



**HAL**  
open science

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

Caroline Japhet, François Cuvelier, Gilles Scarella

► **To cite this version:**

Caroline Japhet, François Cuvelier, Gilles Scarella. An efficient way to perform the assembly of finite element matrices in Matlab and Octave. 2013. hal-00785101v1

**HAL Id: hal-00785101**

**<https://hal.science/hal-00785101v1>**

Preprint submitted on 5 Feb 2013 (v1), last revised 14 May 2013 (v2)

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

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

# AN EFFICIENT WAY TO PERFORM THE ASSEMBLY OF FINITE ELEMENT MATRICES IN MATLAB AND OCTAVE

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

**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. We present numerical results which illustrate the computational costs of the different approaches.

**1. Introduction.** Usually, finite elements methods [Cia02, Joh09] 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 [BMP89]. Solving accurately these problems requires meshes containing a large number of elements and thus the assembly of large sparse matrices.

Matlab [Mat12] and GNU Octave [Oct12] are efficient numerical computing softwares using matrix-based language for teaching or industry calculations. However, the classical assembly algorithms (see for example [LP98]) basically implemented in Matlab/Octave are much less efficient than when implemented with other languages.

In [Dav06] 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 [Dav06]. Other more efficient algorithms have been proposed recently in [Che11, RV11, HJ12, Che13]. More precisely, in [HJ12], 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 [RV11], 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 [Che13], 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 [Che11], 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. Our method extends easily to the  $P_k$  finite elements case,  $k \geq 2$ , and in 3D, see [CJS]. We compare the performances of this code with the ones obtained with the standard algorithms and with those proposed in [Che11, RV11, HJ12, Che13]. We also show that this implementation is able to compete in performance with dedicated software such as FreeFEM++ [Hec12]. All the computations are done on our reference computer <sup>1</sup> 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 [CJS12]. 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 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 [Dav06]. 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 [Che11, RV11, HJ12, Che13]. The full listings of the routines used in the paper are given in Appendix B.

---

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

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

<sup>1</sup>2 x Intel Xeon E5645(6 cores) at 2.40Ghz, 32Go RAM, supported by GNR MoMaS

**2. Notations.** Let  $\Omega$  be an open bounded subset of  $\mathbb{R}^2$ . We use a triangulation  $\Omega_h$  of  $\Omega$  described by :

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

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  $n_q$  are sparse, and their coefficients are defined by

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

where  $\varphi_i$  are the usual basis functions,  $w$  is a function defined on  $\Omega$  and  $\langle \cdot, \cdot \rangle$  is the usual scalar product in  $\mathbb{R}^2$ . More details are given in [Cuv08]. To assemble this type of matrix, 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 defined by

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

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

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

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

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

We now give the usual assembly algorithm using these element matrices with a loop through the triangles.

**3. The classical algorithm.** We describe the assembly of a given matrix  $\mathbb{M}$  from its associated element matrix  $\mathbb{E}$ . We suppose that the `ElemMat` routine computing the element matrix is known.

LISTING 1  
Classical assembly

---

```

1  M=sparse(nq, nq);
2  for k=1:nme
3      E=ElemMat(areas(k), ...);
4      for il=1:3
5          i=me(il, k);
6          for jl=1:3
7              j=me(jl, k);
8              M(i, j)=M(i, j)+E(il, jl);
9          end
10     end
11 end

```

---

We aim to compare the performances of this code (see Appendix B.2 for the complete listings) with those obtained with FreeFEM++ [Hec12]. 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++.

---

```

1  mesh Th(...);
2  fespace Vh(Th,P1); // P1 FE-space
3  varf vMass (u,v)= int2d(Th)( u*v);
4  varf vMassW (u,v)= int2d(Th)( w*u*v);
5  varf vStiff (u,v)= int2d(Th)( dx(u)*dx(v) + dy(u)*dy(v) );
6  // Assembly
7  matrix M= vMass(Vh,Vh); // Build mass matrix
8  matrix Mw = vMassW(Vh,Vh); // Build weighted mass matrix
9  matrix S = vStiff(Vh,Vh); // Build stiffness matrix

```

---

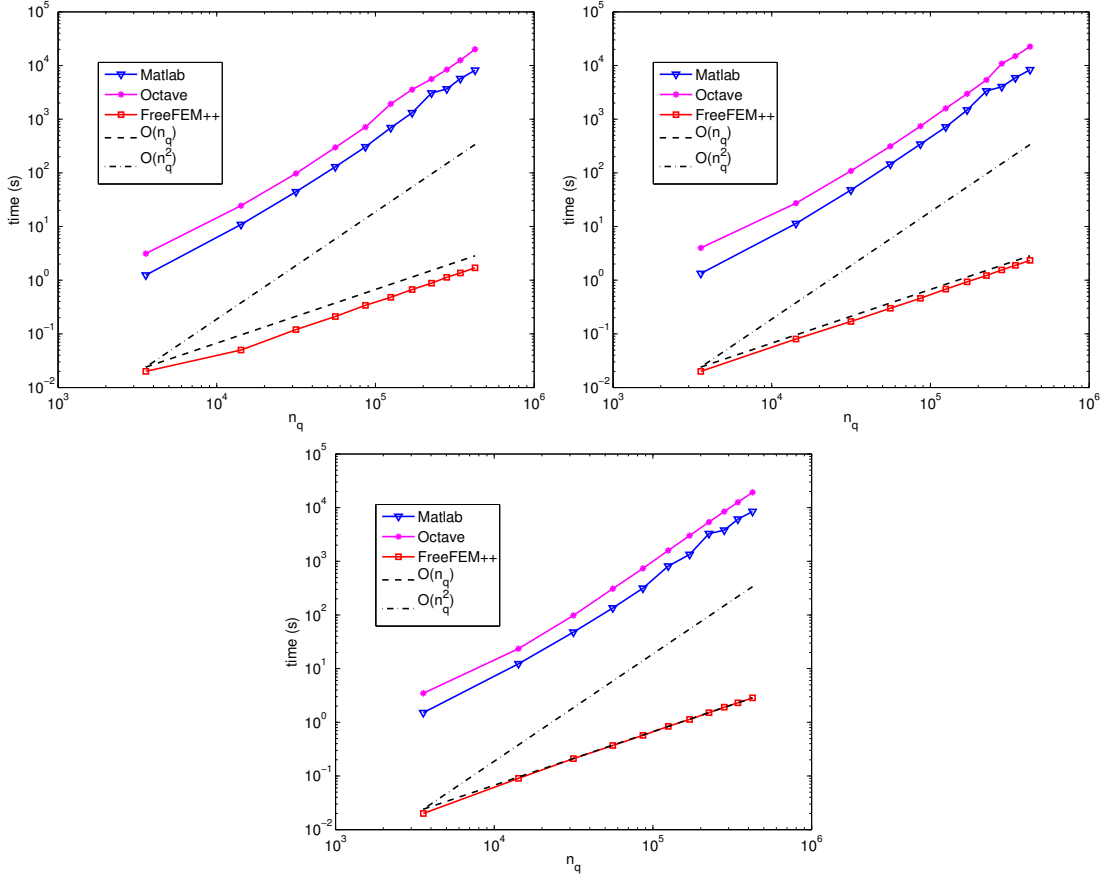


FIG. 3.1. Comparison of the usual assembly algorithms in Matlab/Octave with FreeFEM++, for the mass (top left), weighted mass (top right) and stiffness (bottom) matrices.

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).

---

```

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

```

---

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

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.** With Matlab or Octave, a sparse matrix  $\mathbb{A} \in \mathcal{M}_{M,N}(\mathbb{R})$  is stored with CSC (Compressed Sparse Column) format using the following three arrays :

$$ia(1 : nnz), ja(1 : N + 1) \text{ and } aa(1 : nnz),$$

where  $nnz$  is the number of non-zeros elements in the matrix  $\mathbb{A}$ . These arrays are defined by

- $aa$  : which contains the  $nnz$  non-zeros elements of  $\mathbb{A}$  stored column-wise.
- $ia$  : which contains the row numbers of the elements stored in  $aa$ .
- $ja$  : 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

$$aa \quad \boxed{1. \quad 5. \quad 1. \quad 2. \quad 6. \quad 4.}$$

$$ia \quad \boxed{1 \quad 2 \quad 3 \quad 3 \quad 1 \quad 2}$$

$$ja \quad \boxed{1 \quad 2 \quad 4 \quad 5 \quad 7}$$

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 row number 1 in  $ia(3)$  :

$$aa \quad \boxed{1. \quad 8. \quad 5. \quad 1. \quad 2. \quad 6. \quad 4.}$$

$$ia \quad \boxed{1 \quad 1 \quad 2 \quad 3 \quad 3 \quad 1 \quad 2}$$

For array  $ja$ , from the number column 2 plus one, one must increment of +1 :

$$ja \quad \boxed{1 \quad 2 \quad 5 \quad 6 \quad 8}$$

The repetition of these operations is expensive upon assembly of the matrix, in the previous codes (here we haven't considered dynamic allocation 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 version 1 (OptV1) .** We will use the following call of the `sparse` Matlab function:

$$M = \text{sparse}(I,J,K,m,n);$$

This command returns a sparse matrix  $M$  of size  $m \times n$  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  $\mathbf{I}_g$ ,  $\mathbf{J}_g$  and  $\mathbf{K}_g$  allowing the storage of the element matrices as well as the position of their elements in the global matrix. The length of each 1d-array is  $9n_{me}$ . Once these arrays are created, the matrix assembly is obtained with the command

$$M = \text{sparse}(\mathbf{I}_g,\mathbf{J}_g,\mathbf{K}_g,nq,nq);$$

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

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

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

$$\mathbb{E}(T_k) = \begin{pmatrix} e_{1,1}^k & e_{1,2}^k & e_{1,3}^k \\ e_{2,1}^k & e_{2,2}^k & e_{2,3}^k \\ e_{3,1}^k & e_{3,2}^k & e_{3,3}^k \end{pmatrix} \Rightarrow \begin{array}{l} \mathbf{K}_k^e : \boxed{e_{1,1}^k \quad e_{2,1}^k \quad e_{3,1}^k \quad e_{1,2}^k \quad e_{2,2}^k \quad e_{3,2}^k \quad e_{1,3}^k \quad e_{2,3}^k \quad e_{3,3}^k} \\ \mathbf{I}_k^e : \boxed{i_1^k \quad i_2^k \quad i_3^k \quad i_1^k \quad i_2^k \quad i_3^k \quad i_1^k \quad i_2^k \quad i_3^k} \\ \mathbf{J}_k^e : \boxed{j_1^k \quad j_1^k \quad j_1^k \quad j_2^k \quad j_2^k \quad j_2^k \quad j_3^k \quad j_3^k \quad j_3^k} \end{array}$$

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

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

---

```

1 E = ElemMat(areas(k), ...); % E : Matrix 3X3
2 Ke = E(:); % Ke : Matrix 9X1
3 Ie = me([1 2 3 1 2 3 1 2 3], k); % Ie : Matrix 9X1
4 Je = me([1 1 1 2 2 2 3 3 3], k); % Je : Matrix 9X1

```

---

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

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

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

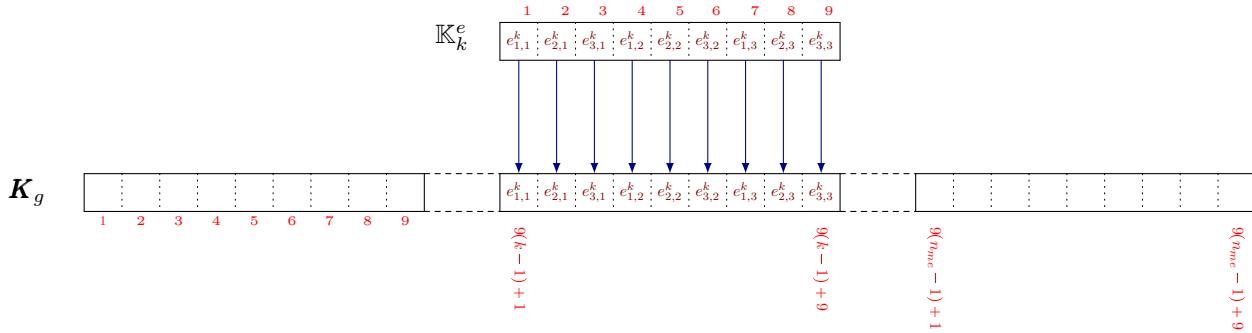


FIG. 5.1. Insertion of an element matrix in the global array - Version 1

We give below the Matlab/Octave associated code where the global vectors  $\mathbf{I}_g$ ,  $\mathbf{J}_g$  and  $\mathbf{K}_g$  are stored column-wise :

LISTING 4  
Optimized assembly - version 1

---

```

1 Ig=zeros(9*nme, 1); Jg=zeros(9*nme, 1); Kg=zeros(9*nme, 1);
2
3 ii=[1 2 3 1 2 3 1 2 3];
4 jj=[1 1 1 2 2 2 3 3 3];
5 kk=1:9;
6 for k=1:nme
7     E=ElemMat(areas(k), ...);
8     Ig(kk)=me(ii, k);
9     Jg(kk)=me(jj, k);
10    Kg(kk)=E(:);
11    kk=kk+9;
12 end
13 M=sparse(Ig, Jg, Kg, nq, nq);

```

---

The complete listings are given in Appendix B.4. 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).

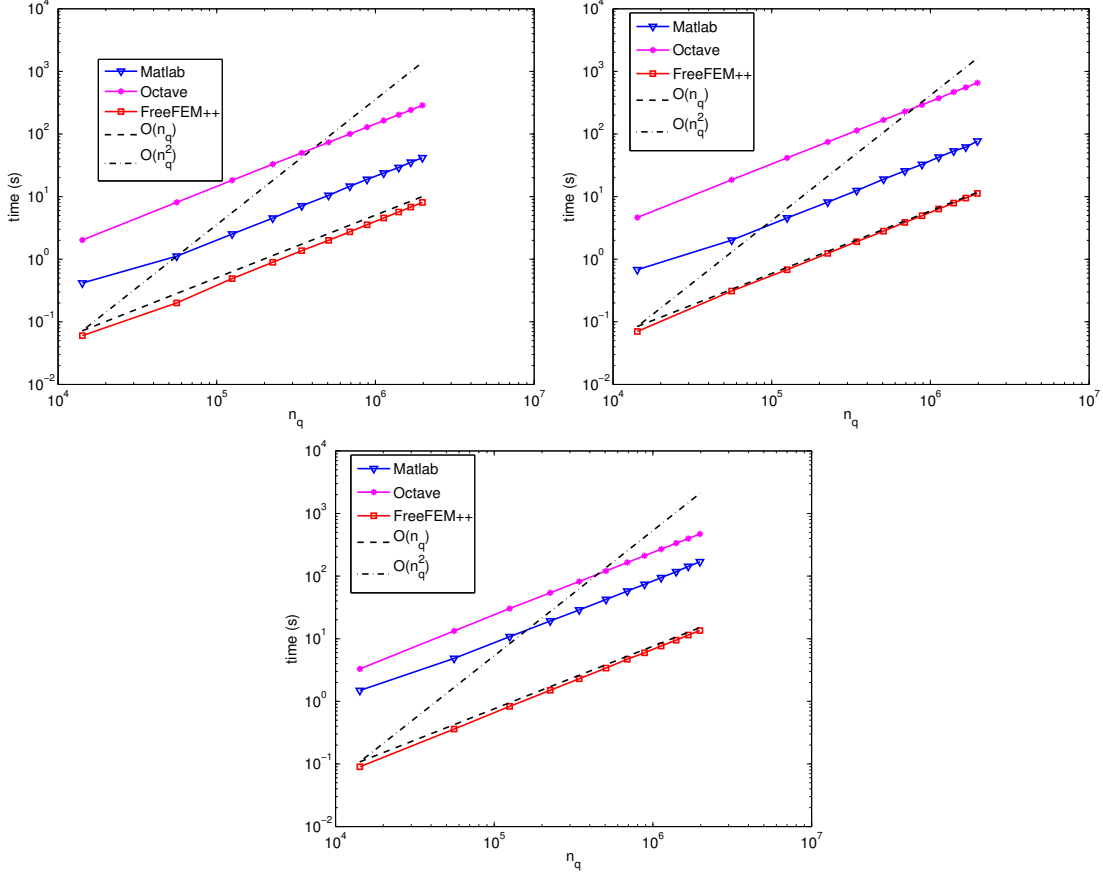


FIG. 5.2. Comparison of the assembly codes : *OptV1* in Matlab/Octave and FreeFEM++, for the mass (top left), weighted mass (top right) and stiffness (bottom) matrices.

The values of the computation times are given in Appendix A.3. The complexity of the Matlab/Octave codes seems now linear (i.e.  $\mathcal{O}(n_q)$ ) 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, for Matlab, see Appendix A.3).

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

**6. Optimized 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 2d-arrays (with nine rows and  $n_{me}$  columns), defined  $\forall k \in \llbracket 1, n_{me} \rrbracket$ ,  $\forall il \in \llbracket 1, 9 \rrbracket$  by

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

The three local arrays  $\mathbf{K}_k^e$ ,  $\mathbf{I}_k^e$  and  $\mathbf{J}_k^e$  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.

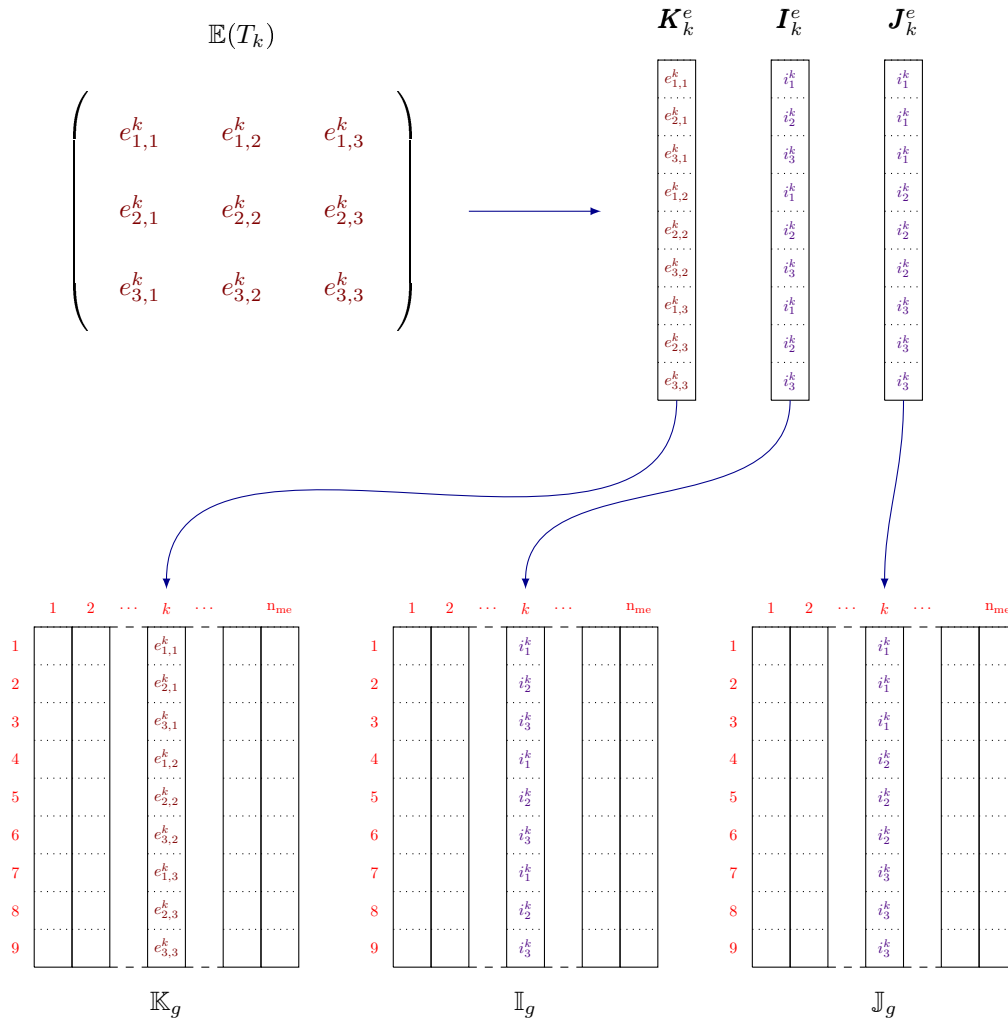


FIG. 6.1. Insertion of an element matrix in the global array - Version 2

Once these arrays are determined, the assembly matrix is obtained with the Matlab/Octave command

$$M = \text{sparse}(I_g(:,), J_g(:,), K_g(:,), n_q, n_q);$$

We remark that the matrices containing global indices  $I_g$  and  $J_g$  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 :

---

```

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

```

---



---

```

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

```

---

It remains to vectorize the computation of the 2d-array  $K_g$ . The usual code, corresponding to a column-wise computation, is :

---

```

1  Kg=zeros(9, nme);
2  for k=1:nme
3    E=ElemMat(areas(k), ...);
4    Kg(:, k)=E(:);
5  end

```

---

The vectorization of this code is done by the computation of the array  $K_g$  **row-wise**, for each matrix assembly. This corresponds to the permutation of the loop through the elements with the local loops, in the classical algorithm. This vectorization is different from the one proposed in [HJ12] as it doesn't use any quadrature



formula and it differs from L. Chen codes [Che11] by the full vectorization of arrays  $\mathbb{I}_g$  and  $\mathbb{J}_g$ .

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

**6.1. Mass matrix.** 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 :  $\forall k \in \llbracket 1, n_{me} \rrbracket$ ,

$$\mathbb{K}_g(\alpha, k) = \frac{|T_k|}{6}, \quad \forall \alpha \in \{1, 5, 9\},$$

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

We then build two arrays  $A_6$  and  $A_{12}$  of size  $1 \times n_{me}$  such that  $\forall k \in \llbracket 1, n_{me} \rrbracket$  :

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

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.

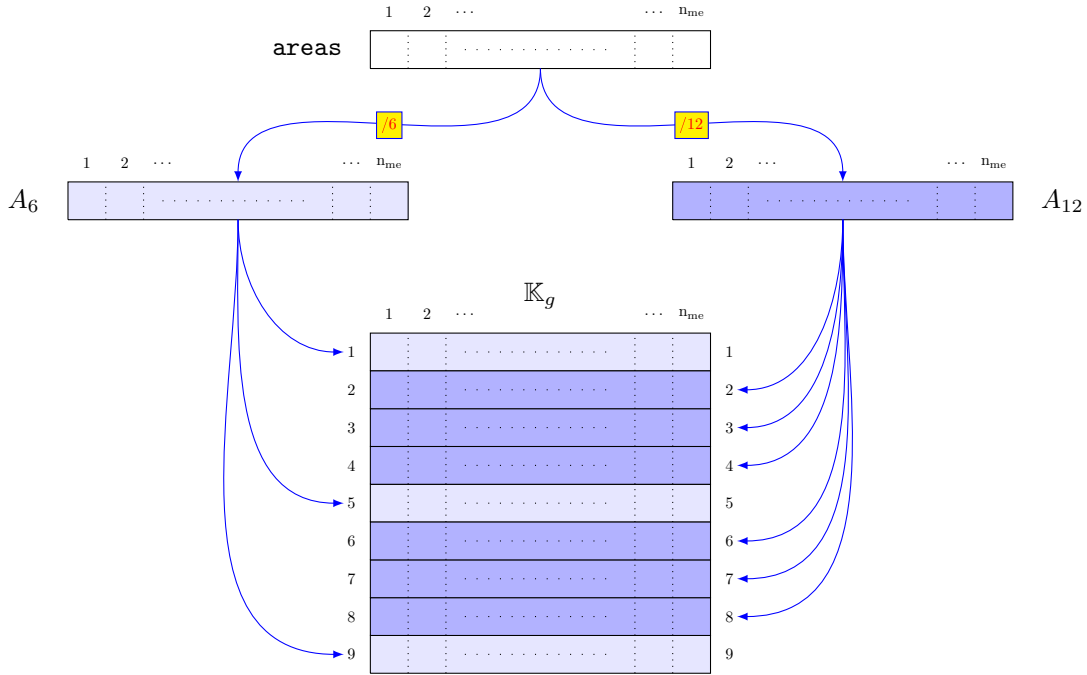


FIG. 6.2. Mass matrix assembly - Version 2

The Matlab/Octave code associated to this technique is :

LISTING 5  
*MassAssemblingP1OptV2.m*

---

```

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

```

---

**6.2. Weighted mass matrix.** The element weighted mass matrices  $\mathbb{M}^{e,[w_h]}(T_k)$  are given by (2.2). We introduce the array  $\mathbf{T}_w$  of size  $1 \times n_q$  defined by  $\mathbf{T}_w(i) = w(q^i)$ ,  $\forall i \in \llbracket 1, n_q \rrbracket$  and the three arrays  $\mathbf{W}_1, \mathbf{W}_2, \mathbf{W}_3$  of size  $1 \times n_{me}$  defined for all  $k \in \llbracket 1, n_{me} \rrbracket$  by

$$\mathbf{W}_1(k) = \frac{|T_k|}{30} \mathbf{T}_w(\text{me}(1, k)), \quad \mathbf{W}_2(k) = \frac{|T_k|}{30} \mathbf{T}_w(\text{me}(2, k)) \quad \text{and} \quad \mathbf{W}_3(k) = \frac{|T_k|}{30} \mathbf{T}_w(\text{me}(3, k)).$$

With these notations, we have

$$\mathbb{M}^{e,[w_h]}(T_k) = \begin{pmatrix} 3\mathbf{W}_1(k) + \mathbf{W}_2(k) + \mathbf{W}_3(k) & \mathbf{W}_1(k) + \mathbf{W}_2(k) + \frac{\mathbf{W}_3(k)}{2} & \mathbf{W}_1(k) + \frac{\mathbf{W}_2(k)}{2} + \mathbf{W}_3(k) \\ \mathbf{W}_1(k) + \mathbf{W}_2(k) + \frac{\mathbf{W}_3(k)}{2} & \mathbf{W}_1(k) + 3\mathbf{W}_2(k) + \mathbf{W}_3(k) & \frac{\mathbf{W}_1(k)}{2} + \mathbf{W}_2(k) + \mathbf{W}_3(k) \\ \mathbf{W}_1(k) + \frac{\mathbf{W}_2(k)}{2} + \mathbf{W}_3(k) & \frac{\mathbf{W}_1(k)}{2} + \mathbf{W}_2(k) + \mathbf{W}_3(k) & \mathbf{W}_1(k) + \mathbf{W}_2(k) + 3\mathbf{W}_3(k) \end{pmatrix}.$$

The code for computing these three arrays is given below, in a non-vectorized form (on the left) and in a vectorized form (in the middle) that may be reduced to a single line (on the right):

<pre> 1 W1=zeros(1,nme); 2 W2=zeros(1,nme); 3 W3=zeros(1,nme); 4 for k=1:nme 5     W1(k)=Tw(me(1,k))*areas(k)/30; 6     W2(k)=Tw(me(2,k))*areas(k)/30; 7     W3(k)=Tw(me(3,k))*areas(k)/30; 8 end </pre>	<pre> 1 Tw=Tw.*areas/30; 2 W1=Tw(me(1,:)); 3 W2=Tw(me(2,:)); 4 W3=Tw(me(3,:)); </pre>	<pre> 1 W=Tw(me).*(ones(3,1)*areas/30); </pre>
--	---	--

Here  $W$  is a matrix of size  $3 \times n_{me}$ , whose  $\ell$ -th row is  $\mathbf{W}_\ell$ ,  $1 \leq \ell \leq 3$ . We follow the method described on Figure 6.1. We have to vectorize the following code for  $\mathbb{K}_g$ :

```

1 Kg=zeros(9,nme);
2 for k=1:nme
3     Me=ElemMassWMat(areas(k),Tw(me(:,k)));
4     Kg(:,k)=Me(:);
5 end

```

Let  $\mathbf{K}_1, \mathbf{K}_2, \mathbf{K}_3, \mathbf{K}_5, \mathbf{K}_6, \mathbf{K}_9$  be six arrays of size  $1 \times n_{me}$  defined, for all  $k \in \llbracket 1, n_{me} \rrbracket$ , by

$$\begin{aligned} \mathbf{K}_1 &= 3\mathbf{W}_1 + \mathbf{W}_2 + \mathbf{W}_3, & \mathbf{K}_2 &= \mathbf{W}_1 + \mathbf{W}_2 + \frac{\mathbf{W}_3}{2}, & \mathbf{K}_3 &= \mathbf{W}_1 + \frac{\mathbf{W}_2}{2} + \mathbf{W}_3, \\ \mathbf{K}_5 &= \mathbf{W}_1 + 3\mathbf{W}_2 + \mathbf{W}_3, & \mathbf{K}_6 &= \frac{\mathbf{W}_1}{2} + \mathbf{W}_2 + \mathbf{W}_3, & \mathbf{K}_9 &= \mathbf{W}_1 + \mathbf{W}_2 + 3\mathbf{W}_3. \end{aligned}$$

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

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

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

```

1 K1=3*W1+W2+W3;
2 K2=W1+W2+W3/2;
3 K3=W1+W2/2+W3;
4 K5=W1+3*W2+W3;
5 K6=W1/2+W2+W3;
6 K9=W1+W2+3*W3;
7 Kg = [K1;K2;K3;K2;K5;K6;K3;K6;K9];

```

We represent this technique on Figure 6.3.

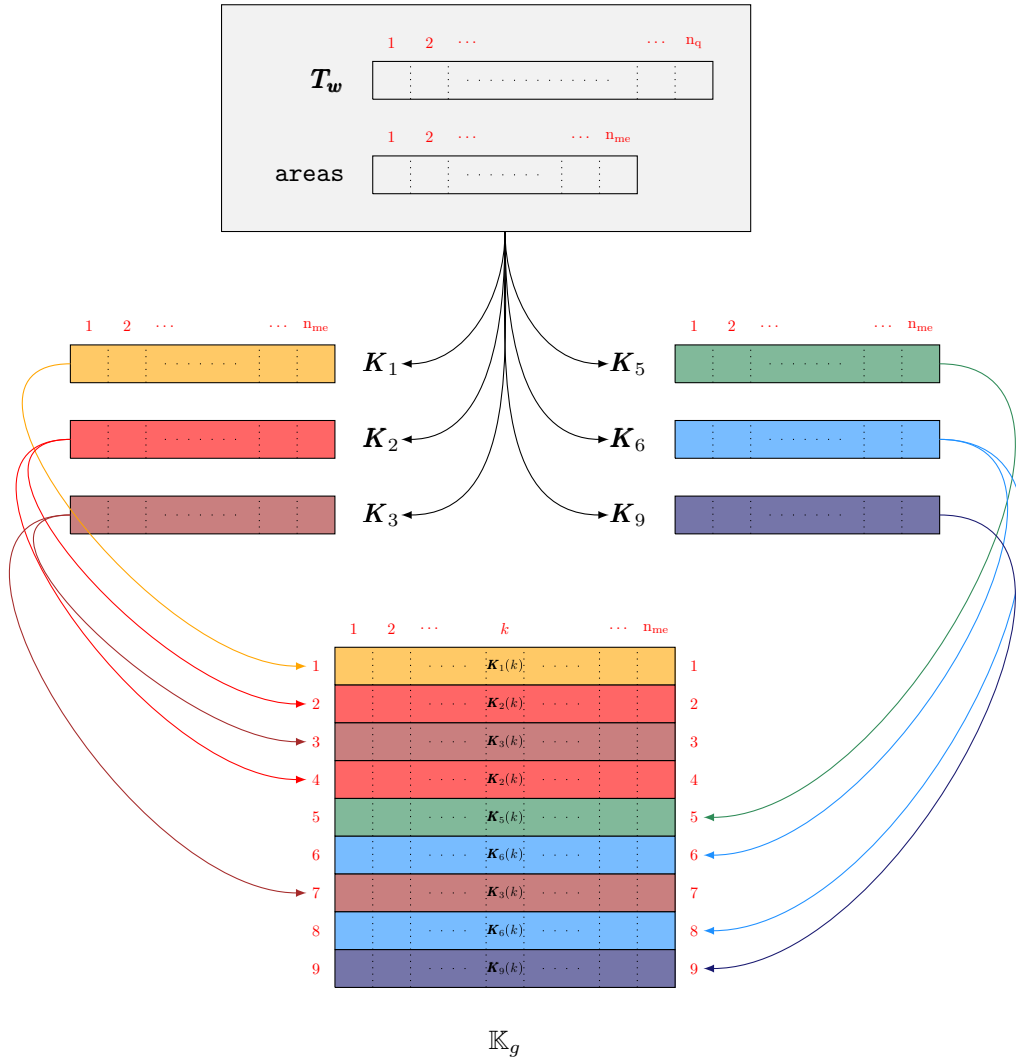


FIG. 6.3. *Weighted mass matrix assembly - Version 2*

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

LISTING 6  
*MassWAssemblingP1OptV2.m*

---

```

1  function M=MassWAssemblingP1OptV2(nq,nme,me,areas,Tw)
2  Ig = me([1 2 3 1 2 3 1 2 3],:);
3  Jg = me([1 1 1 2 2 2 3 3 3],:);
4  W=Tw(me).*(ones(3,1)*areas/30);
5  Kg=zeros(9,length(areas));
6  Kg(1,:)=3*W(1,:)+W(2,:)+W(3,:);
7  Kg(2,:)=W(1,:)+W(2,:)+W(3,:)/2;
8  Kg(3,:)=W(1,:)+W(2,:)/2+W(3,:);
9  Kg(5,:)=W(1,:)+3*W(2,:)+W(3,:);
10 Kg(6,:)=W(1,:)/2+W(2,:)+W(3,:);
11 Kg(9,:)=W(1,:)+W(2,:)+3*W(3,:);
12 Kg([4, 7, 8],:)=Kg([2, 3, 6],:);
13 M = sparse(Ig(:),Jg(:),Kg(:),nq,nq);

```

---

**6.3. Stiffness matrix.** The three vertices of the triangle  $T_k$  are  $q^{\text{me}(1,k)}$ ,  $q^{\text{me}(2,k)}$  and  $q^{\text{me}(3,k)}$ . We define  $\mathbf{u}^k = q^{\text{me}(2,k)} - q^{\text{me}(3,k)}$ ,  $\mathbf{v}^k = q^{\text{me}(3,k)} - q^{\text{me}(1,k)}$  and  $\mathbf{w}^k = q^{\text{me}(1,k)} - q^{\text{me}(2,k)}$ . Then, the element stiffness matrix

associated to  $T_k$  is

$$\mathbb{S}^e(T_k) = \frac{1}{4|T_k|} \begin{pmatrix} \langle \mathbf{u}^k, \mathbf{u}^k \rangle & \langle \mathbf{u}^k, \mathbf{v}^k \rangle & \langle \mathbf{u}^k, \mathbf{w}^k \rangle \\ \langle \mathbf{v}^k, \mathbf{u}^k \rangle & \langle \mathbf{v}^k, \mathbf{v}^k \rangle & \langle \mathbf{v}^k, \mathbf{w}^k \rangle \\ \langle \mathbf{w}^k, \mathbf{u}^k \rangle & \langle \mathbf{w}^k, \mathbf{v}^k \rangle & \langle \mathbf{w}^k, \mathbf{w}^k \rangle \end{pmatrix}.$$

We introduce the six arrays  $\mathbf{K}_1, \mathbf{K}_2, \mathbf{K}_3, \mathbf{K}_5, \mathbf{K}_6$  and  $\mathbf{K}_9$  of size  $1 \times n_{me}$  such that,  $\forall k \in \llbracket 1, n_{me} \rrbracket$ ,

$$\begin{aligned} \mathbf{K}_1(k) &= \frac{\langle \mathbf{u}^k, \mathbf{u}^k \rangle}{4|T_k|}, & \mathbf{K}_2(k) &= \frac{\langle \mathbf{u}^k, \mathbf{v}^k \rangle}{4|T_k|}, & \mathbf{K}_3(k) &= \frac{\langle \mathbf{u}^k, \mathbf{w}^k \rangle}{4|T_k|}, \\ \mathbf{K}_5(k) &= \frac{\langle \mathbf{v}^k, \mathbf{v}^k \rangle}{4|T_k|}, & \mathbf{K}_6(k) &= \frac{\langle \mathbf{v}^k, \mathbf{w}^k \rangle}{4|T_k|}, & \mathbf{K}_9(k) &= \frac{\langle \mathbf{w}^k, \mathbf{w}^k \rangle}{4|T_k|}. \end{aligned}$$

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

---

```

1 Kg = [K1;K2;K3;K2;K5;K6;K3;K6;K9];
2 R = sparse(Ig(:), Jg(:), Kg(:), nq, nq);

```

---

We now describe the vectorized computation of these six arrays. We introduce the arrays  $\mathbf{q}_\alpha \in \mathcal{M}_{2, n_{me}}(\mathbb{R})$ ,  $\alpha \in \llbracket 1, 3 \rrbracket$ , containing the coordinates of the three vertices  $\alpha = 1, 2, 3$  of the triangle  $T_k$  :

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

We give the code for these arrays in a non-vectorized form (on the left) and in a vectorized form (on the right) :

<pre> 1 q1=zeros(2, nme); q2=zeros(2, nme); q3=zeros(2, nme); 2 for k=1:nme 3     q1(:, k)=q(:, me(1, k)); 4     q2(:, k)=q(:, me(2, k)); 5     q3(:, k)=q(:, me(3, k)); 6 end </pre>	<pre> 1 q1=q(:, me(1, :)); 2 q2=q(:, me(2, :)); 3 q3=q(:, me(3, :)); </pre>
---	---

We trivially obtain the three arrays  $\mathbf{u}, \mathbf{v}$  and  $\mathbf{w}$  of size  $2 \times n_{me}$  whose  $k$ -th column is  $q^{me(2,k)} - q^{me(3,k)}$ ,  $q^{me(3,k)} - q^{me(1,k)}$  and  $q^{me(1,k)} - q^{me(2,k)}$  respectively.

The associated code is :

---

```

1 u=q2-q3;
2 v=q3-q1;
3 w=q1-q2;

```

---

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

---

```

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

```

---

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

---

```

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

```

---

**6.4. Numerical results.** We compare the performances of the `OptV2` codes with those of `FreeFEM++` and the methods in [Che11, RV11, HJ12, Che13]. The domain  $\Omega$  is the unit disk.

**6.4.1. Comparison with `FreeFEM++`.** On Figure 6.4, we show the computation times of the `OptV2` codes in Matlab and Octave and of the `FreeFEM++` codes, versus the number of vertices of the mesh. We give

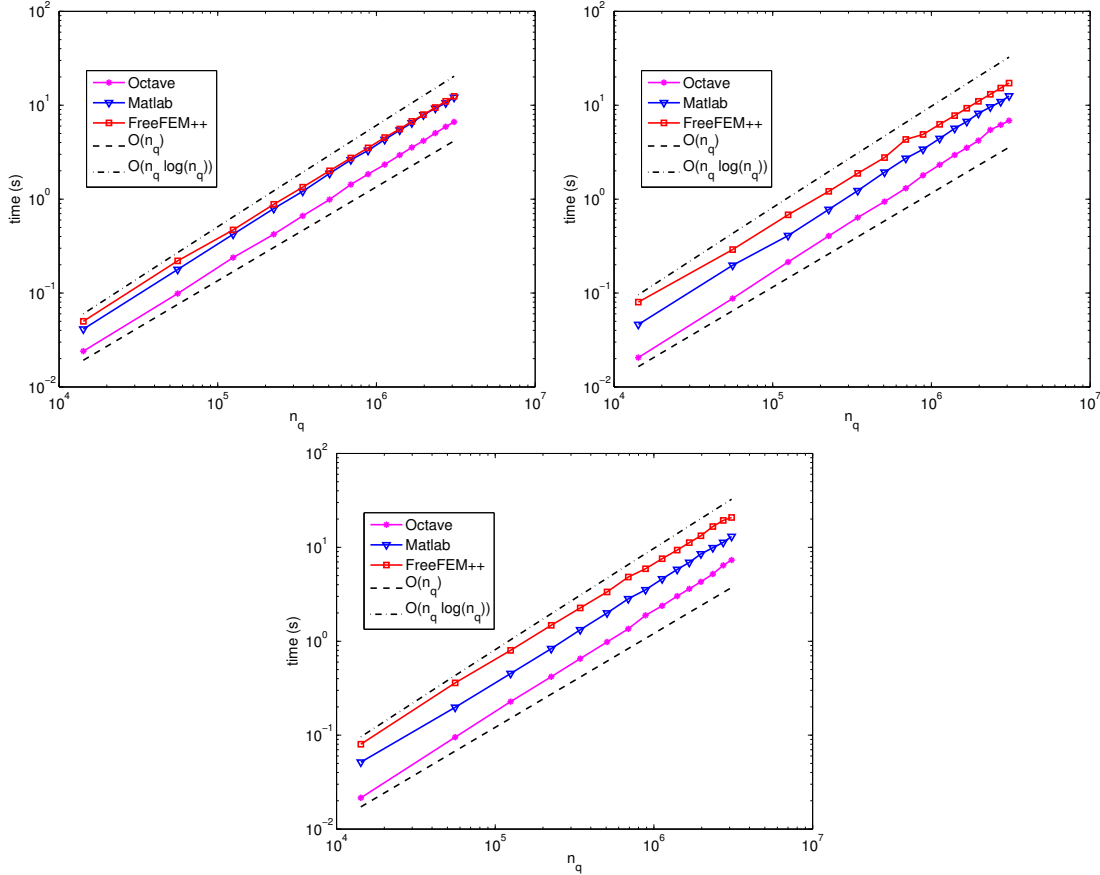


FIG. 6.4. Comparison of the assembly codes : `OptV2` in Matlab/Octave and `FreeFEM++`, for the mass matrix (top left), the weighted mass matrix (top right) and the stiffness matrix (bottom).

in Appendix A.4 the corresponding computation times values.

The complexity of the Matlab/Octave codes is still linear ( $\mathcal{O}(n_q)$ ) and slightly better than the one of `FreeFEM++`. Moreover, and 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 [Dav12], 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.

**6.4.2. Comparison with the assembly codes of [Che11, RV11, HJ12, Che13].** We compare, for the mass and stiffness matrices, the assembly codes proposed by T. Rahman and J. Valdman [RV11], A. Hannukainen and M. Juntunen [HJ12] and L. Chen [Che11, Che13] to the `OptV2` version developed in this paper. The computations have been done on our reference computer. On Figure 6.5 (with Matlab) and Figure 6.6 (with Octave), we show the computation times versus the number of vertices of the mesh (unit disk), for these different codes. The associated values are given in Tables 7.1 to 7.4. For large sparse matrices, our `OptV2` version allows gains in computational performance of 5% to 20%, compared to the other vectorized codes (for sufficiently large meshes).

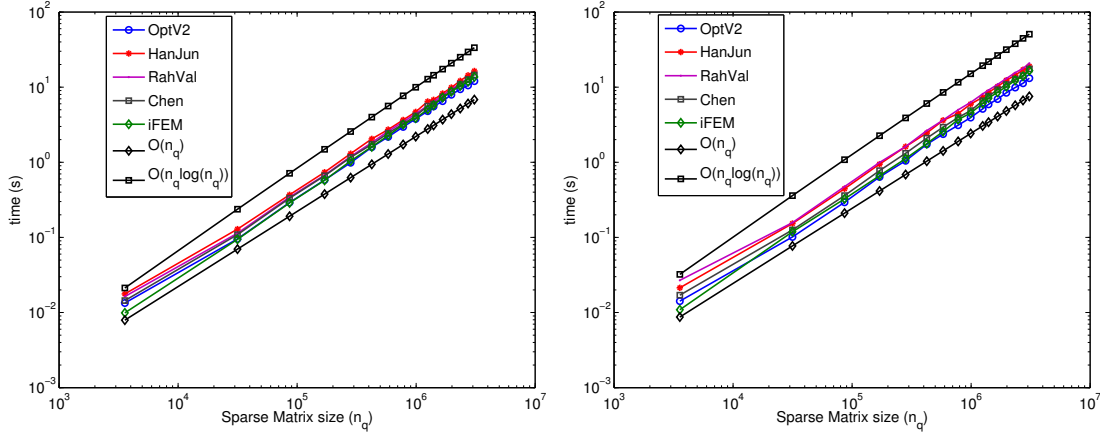


FIG. 6.5. Comparison of the assembly codes in Matlab R2012b : `OptV2` and [HJ12, RV11, Che11, Che13], for the mass (left) and stiffness (right) matrices.

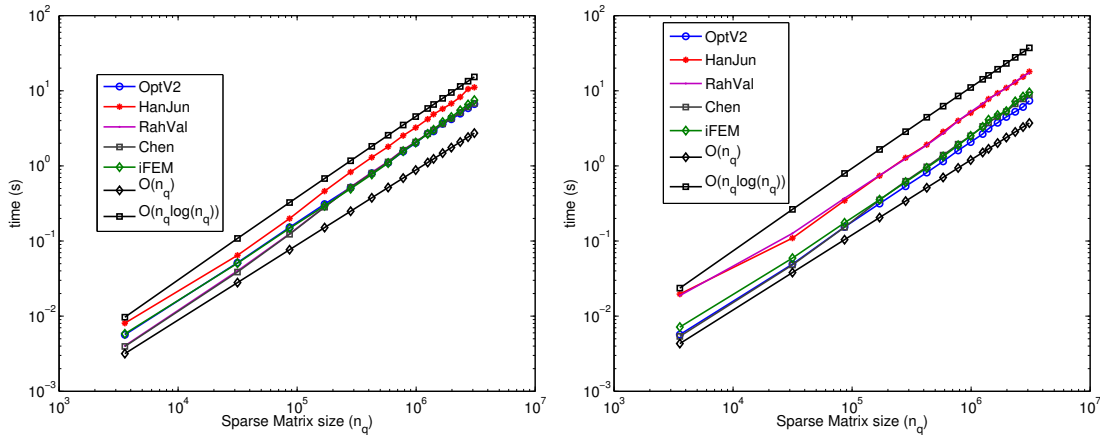


FIG. 6.6. Comparison of the assembly codes in Octave 3.6.3 : `OptV2` and [HJ12, RV11, Che11, Che13], for the mass (left) and stiffness (right) matrices.

**7. Conclusion.** For three examples of matrices, from the classical code we have built step by step the assembly codes to obtain a fully vectorized form. For each version, we have described the algorithm and its associated complexity. The assembly of matrices of size 10<sup>6</sup>, on our reference computer, is obtained in less than 4 seconds (resp. about 2 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. In mechanics, the same techniques have been used for the elastic stiffness matrix in dimension 2 and the gains obtained are about the same order of magnitude.

Moreover, in Matlab, it is possible to further improve the performances of the `OptV2` codes by using a GPU card. Preliminary results give a computation time divided by a factor 6 (compared to the `OptV2` without GPU).

$n_q$	OptV2	HanJun	RahVal	Chen	iFEM
3576	0.013 (s) x 1.00	0.018 (s) x 0.76	0.016 (s) x 0.82	0.014 (s) x 0.93	0.010 (s) x 1.35
31575	0.095 (s) x 1.00	0.129 (s) x 0.74	0.114 (s) x 0.84	0.109 (s) x 0.88	0.094 (s) x 1.01
86488	0.291 (s) x 1.00	0.368 (s) x 0.79	0.344 (s) x 0.85	0.333 (s) x 0.87	0.288 (s) x 1.01
170355	0.582 (s) x 1.00	0.736 (s) x 0.79	0.673 (s) x 0.86	0.661 (s) x 0.88	0.575 (s) x 1.01
281769	0.986 (s) x 1.00	1.303 (s) x 0.76	1.195 (s) x 0.83	1.162 (s) x 0.85	1.041 (s) x 0.95
424178	1.589 (s) x 1.00	2.045 (s) x 0.78	1.825 (s) x 0.87	1.735 (s) x 0.92	1.605 (s) x 0.99
582024	2.179 (s) x 1.00	2.724 (s) x 0.80	2.588 (s) x 0.84	2.438 (s) x 0.89	2.267 (s) x 0.96
778415	2.955 (s) x 1.00	3.660 (s) x 0.81	3.457 (s) x 0.85	3.240 (s) x 0.91	3.177 (s) x 0.93
992675	3.774 (s) x 1.00	4.682 (s) x 0.81	4.422 (s) x 0.85	4.146 (s) x 0.91	3.868 (s) x 0.98
1251480	4.788 (s) x 1.00	6.443 (s) x 0.74	5.673 (s) x 0.84	5.590 (s) x 0.86	5.040 (s) x 0.95
1401129	5.526 (s) x 1.00	6.790 (s) x 0.81	6.412 (s) x 0.86	5.962 (s) x 0.93	5.753 (s) x 0.96
1671052	6.507 (s) x 1.00	8.239 (s) x 0.79	7.759 (s) x 0.84	7.377 (s) x 0.88	7.269 (s) x 0.90
1978602	7.921 (s) x 1.00	9.893 (s) x 0.80	9.364 (s) x 0.85	8.807 (s) x 0.90	8.720 (s) x 0.91
2349573	9.386 (s) x 1.00	12.123 (s) x 0.77	11.160 (s) x 0.84	10.969 (s) x 0.86	10.388 (s) x 0.90
2732448	10.554 (s) x 1.00	14.343 (s) x 0.74	13.087 (s) x 0.81	12.680 (s) x 0.83	11.842 (s) x 0.89
3085628	12.034 (s) x 1.00	16.401 (s) x 0.73	14.950 (s) x 0.80	14.514 (s) x 0.83	13.672 (s) x 0.88

TABLE 7.1

Computational cost, in Matlab (R2012b), of the *Mass* matrix assembly versus  $n_q$ , with the *OptV2* version (column 2) and with the codes in [HJ12, RV11, Che11, Che13] (columns 3-6) : time in seconds (top value) and speedup (bottom value). The speedup reference is *OptV2* version.

$n_q$	OptV2	HanJun	RahVal	Chen	iFEM
3576	0.014 (s) x 1.00	0.021 (s) x 0.66	0.027 (s) x 0.53	0.017 (s) x 0.83	0.011 (s) x 1.30
31575	0.102 (s) x 1.00	0.153 (s) x 0.66	0.157 (s) x 0.65	0.126 (s) x 0.81	0.119 (s) x 0.86
86488	0.294 (s) x 1.00	0.444 (s) x 0.66	0.474 (s) x 0.62	0.360 (s) x 0.82	0.326 (s) x 0.90
170355	0.638 (s) x 1.00	0.944 (s) x 0.68	0.995 (s) x 0.64	0.774 (s) x 0.82	0.663 (s) x 0.96
281769	1.048 (s) x 1.00	1.616 (s) x 0.65	1.621 (s) x 0.65	1.316 (s) x 0.80	1.119 (s) x 0.94
424178	1.733 (s) x 1.00	2.452 (s) x 0.71	2.634 (s) x 0.66	2.092 (s) x 0.83	1.771 (s) x 0.98
582024	2.369 (s) x 1.00	3.620 (s) x 0.65	3.648 (s) x 0.65	2.932 (s) x 0.81	2.565 (s) x 0.92
778415	3.113 (s) x 1.00	4.446 (s) x 0.70	4.984 (s) x 0.62	3.943 (s) x 0.79	3.694 (s) x 0.84
992675	3.933 (s) x 1.00	5.948 (s) x 0.66	6.270 (s) x 0.63	4.862 (s) x 0.81	4.525 (s) x 0.87
1251480	5.142 (s) x 1.00	7.320 (s) x 0.70	8.117 (s) x 0.63	6.595 (s) x 0.78	6.056 (s) x 0.85
1401129	5.901 (s) x 1.00	8.510 (s) x 0.69	9.132 (s) x 0.65	7.590 (s) x 0.78	7.148 (s) x 0.83
1671052	6.937 (s) x 1.00	10.174 (s) x 0.68	10.886 (s) x 0.64	9.233 (s) x 0.75	8.557 (s) x 0.81
1978602	8.410 (s) x 1.00	12.315 (s) x 0.68	13.006 (s) x 0.65	10.845 (s) x 0.78	10.153 (s) x 0.83
2349573	9.892 (s) x 1.00	14.384 (s) x 0.69	15.585 (s) x 0.63	12.778 (s) x 0.77	12.308 (s) x 0.80
2732448	11.255 (s) x 1.00	17.035 (s) x 0.66	17.774 (s) x 0.63	14.259 (s) x 0.79	13.977 (s) x 0.81
3085628	13.157 (s) x 1.00	18.938 (s) x 0.69	20.767 (s) x 0.63	17.419 (s) x 0.76	16.575 (s) x 0.79

TABLE 7.2

Computational cost, in Matlab (R2012b), of the *Stiffness* matrix assembly versus  $n_q$ , with the *OptV2* version (column 2) and with the codes in [HJ12, RV11, Che11, Che13] (columns 3-6) : time in seconds (top value) and speedup (bottom value). The speedup reference is *OptV2* version.

$n_q$	OptV2	HanJun	RahVal	Chen	iFEM
3576	0.006 (s) x 1.00	0.008 (s) x 0.70	0.004 (s) x 1.40	0.004 (s) x 1.43	0.006 (s) x 0.97
31575	0.051 (s) x 1.00	0.065 (s) x 0.80	0.040 (s) x 1.29	0.039 (s) x 1.33	0.051 (s) x 1.02
86488	0.152 (s) x 1.00	0.199 (s) x 0.76	0.125 (s) x 1.22	0.123 (s) x 1.24	0.148 (s) x 1.03
170355	0.309 (s) x 1.00	0.462 (s) x 0.67	0.284 (s) x 1.09	0.282 (s) x 1.10	0.294 (s) x 1.05
281769	0.515 (s) x 1.00	0.828 (s) x 0.62	0.523 (s) x 0.99	0.518 (s) x 1.00	0.497 (s) x 1.04
424178	0.799 (s) x 1.00	1.297 (s) x 0.62	0.820 (s) x 0.97	0.800 (s) x 1.00	0.769 (s) x 1.04
582024	1.101 (s) x 1.00	1.801 (s) x 0.61	1.145 (s) x 0.96	1.127 (s) x 0.98	1.091 (s) x 1.01
778415	1.549 (s) x 1.00	2.530 (s) x 0.61	1.633 (s) x 0.95	1.617 (s) x 0.96	1.570 (s) x 0.99
992675	2.020 (s) x 1.00	3.237 (s) x 0.62	2.095 (s) x 0.96	2.075 (s) x 0.97	2.049 (s) x 0.99
1251480	2.697 (s) x 1.00	4.190 (s) x 0.64	2.684 (s) x 1.01	2.682 (s) x 1.01	2.666 (s) x 1.01
1401129	2.887 (s) x 1.00	4.874 (s) x 0.59	3.161 (s) x 0.91	2.989 (s) x 0.97	3.025 (s) x 0.95
1671052	3.622 (s) x 1.00	5.750 (s) x 0.63	3.646 (s) x 0.99	3.630 (s) x 1.00	3.829 (s) x 0.95
1978602	4.176 (s) x 1.00	6.766 (s) x 0.62	4.293 (s) x 0.97	4.277 (s) x 0.98	4.478 (s) x 0.93
2349573	4.966 (s) x 1.00	8.267 (s) x 0.60	5.155 (s) x 0.96	5.125 (s) x 0.97	5.499 (s) x 0.90
2732448	5.862 (s) x 1.00	10.556 (s) x 0.56	6.080 (s) x 0.96	6.078 (s) x 0.96	6.575 (s) x 0.89
3085628	6.634 (s) x 1.00	11.109 (s) x 0.60	6.833 (s) x 0.97	6.793 (s) x 0.98	7.500 (s) x 0.88

TABLE 7.3

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 [HJ12, RV11, Che11, Che13] (columns 3-6) : time in seconds (top value) and speedup (bottom value). The speedup reference is *OptV2* version.

$n_q$	OptV2	HanJun	RahVal	Chen	iFEM
3576	0.006 (s) x 1.00	0.020 (s) x 0.29	0.019 (s) x 0.30	0.005 (s) x 1.05	0.007 (s) x 0.79
31575	0.049 (s) x 1.00	0.109 (s) x 0.45	0.127 (s) x 0.39	0.048 (s) x 1.02	0.059 (s) x 0.83
86488	0.154 (s) x 1.00	0.345 (s) x 0.44	0.371 (s) x 0.41	0.152 (s) x 1.01	0.175 (s) x 0.88
170355	0.315 (s) x 1.00	0.740 (s) x 0.43	0.747 (s) x 0.42	0.353 (s) x 0.89	0.355 (s) x 0.89
281769	0.536 (s) x 1.00	1.280 (s) x 0.42	1.243 (s) x 0.43	0.624 (s) x 0.86	0.609 (s) x 0.88
424178	0.815 (s) x 1.00	1.917 (s) x 0.42	1.890 (s) x 0.43	0.970 (s) x 0.84	0.942 (s) x 0.86
582024	1.148 (s) x 1.00	2.846 (s) x 0.40	2.707 (s) x 0.42	1.391 (s) x 0.83	1.336 (s) x 0.86
778415	1.604 (s) x 1.00	3.985 (s) x 0.40	3.982 (s) x 0.40	1.945 (s) x 0.82	1.883 (s) x 0.85
992675	2.077 (s) x 1.00	5.076 (s) x 0.41	5.236 (s) x 0.40	2.512 (s) x 0.83	2.514 (s) x 0.83
1251480	2.662 (s) x 1.00	6.423 (s) x 0.41	6.752 (s) x 0.39	3.349 (s) x 0.79	3.307 (s) x 0.81
1401129	3.128 (s) x 1.00	7.766 (s) x 0.40	7.748 (s) x 0.40	3.761 (s) x 0.83	4.120 (s) x 0.76
1671052	3.744 (s) x 1.00	9.310 (s) x 0.40	9.183 (s) x 0.41	4.533 (s) x 0.83	4.750 (s) x 0.79
1978602	4.482 (s) x 1.00	10.939 (s) x 0.41	10.935 (s) x 0.41	5.268 (s) x 0.85	5.361 (s) x 0.84
2349573	5.253 (s) x 1.00	12.973 (s) x 0.40	13.195 (s) x 0.40	6.687 (s) x 0.79	7.227 (s) x 0.73
2732448	6.082 (s) x 1.00	15.339 (s) x 0.40	15.485 (s) x 0.39	7.782 (s) x 0.78	8.376 (s) x 0.73
3085628	7.363 (s) x 1.00	18.001 (s) x 0.41	17.375 (s) x 0.42	8.833 (s) x 0.83	9.526 (s) x 0.77

TABLE 7.4

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 [HJ12, RV11, Che11, Che13] (columns 3-6) : time in seconds (top value) and speedup (bottom value). The speedup reference is *OptV2* version.



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

### A.1. Classical 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  $n_q$ , 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  $n_q$ , 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 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 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 code vs FreeFEM++.

$n_q$	Octave (3.6.3)	Matlab (R2012b)	FreeFEM++ (3.20)
14222	0.024 (s) x 1.00	0.041 (s) x 0.58	0.050 (s) x 0.48
55919	0.099 (s) x 1.00	0.178 (s) x 0.55	0.220 (s) x 0.45
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)
14222	0.021 (s) x 1.00	0.046 (s) x 0.44	0.080 (s) x 0.26
55919	0.088 (s) x 1.00	0.197 (s) x 0.44	0.290 (s) x 0.30
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)
14222	0.022 (s) x 1.00	0.052 (s) x 0.42	0.080 (s) x 0.27
55919	0.095 (s) x 1.00	0.198 (s) x 0.48	0.360 (s) x 0.26
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  $n_q$ , 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. Codes.

### B.1. Element matrices.

LISTING 8

*ElemMassMatP1.m*

---

```

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

```

---

LISTING 9

*ElemMassWMatP1.m*

---

```

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

```

---

LISTING 10

*ElemStiffMatP1.m*

---

```

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

```

---

### B.2. Classical code.

LISTING 11

*MassAssemblingP1base.m*

---

```

1 function M=MassAssemblingP1base( nq , nme , me , areas )
2 M=sparse( nq , nq );
3 for k=1:nme
4 E=ElemMassMatP1( areas( k ) );
5 for il=1:3
6 i=me( il , k );
7 for jl=1:3
8 j=me( jl , k );
9 M( i , j )=M( i , j )+E( il , jl );
10 end
11 end
12 end

```

---

```

1 function M=MassWAssemblingP1base(nq,nme,me,areas,Tw)
2 M=sparse(nq,nq);
3 for k=1:nme
4     for il=1:3
5         i=me(il,k);
6         Twloc(il)=Tw(i);
7     end
8     E=ElemMassWMatP1(areas(k),Twloc);
9     for il=1:3
10        i=me(il,k);
11        for jl=1:3
12            j=me(jl,k);
13            M(i,j)=M(i,j)+E(il,jl);
14        end
15    end
16 end

```

---

```

1 function R=StiffAssemblingP1base(nq,nme,q,me,areas)
2 R=sparse(nq,nq);
3 for k=1:nme
4     E=ElemStiffMatP1(q(:,me(1,k)),q(:,me(2,k)),q(:,me(3,k)),areas(k));
5     for il=1:3
6         i=me(il,k);
7         for jl=1:3
8             j=me(jl,k);
9             R(i,j)=R(i,j)+E(il,jl);
10        end
11    end
12 end

```

---

### B.3. Optimized codes - Version 0.

```

1 function M=MassAssemblingP1OptV0(nq,nme,me,areas)
2 M=sparse(nq,nq);
3 for k=1:nme
4     I=me(:,k);
5     M(I,I)=M(I,I)+ElemMassMatP1(areas(k));
6 end

```

---

```

1 function M=MassWAssemblingP1OptV0(nq,nme,me,areas,Tw)
2 M=sparse(nq,nq);
3 for k=1:nme
4     I=me(:,k);
5     M(I,I)=M(I,I)+ElemMassWMatP1(areas(k),Tw(me(:,k)));
6 end

```

---

```

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

```

---

## B.4. Optimized codes - Version 1.

LISTING 17  
*MassAssemblingP1OptV1.m*

---

```
1 function M=MassAssemblingP1OptV1(nq,nme,me,areas)
2 Ig=zeros(9*nme,1);Jg=zeros(9*nme,1);Kg=zeros(9*nme,1);
3
4 ii=[1 2 3 1 2 3 1 2 3];
5 jj=[1 1 1 2 2 2 3 3 3];
6 kk=1:9;
7 for k=1:nme
8     E=ElemMassMatP1(areas(k));
9     Ig(kk)=me(ii,k);
10    Jg(kk)=me(jj,k);
11    Kg(kk)=E(:);
12    kk=kk+9;
13 end
14 M=sparse(Ig,Jg,Kg,nq,nq);
```

---

LISTING 18  
*MassWAssemblingP1OptV1.m*

---

```
1 function M=MassWAssemblingP1OptV1(nq,nme,me,areas,Tw)
2 Ig=zeros(9*nme,1);Jg=zeros(9*nme,1);Kg=zeros(9*nme,1);
3
4 ii=[1 2 3 1 2 3 1 2 3];
5 jj=[1 1 1 2 2 2 3 3 3];
6 kk=1:9;
7 for k=1:nme
8     E=ElemMassWMat(areas(k),Tw(me(:,k)));
9     Ig(kk)=me(ii,k);
10    Jg(kk)=me(jj,k);
11    Kg(kk)=E(:);
12    kk=kk+9;
13 end
14 M=sparse(Ig,Jg,Kg,nq,nq);
```

---

LISTING 19  
*StiffAssemblingP1OptV1.m*

---

```
1 function R=StiffAssemblingP1OptV1(nq,nme,q,me,areas)
2 Ig=zeros(nme*9,1);Jg=zeros(nme*9,1);
3 Kg=zeros(nme*9,1);
4
5 ii=[1 2 3 1 2 3 1 2 3];
6 jj=[1 1 1 2 2 2 3 3 3];
7 kk=1:9;
8 for k=1:nme
9     Me=ElemStiffMatP1(q(:,me(1,k)),q(:,me(2,k)),q(:,me(3,k)),areas(k));
10    Ig(kk)=me(ii,k);
11    Jg(kk)=me(jj,k);
12    Kg(kk)=Me(:);
13    kk=kk+9;
14 end
15 R=sparse(Ig,Jg,Kg,nq,nq);
```

---

## B.5. Optimized codes - Version 2.

LISTING 20  
*MassAssemblingP1OptV2.m*

---

```
1 function M=MassAssemblingP1OptV2(nq,nme,me,areas)
2 me=double(me);
3 Ig = me([1 2 3 1 2 3 1 2 3],:);
4 Jg = me([1 1 1 2 2 2 3 3 3],:);
5 a6=areas/6;
6 a12=areas/12;
7 Kg = [a6;a12;a12;a12;a6;a12;a12;a12;a6];
8 M = sparse(Ig,Jg,Kg,nq,nq);
```

---

---

```

1 function M=MassWAssemblingP1OptV2(nq,nme,me,areas,Tw)
2 Ig = me([1 2 3 1 2 3 1 2 3],:);
3 Jg = me([1 1 1 2 2 2 3 3 3],:);
4 W=Tw(me).*(ones(3,1)*areas/30);
5 Kg=zeros(9,length(areas));
6 Kg(1,:)=3*W(1,:)+W(2,:)+W(3,:);
7 Kg(2,:)=W(1,:)+W(2,:)+W(3,)/2;
8 Kg(3,:)=W(1,:)+W(2,)/2+W(3,:);
9 Kg(5,:)=W(1,:)+3*W(2,:)+W(3,:);
10 Kg(6,:)=W(1,)/2+W(2,:)+W(3,:);
11 Kg(9,:)=W(1,:)+W(2,)+3*W(3,:);
12 Kg([4, 7, 8],:)=Kg([2, 3, 6],:);
13 M = sparse(Ig,Jg,Kg,nq,nq);

```

---



---

```

1 function R=StiffAssemblingP1OptV2(nq,nme,q,me,areas)
2 Ig = me([1 2 3 1 2 3 1 2 3],:);
3 Jg = me([1 1 1 2 2 2 3 3 3],:);
4
5 q1 =q(:,me(1,:)); q2 =q(:,me(2,:)); q3 =q(:,me(3,:));
6 u = q2-q3; v=q3-q1; w=q1-q2;
7 clear q1 q2 q3
8 areas4=4*areas;
9 Kg=zeros(9,nme);
10 Kg(1,:)=sum(u.*u,1)./areas4;
11 Kg(2,:)=sum(v.*u,1)./areas4;
12 Kg(3,:)=sum(w.*u,1)./areas4;
13 Kg(5,:)=sum(v.*v,1)./areas4;
14 Kg(6,:)=sum(w.*v,1)./areas4;
15 Kg(9,:)=sum(w.*w,1)./areas4;
16 Kg([4, 7, 8],:)=Kg([2, 3, 6],:);
17 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.

Sparse dim	R2012b	R2012a	R2011b	R2011a
400	0.029 (s) ( $\times 1.00$ )	0.027 (s) ( $\times 1.07$ )	0.026 (s) ( $\times 1.11$ )	0.021 (s) ( $\times 1.40$ )
1600	0.167 (s) ( $\times 1.00$ )	0.155 (s) ( $\times 1.07$ )	0.139 (s) ( $\times 1.20$ )	0.116 (s) ( $\times 1.44$ )
3600	0.557 (s) ( $\times 1.00$ )	0.510 (s) ( $\times 1.09$ )	0.461 (s) ( $\times 1.21$ )	0.355 (s) ( $\times 1.57$ )
6400	1.406 (s) ( $\times 1.00$ )	1.278 (s) ( $\times 1.10$ )	1.150 (s) ( $\times 1.22$ )	0.843 (s) ( $\times 1.67$ )
10000	4.034 (s) ( $\times 1.00$ )	2.761 (s) ( $\times 1.46$ )	1.995 (s) ( $\times 2.02$ )	1.767 (s) ( $\times 2.28$ )
14400	8.545 (s) ( $\times 1.00$ )	6.625 (s) ( $\times 1.29$ )	3.734 (s) ( $\times 2.29$ )	3.295 (s) ( $\times 2.59$ )
19600	16.643 (s) ( $\times 1.00$ )	13.586 (s) ( $\times 1.22$ )	6.908 (s) ( $\times 2.41$ )	6.935 (s) ( $\times 2.40$ )
25600	29.489 (s) ( $\times 1.00$ )	27.815 (s) ( $\times 1.06$ )	12.367 (s) ( $\times 2.38$ )	11.175 (s) ( $\times 2.64$ )
32400	47.478 (s) ( $\times 1.00$ )	47.037 (s) ( $\times 1.01$ )	18.457 (s) ( $\times 2.57$ )	16.825 (s) ( $\times 2.82$ )
40000	73.662 (s) ( $\times 1.00$ )	74.188 (s) ( $\times 0.99$ )	27.753 (s) ( $\times 2.65$ )	25.012 (s) ( $\times 2.95$ )

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

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 'MassAssemblingP10ptV0' 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);
```

#### REFERENCES

- [BMP89] C. Bernardi, Y. Maday, and A. Patera. A new nonconforming approach to domain decomposition: the mortar element method. *Nonlinear Partial Differential Equations and their Applications*, 1989.
- [Che11] L. Chen. Programming of Finite Element Methods in Matlab. <http://math.uci.edu/~chenlong/226/Ch3FEMCode.pdf>, 2011.
- [Che13] L. Chen. iFEM, a Matlab software package. <http://math.uci.edu/~chenlong/programming.html>, 2013.
- [Cia02] P. G. Ciarlet. *The Finite Element Method for Elliptic Problems*. SIAM, Philadelphia, 2002.
- [CJS] 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 2d and in 3d. in preparation.
- [CJS12] F. Cuvelier, C. Japhet, and G. Scarella. OptFEM2DP1, a MATLAB/Octave software package codes. <http://www.math.univ-paris13.fr/~cuvelier>, 2012.
- [Cuv08] F. Cuvelier. Méthodes des éléments finis. De la théorie à la programmation. Lecture Notes. <http://www.math.univ-paris13.fr/~cuvelier/docs/poly/polyFEM2D.pdf>, 2008.
- [Dav06] Timothy A. Davis. *Direct Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, 2006.
- [Dav12] Timothy A. Davis. SuiteSparse packages, released 4.0.2. <http://www.cise.ufl.edu/research/sparse/SuiteSparse/>, 2012.
- [Hec12] F. Hecht. Freefem++. <http://www.freefem.org/ff++/index.htm>, 2012.
- [HJ12] A. Hannukainen and M. Juntunen. Implementing the Finite Element Assembly in Interpreted Languages. <http://users.tkk.fi/~mojuntun/preprints/matvecSISC.pdf>, 2012.
- [Joh09] C. Johnson. *Numerical Solution of Partial Differential Equations by the Finite Element Method*. Dover Publications, Inc, 2009.
- [LP98] B. Lucquin and O. Pironneau. *Introduction to Scientific Computing*. John Wiley & Sons Ltd, 1998.
- [Mat12] Matlab. <http://www.mathworks.com/>, 2012.
- [Oct12] GNU Octave. <http://www.gnu.org/software/octave/>, 2012.
- [RV11] T. Rahman and J. Valdman. Fast MATLAB assembly of FEM matrices in 2D and 3D: Nodal elements. *Appl. Math. Comput.*, 2011.