082	686164	2.862 (s) x 1.00	5.102 (s) x 0.56	30.840 (s) x 0.09
506706	1013412	4.304 (s) x 1.00	7.728 (s) x 0.56	45.930 (s) x 0.09
689716	1379432	5.865 (s) x 1.00	10.619 (s) x 0.55	62.170 (s) x 0.09
885521	1771042	8.059 (s) x 1.00	13.541 (s) x 0.60	79.910 (s) x 0.10
1127090	2254180	9.764 (s) x 1.00	17.656 (s) x 0.55	101.730 (s) x 0.10
1401129	2802258	12.893 (s) x 1.00	22.862 (s) x 0.56	126.470 (s) x 0.10

TABLE 7.1

Computational cost of the `StiffElas` *matrix assembly versus* n_q/n_{df}*, with the* `OptV2` *Matlab/Octave codes (columns 3,4) and with FreeFEM++ (column 5) : time in seconds (top value) and speedup (bottom value). The speedup reference is* `OptV2` *Octave version.*

As observed in Remark 6.2, Octave gives better results than Matlab only for the `OptV2` codes. Using `cs_sparse` function instead of Matlab `sparse` function is approximately 1.1 (resp. 2.5) times faster for `OptV1` (resp. `OptV2`) version as shown on Figure 7.3.

8. Conclusion. For several examples of matrices, from the classical code we have built step by step the codes to perform the assembly of these matrices to obtain a fully vectorized form. For each version, we have described the algorithm and estimated its numerical complexity. The assembly of the mass, weighted mass and stiffness matrices of size 10^6, on our reference computer, is obtained in less than 4 seconds (resp. about 2 seconds) with Matlab (resp. with Octave). The assembly of the elastic stiffness matrix of size 10^6, is computed in less than 8 seconds (resp. about 4 seconds) with Matlab (resp. with Octave).

These optimization techniques in Matlab/Octave may be extended to other types of matrices, for higher order or others finite elements (P_k, Q_k, ...) and in 3D.

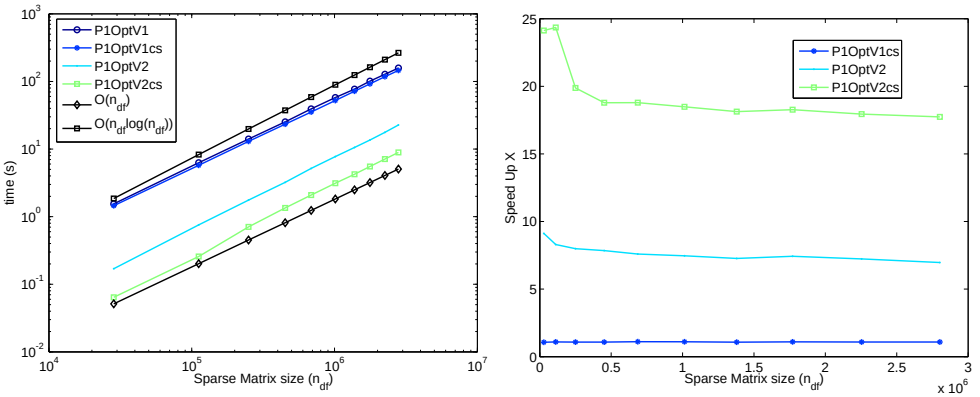


FIG. 7.3. *Computational cost of the StiffElasAssembling functions versus* n_{df}*, with Matlab (R2012b) : time in seconds (left) and speedup (right). The speedup reference is* `OptV1` *version.*

In Matlab, it is possible to further improve the performances of the `OptV2` codes by using Nvidia GPU cards. Preliminary Matlab results give a computation time divided by a factor 5 on a Nvidia `GTX 590` GPU card (compared to the `OptV2` without GPU).

Appendix A. Comparison of the performances with FreeFEM++.

A.1. Classical matrix assembly code vs FreeFEM++.

n_q	Matlab (R2012b)	Octave (3.6.3)	FreeFEM++ (3.2)
3576	1.242 (s) x 1.00	3.131 (s) x 0.40	0.020 (s) x 62.09
14222	10.875 (s) x 1.00	24.476 (s) x 0.44	0.050 (s) x 217.49
31575	44.259 (s) x 1.00	97.190 (s) x 0.46	0.120 (s) x 368.82
55919	129.188 (s) x 1.00	297.360 (s) x 0.43	0.210 (s) x 615.18
86488	305.606 (s) x 1.00	711.407 (s) x 0.43	0.340 (s) x 898.84
125010	693.431 (s) x 1.00	1924.729 (s) x 0.36	0.480 (s) x 1444.65
170355	1313.800 (s) x 1.00	3553.827 (s) x 0.37	0.670 (s) x 1960.89
225547	3071.727 (s) x 1.00	5612.940 (s) x 0.55	0.880 (s) x 3490.60
281769	3655.551 (s) x 1.00	8396.219 (s) x 0.44	1.130 (s) x 3235.00
343082	5701.736 (s) x 1.00	12542.198 (s) x 0.45	1.360 (s) x 4192.45
424178	8162.677 (s) x 1.00	20096.736 (s) x 0.41	1.700 (s) x 4801.57

TABLE A.1

Computational cost of the `Mass` *matrix assembly versus* n_q*, with the* `basic` *Matlab/Octave version (columns* $2, 3$*) and with FreeFEM++ (column* 4*) : time in seconds (top value) and speedup (bottom value). The speedup reference is* `basic` *Matlab version.*

n_q	Matlab (R2012b)	Octave (3.6.3)	FreeFEM++ (3.2)
3576	1.333 (s) x 1.00	3.988 (s) x 0.33	0.020 (s) x 66.64
14222	11.341 (s) x 1.00	27.156 (s) x 0.42	0.080 (s) x 141.76
31575	47.831 (s) x 1.00	108.659 (s) x 0.44	0.170 (s) x 281.36
55919	144.649 (s) x 1.00	312.947 (s) x 0.46	0.300 (s) x 482.16
86488	341.704 (s) x 1.00	739.720 (s) x 0.46	0.460 (s) x 742.84
125010	715.268 (s) x 1.00	1591.508 (s) x 0.45	0.680 (s) x 1051.86
170355	1480.894 (s) x 1.00	2980.546 (s) x 0.50	0.930 (s) x 1592.36
225547	3349.900 (s) x 1.00	5392.549 (s) x 0.62	1.220 (s) x 2745.82
281769	4022.335 (s) x 1.00	10827.269 (s) x 0.37	1.550 (s) x 2595.05
343082	5901.041 (s) x 1.00	14973.076 (s) x 0.39	1.890 (s) x 3122.24
424178	8342.178 (s) x 1.00	22542.074 (s) x 0.37	2.340 (s) x 3565.03

TABLE A.2

Computational cost of the **`MassW`** *matrix assembly versus* $\mathrm{n_q}$*, with the* **`basic`** *Matlab/Octave version (columns* $2, 3$*) and with FreeFEM++ (column* 4*) : time in seconds (top value) and speedup (bottom value). The speedup reference is* **`basic`** *Matlab version.*

n_q	Matlab (R2012b)	Octave (3.6.3)	FreeFEM++ (3.2)
3576	1.508 (s) x 1.00	3.464 (s) x 0.44	0.020 (s) x 75.40
14222	12.294 (s) x 1.00	23.518 (s) x 0.52	0.090 (s) x 136.60
31575	47.791 (s) x 1.00	97.909 (s) x 0.49	0.210 (s) x 227.58
55919	135.202 (s) x 1.00	308.382 (s) x 0.44	0.370 (s) x 365.41
86488	314.966 (s) x 1.00	736.435 (s) x 0.43	0.570 (s) x 552.57
125010	812.572 (s) x 1.00	1594.866 (s) x 0.51	0.840 (s) x 967.35
170355	1342.657 (s) x 1.00	3015.801 (s) x 0.45	1.130 (s) x 1188.19
225547	3268.987 (s) x 1.00	5382.398 (s) x 0.61	1.510 (s) x 2164.89
281769	3797.105 (s) x 1.00	8455.267 (s) x 0.45	1.910 (s) x 1988.01
343082	6085.713 (s) x 1.00	12558.432 (s) x 0.48	2.310 (s) x 2634.51
424178	8462.518 (s) x 1.00	19274.656 (s) x 0.44	2.860 (s) x 2958.92

TABLE A.3

Computational cost of the **`Stiff`** *matrix assembly versus* $\mathrm{n_q}$*, with the* **`basic`** *Matlab/Octave version (columns* $2, 3$*) and with FreeFEM++ (column* 4*) : time in seconds (top value) and speedup (bottom value). The speedup reference is* **`basic`** *Matlab version.*

A.2. OptV0 matrix assembly code vs FreeFEM++.

n_q	Matlab (R2012b)	Octave (3.6.3)	FreeFEM++ (3.2)
3576	0.533 (s) x 1.00	1.988 (s) x 0.27	0.020 (s) x 26.67
14222	5.634 (s) x 1.00	24.027 (s) x 0.23	0.050 (s) x 112.69
31575	29.042 (s) x 1.00	106.957 (s) x 0.27	0.120 (s) x 242.02
55919	101.046 (s) x 1.00	315.618 (s) x 0.32	0.210 (s) x 481.17
86488	250.771 (s) x 1.00	749.639 (s) x 0.33	0.340 (s) x 737.56
125010	562.307 (s) x 1.00	1582.636 (s) x 0.36	0.480 (s) x 1171.47
170355	1120.008 (s) x 1.00	2895.512 (s) x 0.39	0.670 (s) x 1671.65
225547	2074.929 (s) x 1.00	4884.057 (s) x 0.42	0.880 (s) x 2357.87
281769	3054.103 (s) x 1.00	7827.873 (s) x 0.39	1.130 (s) x 2702.75
343082	4459.816 (s) x 1.00	11318.536 (s) x 0.39	1.360 (s) x 3279.28
424178	7638.798 (s) x 1.00	17689.047 (s) x 0.43	1.700 (s) x 4493.41

TABLE A.4

Computational cost of the `Mass` *matrix assembly versus* n_q, *with the* `OptV0` *Matlab/Octave version (columns* 2, 3*) and with FreeFEM++ (column* 4*) : time in seconds (top value) and speedup (bottom value). The speedup reference is* `OptV0` *Matlab version.*

n_q	Matlab (R2012b)	Octave (3.6.3)	FreeFEM++ (3.2)
3576	0.638 (s) x 1.00	3.248 (s) x 0.20	0.020 (s) x 31.89
14222	6.447 (s) x 1.00	27.560 (s) x 0.23	0.080 (s) x 80.58
31575	36.182 (s) x 1.00	114.969 (s) x 0.31	0.170 (s) x 212.83
55919	125.339 (s) x 1.00	320.114 (s) x 0.39	0.300 (s) x 417.80
86488	339.268 (s) x 1.00	771.449 (s) x 0.44	0.460 (s) x 737.54
125010	584.245 (s) x 1.00	1552.844 (s) x 0.38	0.680 (s) x 859.18
170355	1304.881 (s) x 1.00	2915.124 (s) x 0.45	0.930 (s) x 1403.10
225547	2394.946 (s) x 1.00	4934.726 (s) x 0.49	1.220 (s) x 1963.07
281769	3620.519 (s) x 1.00	8230.834 (s) x 0.44	1.550 (s) x 2335.82
343082	5111.303 (s) x 1.00	11788.945 (s) x 0.43	1.890 (s) x 2704.39
424178	8352.331 (s) x 1.00	18289.219 (s) x 0.46	2.340 (s) x 3569.37

TABLE A.5

Computational cost of the `MassW` *matrix assembly versus* n_q, *with the* `OptV0` *Matlab/Octave version (columns* 2, 3*) and with FreeFEM++ (column* 4*) : time in seconds (top value) and speedup (bottom value). The speedup reference is* `OptV0` *Matlab version.*

n_q	Matlab (R2012b)	Octave (3.6.3)	FreeFEM++ (3.2)
3576	0.738 (s) x 1.00	2.187 (s) x 0.34	0.020 (s) x 36.88
14222	6.864 (s) x 1.00	23.037 (s) x 0.30	0.090 (s) x 76.26
31575	32.143 (s) x 1.00	101.787 (s) x 0.32	0.210 (s) x 153.06
55919	99.828 (s) x 1.00	306.232 (s) x 0.33	0.370 (s) x 269.81
86488	259.689 (s) x 1.00	738.838 (s) x 0.35	0.570 (s) x 455.59
125010	737.888 (s) x 1.00	1529.401 (s) x 0.48	0.840 (s) x 878.44
170355	1166.721 (s) x 1.00	2878.325 (s) x 0.41	1.130 (s) x 1032.50
225547	2107.213 (s) x 1.00	4871.663 (s) x 0.43	1.510 (s) x 1395.51
281769	3485.933 (s) x 1.00	7749.715 (s) x 0.45	1.910 (s) x 1825.10
343082	5703.957 (s) x 1.00	11464.992 (s) x 0.50	2.310 (s) x 2469.25
424178	8774.701 (s) x 1.00	17356.351 (s) x 0.51	2.860 (s) x 3068.08

TABLE A.6

Computational cost of the `Stiff` *matrix assembly versus* n_q, *with the* `OptV0` *Matlab/Octave version (columns* $2, 3$*) and with FreeFEM++ (column* 4*) : time in seconds (top value) and speedup (bottom value). The speedup reference is OptV0 Matlab version.*

A.3. OptV1 matrix assembly code vs FreeFEM++.

n_q	Matlab (R2012b)	Octave (3.6.3)	FreeFEM++ (3.20)
14222	0.416 (s) x 1.00	2.022 (s) x 0.21	0.060 (s) x 6.93
55919	1.117 (s) x 1.00	8.090 (s) x 0.14	0.200 (s) x 5.58
125010	2.522 (s) x 1.00	18.217 (s) x 0.14	0.490 (s) x 5.15
225547	4.524 (s) x 1.00	32.927 (s) x 0.14	0.890 (s) x 5.08
343082	7.105 (s) x 1.00	49.915 (s) x 0.14	1.370 (s) x 5.19
506706	10.445 (s) x 1.00	73.487 (s) x 0.14	2.000 (s) x 5.22
689716	14.629 (s) x 1.00	99.967 (s) x 0.15	2.740 (s) x 5.34
885521	18.835 (s) x 1.00	128.529 (s) x 0.15	3.550 (s) x 5.31
1127090	23.736 (s) x 1.00	163.764 (s) x 0.14	4.550 (s) x 5.22
1401129	29.036 (s) x 1.00	202.758 (s) x 0.14	5.680 (s) x 5.11
1671052	35.407 (s) x 1.00	242.125 (s) x 0.15	6.810 (s) x 5.20
1978602	41.721 (s) x 1.00	286.568 (s) x 0.15	8.070 (s) x 5.17

TABLE A.7

Computational cost of the `Mass` *matrix assembly versus* n_q, *with the* `OptV1` *Matlab/Octave version (columns* $2, 3$*) and with FreeFEM++ (column* 4*) : time in seconds (top value) and speedup (bottom value). The speedup reference is* `OptV1` *Matlab version.*

n_q	Matlab (R2012b)	Octave (3.6.3)	FreeFEM++ (3.20)
14222	0.680 (s) x 1.00	4.633 (s) x 0.15	0.070 (s) x 9.71
55919	2.013 (s) x 1.00	18.491 (s) x 0.11	0.310 (s) x 6.49
125010	4.555 (s) x 1.00	41.485 (s) x 0.11	0.680 (s) x 6.70
225547	8.147 (s) x 1.00	74.632 (s) x 0.11	1.240 (s) x 6.57
343082	12.462 (s) x 1.00	113.486 (s) x 0.11	1.900 (s) x 6.56
506706	18.962 (s) x 1.00	167.979 (s) x 0.11	2.810 (s) x 6.75
689716	25.640 (s) x 1.00	228.608 (s) x 0.11	3.870 (s) x 6.63
885521	32.574 (s) x 1.00	292.502 (s) x 0.11	4.950 (s) x 6.58
1127090	42.581 (s) x 1.00	372.115 (s) x 0.11	6.340 (s) x 6.72
1401129	53.395 (s) x 1.00	467.396 (s) x 0.11	7.890 (s) x 6.77
1671052	61.703 (s) x 1.00	554.376 (s) x 0.11	9.480 (s) x 6.51
1978602	77.085 (s) x 1.00	656.220 (s) x 0.12	11.230 (s) x 6.86

TABLE A.8

Computational cost of the `MassW` *matrix assembly versus* n_q, *with the* `OptV1` *Matlab/Octave version (columns* $2, 3$*) and with FreeFEM++ (column* 4*) : time in seconds (top value) and speedup (bottom value). The speedup reference is* `OptV1` *Matlab version.*

n_q	Matlab (R2012b)	Octave (3.6.3)	FreeFEM++ (3.20)
14222	1.490 (s) x 1.00	3.292 (s) x 0.45	0.090 (s) x 16.55
55919	4.846 (s) x 1.00	13.307 (s) x 0.36	0.360 (s) x 13.46
125010	10.765 (s) x 1.00	30.296 (s) x 0.36	0.830 (s) x 12.97
225547	19.206 (s) x 1.00	54.045 (s) x 0.36	1.500 (s) x 12.80
343082	28.760 (s) x 1.00	81.988 (s) x 0.35	2.290 (s) x 12.56
506706	42.309 (s) x 1.00	121.058 (s) x 0.35	3.390 (s) x 12.48
689716	57.635 (s) x 1.00	164.955 (s) x 0.35	4.710 (s) x 12.24
885521	73.819 (s) x 1.00	211.515 (s) x 0.35	5.960 (s) x 12.39
1127090	94.438 (s) x 1.00	269.490 (s) x 0.35	7.650 (s) x 12.34
1401129	117.564 (s) x 1.00	335.906 (s) x 0.35	9.490 (s) x 12.39
1671052	142.829 (s) x 1.00	397.392 (s) x 0.36	11.460 (s) x 12.46
1978602	169.266 (s) x 1.00	471.031 (s) x 0.36	13.470 (s) x 12.57

TABLE A.9

Computational cost of the `Stiff` *matrix assembly versus* n_q, *with the* `OptV1` *Matlab/Octave version (columns* $2, 3$*) and with FreeFEM++ (column* 4*) : time in seconds (top value) and speedup (bottom value). The speedup reference is* `OptV1` *Matlab version.*

A.4. OptV2 matrix assembly code vs FreeFEM++.

n_q	Octave (3.6.3)	Matlab (R2012b)	FreeFEM++ (3.20)
125010	0.239 (s) x 1.00	0.422 (s) x 0.57	0.470 (s) x 0.51
225547	0.422 (s) x 1.00	0.793 (s) x 0.53	0.880 (s) x 0.48
343082	0.663 (s) x 1.00	1.210 (s) x 0.55	1.340 (s) x 0.49
506706	0.990 (s) x 1.00	1.876 (s) x 0.53	2.000 (s) x 0.49
689716	1.432 (s) x 1.00	2.619 (s) x 0.55	2.740 (s) x 0.52
885521	1.843 (s) x 1.00	3.296 (s) x 0.56	3.510 (s) x 0.53
1127090	2.331 (s) x 1.00	4.304 (s) x 0.54	4.520 (s) x 0.52
1401129	2.945 (s) x 1.00	5.426 (s) x 0.54	5.580 (s) x 0.53
1671052	3.555 (s) x 1.00	6.480 (s) x 0.55	6.720 (s) x 0.53
1978602	4.175 (s) x 1.00	7.889 (s) x 0.53	7.940 (s) x 0.53
2349573	5.042 (s) x 1.00	9.270 (s) x 0.54	9.450 (s) x 0.53
2732448	5.906 (s) x 1.00	10.558 (s) x 0.56	11.000 (s) x 0.54
3085628	6.640 (s) x 1.00	12.121 (s) x 0.55	12.440 (s) x 0.53

TABLE A.10

Computational cost of the `Mass` *matrix assembly versus* n_q, *with the* `OptV2` *Matlab/Octave codes (columns* $2, 3$*) and with FreeFEM++ (column* 4*) : time in seconds (top value) and speedup (bottom value). The speedup reference is* `OptV2` *Octave version.*

n_q	Octave (3.6.3)	Matlab (R2012b)	FreeFEM++ (3.20)
125010	0.214 (s) x 1.00	0.409 (s) x 0.52	0.680 (s) x 0.31
225547	0.405 (s) x 1.00	0.776 (s) x 0.52	1.210 (s) x 0.33
343082	0.636 (s) x 1.00	1.229 (s) x 0.52	1.880 (s) x 0.34
506706	0.941 (s) x 1.00	1.934 (s) x 0.49	2.770 (s) x 0.34
689716	1.307 (s) x 1.00	2.714 (s) x 0.48	4.320 (s) x 0.30
885521	1.791 (s) x 1.00	3.393 (s) x 0.53	4.880 (s) x 0.37
1127090	2.320 (s) x 1.00	4.414 (s) x 0.53	6.260 (s) x 0.37
1401129	2.951 (s) x 1.00	5.662 (s) x 0.52	7.750 (s) x 0.38
1671052	3.521 (s) x 1.00	6.692 (s) x 0.53	9.290 (s) x 0.38
1978602	4.201 (s) x 1.00	8.169 (s) x 0.51	11.000 (s) x 0.38
2349573	5.456 (s) x 1.00	9.564 (s) x 0.57	13.080 (s) x 0.42
2732448	6.178 (s) x 1.00	10.897 (s) x 0.57	15.220 (s) x 0.41
3085628	6.854 (s) x 1.00	12.535 (s) x 0.55	17.190 (s) x 0.40

TABLE A.11

Computational cost of the `MassW` *matrix assembly versus* n_q, *with the* `OptV2` *Matlab/Octave codes (columns* $2, 3$*) and with FreeFEM++ (column* 4*) : time in seconds (top value) and speedup (bottom value). The speedup reference is* `OptV2` *Octave version.*

n_q	Octave (3.6.3)	Matlab (R2012b)	FreeFEM++ (3.20)
125010	0.227 (s) x 1.00	0.453 (s) x 0.50	0.800 (s) x 0.28
225547	0.419 (s) x 1.00	0.833 (s) x 0.50	1.480 (s) x 0.28
343082	0.653 (s) x 1.00	1.323 (s) x 0.49	2.260 (s) x 0.29
506706	0.981 (s) x 1.00	1.999 (s) x 0.49	3.350 (s) x 0.29
689716	1.354 (s) x 1.00	2.830 (s) x 0.48	4.830 (s) x 0.28
885521	1.889 (s) x 1.00	3.525 (s) x 0.54	5.910 (s) x 0.32
1127090	2.385 (s) x 1.00	4.612 (s) x 0.52	7.560 (s) x 0.32
1401129	3.021 (s) x 1.00	5.810 (s) x 0.52	9.350 (s) x 0.32
1671052	3.613 (s) x 1.00	6.899 (s) x 0.52	11.230 (s) x 0.32
1978602	4.294 (s) x 1.00	8.504 (s) x 0.50	13.280 (s) x 0.32
2349573	5.205 (s) x 1.00	9.886 (s) x 0.53	16.640 (s) x 0.31
2732448	6.430 (s) x 1.00	11.269 (s) x 0.57	19.370 (s) x 0.33
3085628	7.322 (s) x 1.00	13.049 (s) x 0.56	20.800 (s) x 0.35

TABLE A.12

Computational cost of the `Stiff` *matrix assembly versus* $\mathrm{n_q}$, *with the* `OptV2` *Matlab/Octave codes (columns* 2, 3*) and with FreeFEM++ (column* 4*) : time in seconds (top value) and speedup (bottom value). The speedup reference is* `OptV2` *Octave version.*

Appendix B. Matrix assembly codes.

B.1. Element matrices.

LISTING 16
ElemMassMatP1.m

```
function AElem=ElemMassMatP1(area)
AElem=(area/12)*[2 1 1; 1 2 1; 1 1 2];
```

LISTING 17
ElemMassWMatP1.m

```
function AElem=ElemMassWMatP1(area,w)
AElem=(area/30)* ...
            [3*w(1)+w(2)+w(3), w(1)+w(2)+w(3)/2, w(1)+w(2)/2+w(3); ...
             w(1)+w(2)+w(3)/2, w(1)+3*w(2)+w(3), w(1)/2+w(2)+w(3); ...
             w(1)+w(2)/2+w(3), w(1)/2+w(2)+w(3), w(1)+w(2)+3*w(3)];
```

LISTING 18
ElemStiffMatP1.m

```
function AElem=ElemStiffMatP1(q1,q2,q3,area)
M=[q2-q3, q3-q1, q1-q2];
AElem=(1/(4*area))*M'*M;
```

B.2. Classical matrix assembly code.

Listing 19
MassAssemblingP1base.m

```
function M=MassAssemblingP1base(nq,nme,me,areas)
M=sparse(nq,nq);
for k=1:nme
    E=ElemMassMatP1(areas(k));
    for il=1:3
        i=me(il,k);
        for jl=1:3
            j=me(jl,k);
            M(i,j)=M(i,j)+E(il,jl);
        end
    end
end
```

Listing 20
MassWAssemblingP1base.m

```
function M=MassWAssemblingP1base(nq,nme,me,areas,Tw)
M=sparse(nq,nq);
for k=1:nme
    for il=1:3
        i=me(il,k);
        Twloc(il)=Tw(i);
    end
    E=ElemMassWMatP1(areas(k),Twloc);
    for il=1:3
        i=me(il,k);
        for jl=1:3
            j=me(jl,k);
            M(i,j)=M(i,j)+E(il,jl);
        end
    end
end
```

Listing 21
StiffAssemblingP1base.m

```
function R=StiffAssemblingP1base(nq,nme,q,me,areas)
R=sparse(nq,nq);
for k=1:nme
    E=ElemStiffMatP1(q(:,me(1,k)),q(:,me(2,k)),q(:,me(3,k)),areas(k));
    for il=1:3
        i=me(il,k);
        for jl=1:3
            j=me(jl,k);
            R(i,j)=R(i,j)+E(il,jl);
        end
    end
end
```

B.3. Optimized matrix assembly codes - Version 0.

Listing 22
MassAssemblingP1OptV0.m

```
function M=MassAssemblingP1OptV0(nq,nme,me,areas)
M=sparse(nq,nq);
for k=1:nme
   I=me(:,k);
   M(I,I)=M(I,I)+ElemMassMatP1(areas(k));
end
```

Listing 23
MassWAssemblingP1OptV0.m

```
function M=MassWAssemblingP1OptV0(nq,nme,me,areas,Tw)
M=sparse(nq,nq);
for k=1:nme
   I=me(:,k);
   M(I,I)=M(I,I)+ElemMassWMatP1(areas(k),Tw(me(:,k)));
end
```

Listing 24
StiffAssemblingP1OptV0.m

```
function R=StiffAssemblingP1OptV0(nq,nme,q,me,areas)
R=sparse(nq,nq);
for k=1:nme
   I=me(:,k);
   Me=ElemStiffMatP1(q(:,me(1,k)),q(:,me(2,k)),q(:,me(3,k)),areas(k));
   R(I,I)=R(I,I)+Me;
end
```

B.4. Optimized matrix assembly codes - Version 1.

Listing 25
MassAssemblingP1OptV1.m

```
function M=MassAssemblingP1OptV1(nq,nme,me,areas)
Ig=zeros(9*nme,1);Jg=zeros(9*nme,1);Kg=zeros(9*nme,1);

ii=[1 2 3 1 2 3 1 2 3];
jj=[1 1 1 2 2 2 3 3 3];
kk=1:9;
for k=1:nme
   E=ElemMassMatP1(areas(k));
   Ig(kk)=me(ii,k);
   Jg(kk)=me(jj,k);
   Kg(kk)=E(:);
   kk=kk+9;
end
M=sparse(Ig,Jg,Kg,nq,nq);
```

Listing 26
MassWAssemblingP1OptV1.m

```
function M=MassWAssemblingP1OptV1(nq,nme,me,areas,Tw)
Ig=zeros(9*nme,1);Jg=zeros(9*nme,1);Kg=zeros(9*nme,1);

ii=[1 2 3 1 2 3 1 2 3];
jj=[1 1 1 2 2 2 3 3 3];
kk=1:9;
for k=1:nme
  E=ElemMassWMat(areas(k),Tw(me(:,k)));
  Ig(kk)=me(ii,k);
  Jg(kk)=me(jj,k);
  Kg(kk)=E(:);
  kk=kk+9;
end
M=sparse(Ig,Jg,Kg,nq,nq);
```

Listing 27
StiffAssemblingP1OptV1.m

```
function R=StiffAssemblingP1OptV1(nq,nme,q,me,areas)
Ig=zeros(nme*9,1);Jg=zeros(nme*9,1);
Kg=zeros(nme*9,1);

ii=[1 2 3 1 2 3 1 2 3];
jj=[1 1 1 2 2 2 3 3 3];
kk=1:9;
for k=1:nme
  Me=ElemStiffMatP1(q(:,me(1,k)),q(:,me(2,k)),q(:,me(3,k)),areas(k));
  Ig(kk)=me(ii,k);
  Jg(kk)=me(jj,k);
  Kg(kk)=Me(:);
  kk=kk+9;
end
R=sparse(Ig,Jg,Kg,nq,nq);
```

Appendix C. Matlab sparse trouble. In this part, we illustrate a problem that we encountered in the development of our codes : decrease of the performances of the assembly codes, for the classical and `OptV0` versions, when migrating from release R2011b to release R2012a or R2012b independently of the operating system used. In fact, this comes from the use of the command M = **sparse**(nq,nq). We illustrate this for the mass matrix assembly, by giving in Table C.1 the computation time of the function MassAssemblingP1OptV0 for different Matlab releases.

This problem has been reported to the MathWorks's development team :

```
As you have correctly pointed out, MATLAB 8.0 (R2012b) seems to perform
slower than the previous releases for this specific case of
reallocation in sparse matrices. I will convey this information to the
development team for further investigation and a possible fix in the
future releases of MATLAB. I apologize for the inconvenience.
```

To fix this issue in the releases R2012a and R2012b, it is recommended by the Matlab's technical support to use the function **spalloc** instead of the function **sparse** :

```
The matrix, 'M' in the function 'MassAssemblingP1OptV0' was initialized
```

```
using SPALLOC instead of the SPARSE command since the maximum number of
non-zeros in M are already known.

Previously existing line of code:
M = sparse(nq,nq);

Modified line of code:
M = spalloc(nq, nq, 9*nme);
```

Sparse dim	R2012b	R2012a	R2011b	R2011a
1600	0.167 (s) (×1.00)	0.155 (s) (×1.07)	0.139 (s) (×1.20)	0.116 (s) (×1.44)
3600	0.557 (s) (×1.00)	0.510 (s) (×1.09)	0.461 (s) (×1.21)	0.355 (s) (×1.57)
6400	1.406 (s) (×1.00)	1.278 (s) (×1.10)	1.150 (s) (×1.22)	0.843 (s) (×1.67)
10000	4.034 (s) (×1.00)	2.761 (s) (×1.46)	1.995 (s) (×2.02)	1.767 (s) (×2.28)
14400	8.545 (s) (×1.00)	6.625 (s) (×1.29)	3.734 (s) (×2.29)	3.295 (s) (×2.59)
19600	16.643 (s) (×1.00)	13.586 (s) (×1.22)	6.908 (s) (×2.41)	6.935 (s) (×2.40)
25600	29.489 (s) (×1.00)	27.815 (s) (×1.06)	12.367 (s) (×2.38)	11.175 (s) (×2.64)
32400	47.478 (s) (×1.00)	47.037 (s) (×1.01)	18.457 (s) (×2.57)	16.825 (s) (×2.82)
40000	73.662 (s) (×1.00)	74.188 (s) (×0.99)	27.753 (s) (×2.65)	25.012 (s) (×2.95)

TABLE C.1
MassAssemblingP1OptV0 for different Matlab releases : computation times and speedup

REFERENCES

[1] C. BERNARDI, Y. MADAY, AND A. PATERA, *A new nonconforming approach to domain decomposition: the mortar element method*, in Nonlinear Partial Differential Equations and their Applications, H. Brezis and J.L. Lions, eds., 1989.

[2] L. CHEN, *Programming of Finite Element Methods in Matlab*, Preprint, University of California Irvine, `http://math.uci.edu/~chenlong/226/Ch3FEMCode.pdf`, 2011.

[3] L. CHEN, *iFEM, a Matlab software package*, University of California Irvine, `http://math.uci.edu/~chenlong/programming.html`, 2013.

[4] P. G. CIARLET, *The finite element method for elliptic problems*, SIAM, Philadelphia, 2002.

[5] F. CUVELIER, *Méthodes des éléments finis. De la théorie à la programmation*, Lecture Notes, Université Paris 13, `http://www.math.univ-paris13.fr/~cuvelier/docs/poly/polyFEM2D.pdf`, 2008.

[6] F. CUVELIER, C. JAPHET, AND G. SCARELLA, *OptFEM2DP1, a MATLAB/Octave software package codes*, Université Paris 13, `http://www.math.univ-paris13.fr/~cuvelier`, 2012.

[7] F. CUVELIER, C. JAPHET, AND G. SCARELLA, *An efficient way to perform the assembly of finite element matrices in Matlab and Octave: the P_k finite element case*, in preparation.

[8] T. A. DAVIS, *Direct Methods for Sparse Linear Systems*, SIAM, 2006.

[9] T. A. DAVIS, *SuiteSparse packages, release 4.0.12*, University of Florida, `http://www.cise.ufl.edu/research/sparse/SuiteSparse`, 2012.

[10] G. DHATT, E. LEFRANÇOIS, AND G. TOUZOT, *Finite Element Method*, Wiley, 2012.

[11] *GNU Octave*, `http://www.gnu.org/software/octave`, 2012.

[12] 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.

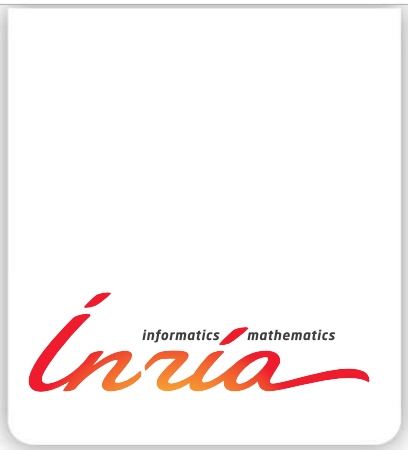
[13] F. HECHT, *FreeFEM++*, `http://www.freefem.org/ff++/index.htm`, 2012.

[14] C. JOHNSON, *Numerical Solution of Partial Differential Equations by the Finite Element Method*, Dover Publications, Inc, 2009.

[15] B. LUCQUIN, AND O. PIRONNEAU, *Introduction to Scientific Computing*, John Wiley & Sons Ltd, 1998.

[16] *MATLAB*, `http://www.mathworks.com`, 2012.

[17] T. RAHMAN, AND J. VALDMAN, *Fast MATLAB assembly of FEM matrices in 2D and 3D: Nodal elements*, Appl. Math. Comput., 219(13) (2013), pp. 7151–7158.



RESEARCH CENTRE
PARIS – ROCQUENCOURT

Domaine de Voluceau, - Rocquencourt
B.P. 105 - 78153 Le Chesnay Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399