OPTIMISATION DE L'ASSEMBLAGE DE MATRICES ÉLÉMENTS FINIS SOUS MATLAB ET OCTAVE

CUVELIER FRANÇOIS*, JAPHET CAROLINE*†, AND SCARELLA GILLES*

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. Des résultats numériques sont donnés pour illustrer les temps calculs des méthodes proposées.

1. Introduction. Habituellement, les méthodes des éléments finis [?, ?] sont utilisées pour résoudre des équations aux dérivées partielles (EDPs) provenant de divers domaines : mécanique, électromagnétisme... Ces méthodes sont basées sur une discrétisation de la formulation varitionnelle associée à l'EDP et nécessitent l'assemblage de matrices creuses de grandes dimensions (e.g. matrices de masse ou de rigidité). Elles sont bien adaptées à des géométries complexes et à divers conditions aux limites et peuvent être couplées à d'autres discrétisations, en utilisant un couplage faible entre les différents sous-domaines avec des maillages non conformes [?]. Résoudre efficacement ces problèmes nécessite des maillages contenant un grand nombre d'éléments et donc l'assemblage de matrices creuses de très grande dimension.

Matlab [?] et GNU Octave [?] utilisent un même langage de haut niveau pour le calcul numérique. Toutefois, les algorithmes classiques d'assemblage (voir par exemple [?]) implémentés basiquement sous Matlab/Octave se révèlent nettement moins performants que leurs analogues sous d'autres langages.

Dans [?] Section 10, T. Davis décrit différentes techniques d'assemblage appliquées à des matrices aléatoires de type éléments finis sans toutefois effectuer l'assemblage des matrices classiques. Une première technique de vectorisation est proposée dans [?]. D'autres algorithmes plus performants ont été proposés récemment dans [?, ?, ?, ?]. Plus précisément, dans [?], une vectorisation est proposée, basée sur la permutation des deux boucles locales avec celle sur les éléments. Cette technique plus formelle permet d'assembler facilement différents types de matrices en se ramenant à l'élément de référence par transformation affine et en effectuant une intégration numérique. Dans [?], les auteurs ont étendu les opérateurs usuels éléments par éléments sur les tableaux en des opérateurs sur des tableaux de matrices en respectant les règles de l'algèbre linéaire. Grâce à ces nouveaux outils et à l'aide d'une formule de quadrature, l'assemblage de différents types de matrices s'effectue sans boucle. Dans [?], L. Chen construit vectoriellement neuf matrices creuses correspondant aux neuf éléments de la matrice élémentaire, et les somme pour obtenir la matrice globale.

Dans cet article, on présente différentes méthodes permettant d'optimiser, sous Matlab/Octave, l'assemblage de matrices de type éléments finis. On aboutit à une technique d'assemblage entièrement vectorisée (sans boucle) et n'utilisant pas de formule de quadrature. Par souci de concision et sans déroger à la compréhension du mécanisme de base de l'assemblage de matrices éléments finis, on va se focaliser sur les éléments finis P_1-Lagrange en dimension 2 pour l'assemblage de matrices. Cet algorithme s'étend naturellement à des éléments finis P_k-Lagrange, $k \geqslant 2$, et à la dimension 3, voir [?].

On compare les performances de cet algorithme avec celles obtenues en utilisant l'algorithme classique ou ceux proposés dans [?, ?, ?, ?]. On montre également que cette nouvelle implémentation rivalise en performances avec un logiciel dédié tel que FreeFEM++ [?]. L'ensemble des calculs a été effectué sur notre machine de référence [1] avec les versions R2012b de Matlab, 3.6.3 d'Octave et 3.20 de FreeFEM++. Les codes Matlab/Octave sont disponibles en ligne [?].

Le papier est organisé comme suit. Dans la Section ?? on définit les données associées au maillage et trois exemples de matrices d'assemblage. Dans la Section ?? on présente l'algorithme classique d'assemblage et on montre son inefficacité par rapport à FreeFEM++. Puis, dans la Section ??, on explique cette inefficacité inhérente au stockage des matrices creuses sous Matlab/Octave. Dans la Section ?? on présente une première technique de vectorisation, dite "version optimisée 1", suggérée par [?]. Ensuite, dans la Section ?? on présente notre nouvelle technique de vectorisation, la "version optimisée 2", et on compare ses performances avec celles obtenues avec FreeFEM++ et les codes proposés dans [?, ?, ?, ?]. Les listings complets des fonctions utilisées dans les différentes sections sont donnés dans l'Annexe ??.

*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

†INRIA Paris-Rocquencourt, BP 105, 78153 Le Chesnay, France.

1. 2 x Intel Xeon E5645 (6 cœurs) à 2.40Ghz, 32Go RAM, financée par le GNR MoMaS

2. Notations. On utilise une triangulation Ω_h de Ω décrite, entre autres, par les données suivantes :

nom	type	dimension	descriptif
n_q	entier	1	nombre total de nœuds du maillage
n_{me}	entier	1	nombre de triangles
q	réel	$2 \times \mathrm{n_q}$	$\mathrm{q}(\alpha, j)$ est la α-ème coordonnée du j-ème sommet, $\alpha \in \{1, 2\}$ et $j \in \{1, \ldots, \mathrm{n_q}\}$. Le j-ème sommet sera aussi noté q^j avec $\mathrm{q}_x^j = \mathrm{q}(1, j)$ et $\mathrm{q}_y^j = \mathrm{q}(2, j)$
me	entier	$3 \times \mathrm{n_{me}}$	tableau de connectivité. $\mathrm{me}(\beta, k)$ indice de stockage, dans le tableau q, du β-ème sommet du triangle d'indice k, $\beta \in \{1, 2, 3\}$ et $k \in \{1, \ldots, \mathrm{n_{me}}\}$.
areas	réel	$1 \times \mathrm{n_{me}}$	$\mathrm{areas}(k)$ aire du k-ème triangle, $k \in \{1, \ldots, \mathrm{n_{me}}\}$.

Dans cet article on va considérer l'assemblage des matrices dites de masse, masse avec poids et rigidité, notées respectivement $\mathbb{M}$, $\mathbb{M}^{[w]}$ et $\mathbb{S}$. Ces matrices d'ordre $\mathrm{n_q}$ sont creuses, et leurs coefficients sont définis par

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

où $\{\varphi_i\}_{1 \leqslant i \leqslant n_q}$ sont les fonctions de base classiques, w une fonction définie sur Ω et $\langle \cdot, \cdot \rangle$ le produit scalaire usuel dans $\mathbb{R}^2$. Plus de détails sont donnés dans [**?**]. Pour assembler l'une de ces matrices, il suffit de connaître sa matrice élémentaire associée. Sur un triangle T de sommets locaux $\tilde{\mathrm{q}}^1$, $\tilde{\mathrm{q}}^2$, $\tilde{\mathrm{q}}^3$ et d'aire $|T|$, la matrice élémentaire de masse est définie par

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

En posant $\tilde{w}_\alpha = w(\tilde{\mathrm{q}}^\alpha)$, $\forall \alpha \in \{1, \ldots, 3\}$, une approximation de la matrice élémentaire de masse avec poids s'écrit

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

En notant $\boldsymbol{u} = \tilde{\mathrm{q}}^2 - \tilde{\mathrm{q}}^3$, $\boldsymbol{v} = \tilde{\mathrm{q}}^3 - \tilde{\mathrm{q}}^1$ et $\boldsymbol{w} = \tilde{\mathrm{q}}^1 - \tilde{\mathrm{q}}^2$, la matrice élémentaire de rigidité est donnée par

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

On décrit maintenant l'algorithme d'assemblage classique en utilisant ces matrices élémentaires et une boucle principale portant sur les triangles.

3. Présentation de l'algorithme classique. Dans l'algorithme qui suit, on donne l'assemblage d'une matrice M générique à partir de sa matrice élémentaire E donnée par la fonction ElemMat (supposée connue).

Listing 1
Assemblage basique

```
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
```

On va comparer les performances de cet algorithme à celles obtenues pour l'assemblage des différentes matrices avec FreeFEM++ [**?**], dont le code est donné dans le Listing **??**. Sur la Figure **??**, on représente les temps de calcul (en secondes) des codes en fonction du nombre de nœuds n_q du maillage, avec l'algorithme basique et avec FreeFEM++. Les valeurs des temps de calculs sont données en Annexe **??**. On note que les performances des codes Matlab/Octave sont en $\mathcal{O}(\mathrm{n_q}^2)$ alors que, sous FreeFEM++, on semble être en $\mathcal{O}(\mathrm{n_q})$.

Listing 2
Algorithme d'assemblage avec 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) );
// Assembly
matrix M= vMass(Vh,Vh);          // Build mass matrix
matrix Mw = vMassW(Vh,Vh);       // Build weighted mass matrix
matrix S = vStiff(Vh,Vh);        // Build stiffness matrix
```

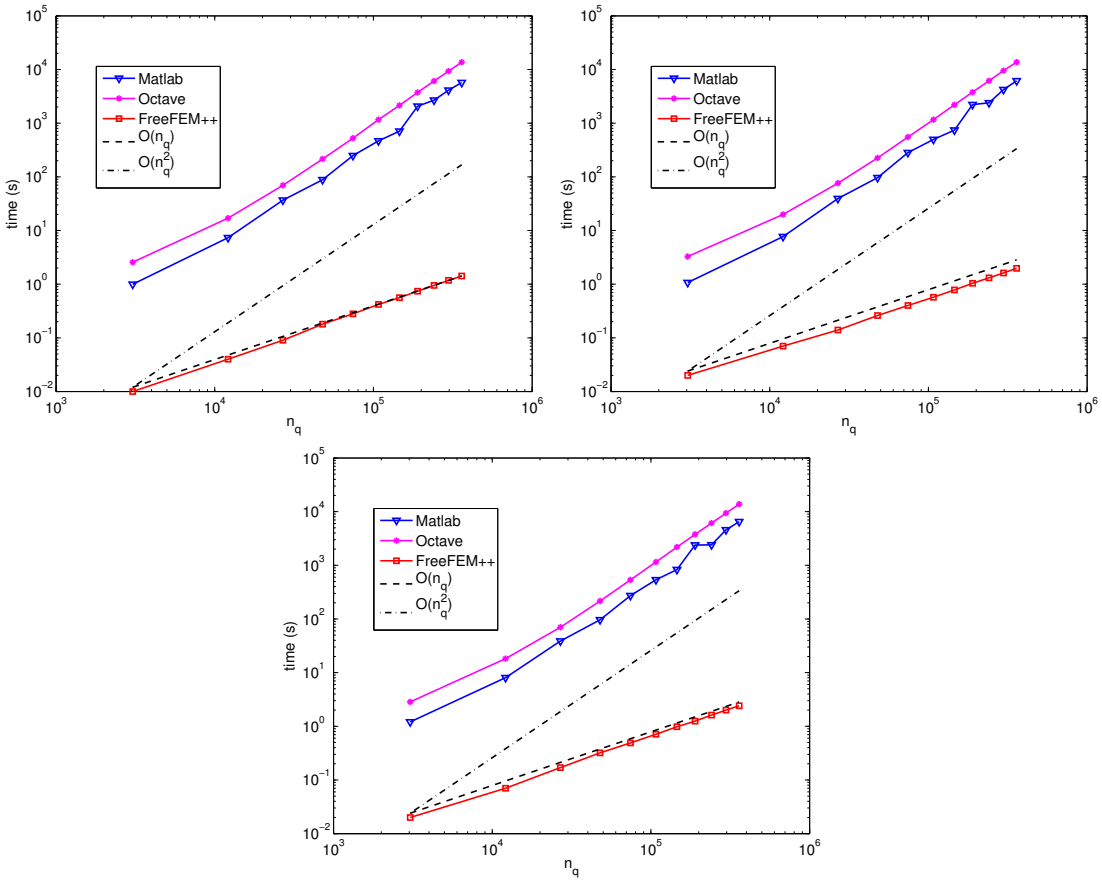

Figure 3.1. *Comparaison des codes basiques d'assemblage écrits en Matlab/Octave avec FreeFEM++, pour les matrices de masse (en haut à gauche), masse avec poids (en haut à droite) et rigidité (en bas).*

Il est aussi surprenant de noter que les performances sous Matlab peuvent être améliorées en utilisant une version antérieure (voir l'Annexe **??**) !

L'objectif des optimisations que l'on va proposer est d'obtenir une efficacité comparable (en temps de calcul) à celle de FreeFEM++. Pour tenter d'améliorer les performances des fonctions d'assemblage basiques (Listing **??**), on peut vectoriser les deux boucles locales pour obtenir le code donné par le Listing **??** :

Listing 3
Assemblage optimisé version 0

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

Toutefois, les performances de ce code restent en $\mathcal{O}(\mathrm{n_q}^2)$ et le code n'est donc pas suffisamment efficace.

Dans le paragraphe suivant, on explique le stockage des matrices creuses sous Matlab/Octave en vue de justifier cette perte d'efficacité.

4. Stockage des matrices creuses. Sous Matlab et Octave, une matrice creuse (ou "sparse matrix"), $\mathbb{A} \in \mathcal{M}_{M,N}(\mathbb{R})$, est stockée sous le format CSC (Compressed Sparse Column), utilisant trois tableaux :

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

où nnz est le nombre d'éléments non nuls de la matrice $\mathbb{A}$. Ces tableaux sont définis par

- aa : contient l'ensemble des nnz éléments non nuls de $\mathbb{A}$ stockés par colonnes.
- ia : contient les numéros de ligne des éléments stockés dans le tableau aa.
- ja : permet de retrouver les éléments d'une colonne de $\mathbb{A}$, sachant que le premier élément non nul de la colonne k de $\mathbb{A}$ est en position $ja(k)$ dans le tableau aa. On a $ja(1) = 1$ et $ja(N+1) = nnz + 1$.

Par exemple, pour la matrice

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

on a $M = 3$, $N = 4$, $nnz = 6$ et

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

Le premier élément non nul de la colonne $k = 3$ de $\mathbb{A}$ est 2, ce nombre est en position 4 dans aa, donc $ja(3) = 4$.

Regardons les opérations à effectuer sur les tableaux aa, ia et ja si l'on modifie la matrice $\mathbb{A}$ en prenant $\mathbb{A}(1,2) = 8$. Celle-ci devient

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

Dans cette opération un élément nul de $\mathbb{A}$ a été remplacé par une valeur non nulle 8 : il va donc falloir la stocker dans les tableaux sachant qu'aucune place n'est prévue. On suppose que les tableaux sont suffisamment grands (pas de souci mémoire), il faut alors décaler d'une case l'ensemble des valeurs des tableaux aa et ia à partir de la 3ème position puis copier la valeur 8 en $aa(3)$ et le numéro de ligne 1 en $ia(3)$:

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

Pour le tableau ja, il faut à partir du numéro de colonne 2 plus un, incrémenter de +1 :

ja	1	2	5	6	8

La répétition de ces opérations est coûteuse lors de l'assemblage de la matrice dans les algorithmes précédents (et on ne parle pas ici des problèmes de réallocation dynamique qui peuvent arriver !).

On présente maintenant la version 1 du code optimisé qui va nous permettre d'améliorer les performances du code basique.

5. Code optimisé : version 1. On va utiliser l'appel suivant de la fonction `sparse` :

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

Cette commande génère une matrice creuse m par n telle que M(I(k),J(k)) = K(k). Les vecteurs I, J et K ont la même longueur. Tous les éléments nuls de K sont ignorés et tous les éléments de K ayant les mêmes indices dans I et J sont sommés.

L'idée est donc de créer trois tableaux globaux $\boldsymbol{I}_g$, $\boldsymbol{J}_g$ et $\boldsymbol{K}_g$ permettant de stocker l'ensemble des matrices élémentaires ainsi que les positions de leurs éléments dans la matrice globale. Chaque tableau est de dimension $9\mathrm{n_{me}}$. Une fois ces tableaux créés, la matrice d'assemblage s'obtient alors par la commande

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

Pour générer ces trois tableaux, on commence par définir trois tableaux locaux $\boldsymbol{K}_k^e$, $\boldsymbol{I}_k^e$ et $\boldsymbol{J}_k^e$ de 9 éléments obtenus à partir d'une matrice élémentaire $\mathbb{E}(T_k)$ d'ordre 3 :

$\boldsymbol{K}_k^e$: ensemble des éléments de la matrice $\mathbb{E}(T_k)$ stockés colonne par colonne,
$\boldsymbol{I}_k^e$: indices globaux de ligne associés aux éléments stockés dans $\boldsymbol{K}_k^e$,
$\boldsymbol{J}_k^e$: indices globaux de colonne associés aux éléments stockés dans $\boldsymbol{K}_k^e$.

On a choisi un stockage des tableaux par colonne dans Matlab/Octave, mais par souci de concision, on les représente sous forme de ligne,

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

$\boldsymbol{K}_k^e$: | $e^k_{1,1}$ | $e^k_{2,1}$ | $e^k_{3,1}$ | $e^k_{1,2}$ | $e^k_{2,2}$ | $e^k_{3,2}$ | $e^k_{1,3}$ | $e^k_{2,3}$ | $e^k_{3,3}$ |

$\boldsymbol{I}_k^e$: | i^k_1 | i^k_2 | i^k_3 | i^k_1 | i^k_2 | i^k_3 | i^k_1 | i^k_2 | i^k_3 |

$\boldsymbol{J}_k^e$: | i^k_1 | i^k_1 | i^k_1 | i^k_2 | i^k_2 | i^k_2 | i^k_3 | i^k_3 | i^k_3 |

avec $i^k_1 = \mathrm{me}(1,k)$, $i^k_2 = \mathrm{me}(2,k)$, $i^k_3 = \mathrm{me}(3,k)$.

Sous Matlab/Octave, pour générer les trois tableaux $\boldsymbol{K}_k^e$, $\boldsymbol{I}_k^e$ et $\boldsymbol{J}_k^e$, on peut utiliser les commandes suivantes :

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

A partir de ces tableaux, il est alors possible de construire les trois tableaux globaux $\boldsymbol{I}_g$, $\boldsymbol{J}_g$ et $\boldsymbol{K}_g$, de dimension $9\mathrm{n_{me}}$, définis $\forall k \in \{1,\ldots,\mathrm{n_{me}}\}$, $\forall il \in \{1,\ldots,9\}$ par

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

On représente, en Figure **??**, l'insertion du tableau local $\boldsymbol{K}_k^e$ dans le tableau global $\boldsymbol{K}_g$ sachant que l'opération est identique pour les deux autres tableaux.

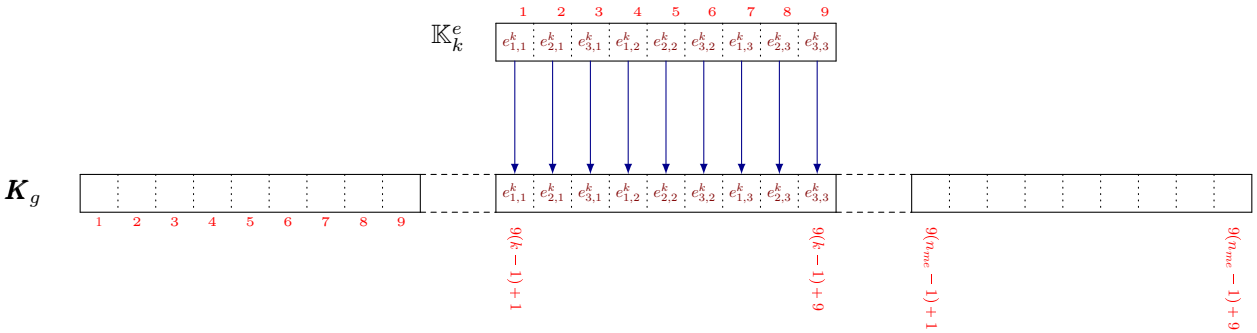

FIGURE 5.1. *Insertion d'une matrice élémentaire - Version 1*

Voici le code Matlab/Octave générique associé à cette technique où les tableaux globaux $\boldsymbol{I}_g$, $\boldsymbol{J}_g$ et $\boldsymbol{K}_g$ sont stockés sous forme de vecteurs colonnes :

LISTING 4
Assemblage optimisé 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);
```

Les listings complets sont donnés dans l'Annexe **??**. En Figure **??**, on représente les temps de calcul des codes sous Matlab et Octave avec leur analogue sous FreeFEM++, en fonction de la taille du maillage (disque unité).

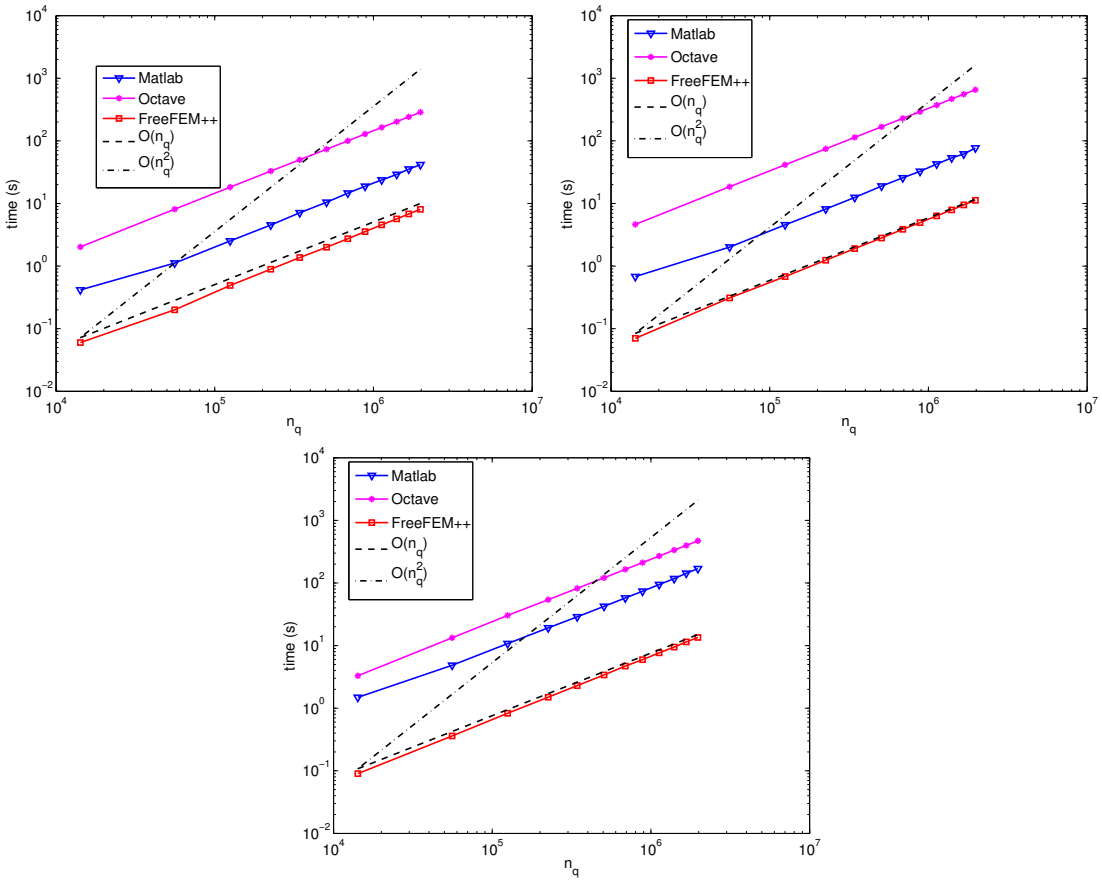

FIGURE 5.2. *Comparaison des codes d'assemblage :* `OptV1` *en Matlab/Octave et FreeFEM++, pour les matrices de masse (en haut à gauche), masse avec poids (en haut à droite) et rigidité (en bas).*

Les performances des codes Matlab/Octave sont maintenant, comme pour FreeFEM++, en $\mathcal{O}(\mathrm{n_q})$. Toutefois, ce dernier reste toujours nettement plus performant (d'un facteur environ 5 pour la matrice de masse, 6.5 pour la matrice de masse avec poids et 12.5 pour la matrice de rigidité), voir l'Annexe **??**.

Afin d'améliorer encore l'efficacité des codes, on introduit maintenant une seconde version optimisée de l'assemblage.

6. Code optimisé : version 2. On présente la version 2 du code optimisé où l'on s'affranchit totalement de boucle.

Pour mieux appréhender la méthode, on construit trois tableaux permettant de stocker l'intégralité des matrices élémentaires ainsi que les positions dans la matrice globale. On note $\mathbb{K}_g$, $\mathbb{I}_g$ et $\mathbb{J}_g$ ces trois tableaux (9 lignes et $\mathrm{n_{me}}$ colonnes) définis $\forall k \in \{1,\dots,\mathrm{n_{me}}\}$, $\forall il \in \{1,\dots,9\}$ par

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

Les trois tableaux locaux $\boldsymbol{K}_k^e$, $\boldsymbol{I}_k^e$ et $\boldsymbol{J}_k^e$ sont donc stockés en colonne k respectivement dans les tableaux globaux $\mathbb{K}_g$, $\mathbb{I}_g$ et $\mathbb{J}_g$.

La méthode naturelle de construction de ces trois tableaux consiste en une boucle sur les triangles T_k et en l'insertion colonne par colonne des tableaux locaux : cette opération est représentée en Figure **??**.

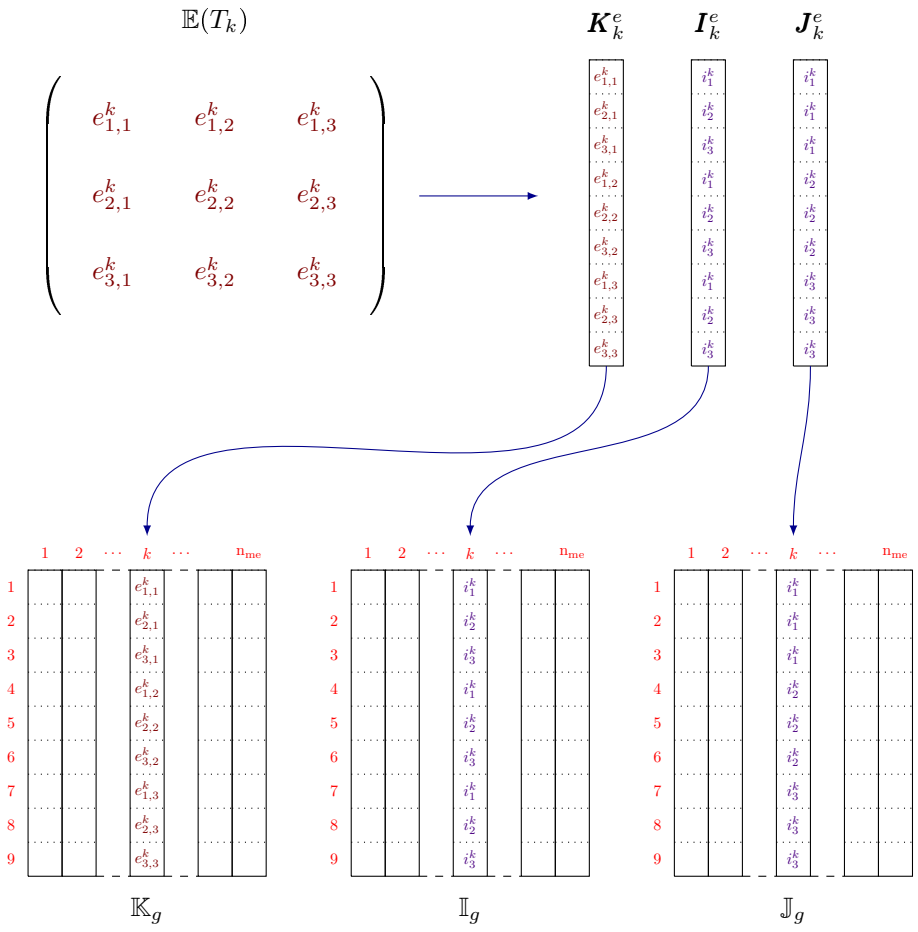

FIGURE 6.1. *Insertion d'une matrice élémentaire - Version 2*

Une fois ces trois tableaux construits, la matrice d'assemblage s'obtient alors par la commande Matlab/Octave

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

Il faut noter que les matrices d'indices globaux $\mathbb{I}_g$ et $\mathbb{J}_g$ peuvent se construire **sans boucle** sous Matlab/Octave. On présente à gauche le code basique pour la création de ces deux matrices et à droite la version vectorisée :

```
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],:);
```

Il nous reste à vectoriser le code correspondant au calcul du tableau $\mathbb{K}_g$. La version basique du code, qui correspond à une construction colonne par colonne, est :

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

Pour vectoriser ce code, on va construire le tableau $\mathbb{K}_g$ **ligne par ligne**. Ceci correspond à la permutation de la boucle sur les éléments avec les boucles locales dans l'algorithme classique. Cette vectorisation est différente de celle proposée dans [**?**] dans le sens où elle n'utilise pas de formule de quadrature et diffère des codes de L. Chen [**?**] par la vectorisation complète des tableaux $\mathbb{I}_g$ et $\mathbb{J}_g$.

Dans la suite, on décrit cette construction pour chacune des matrices étudiées.

6.1. Matrice de masse. La matrice de masse élémentaire associée au triangle T_k est donnée par (**??**). Le tableau $\mathbb{K}_g$ est donc défini par : $\forall k \in \{1, \ldots, \mathrm{n_{me}}\}$,

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

On construit alors deux tableaux A_6 et A_{12} de dimension $1 \times \mathrm{n_{me}}$ tels que $\forall k \in \{1, \ldots, \mathrm{n_{me}}\}$:

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

Les lignes $\{1, 5, 9\}$ du tableau $\mathbb{K}_g$ correspondent à A_6 et les lignes $\{2, 3, 4, 6, 7, 8\}$ à A_{12}. Une représentation est donnée en Figure **??**.

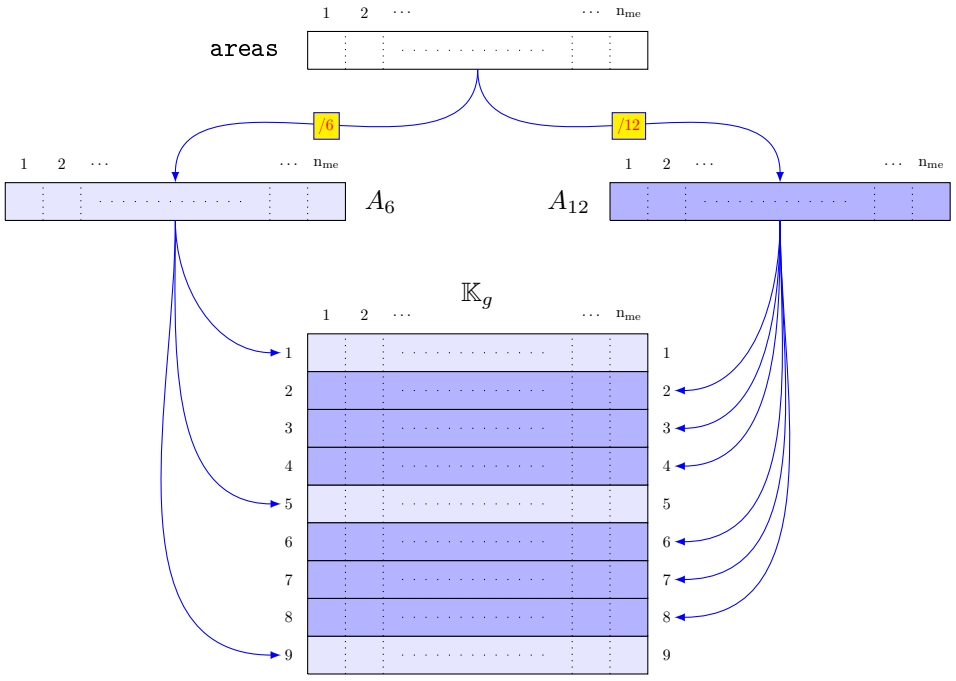

FIGURE 6.2. *Assemblage de la matrice de masse - Version 2*

Voici le code Matlab/Octave associé à cette technique :

LISTING 5
MassAssemblingP1OptV2.m

```
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);
```

6.2. Matrice de masse avec poids. Les matrices élémentaires de masse avec poids $\mathbb{M}^{e,[w_h]}(T_k)$ sont données par (**??**). On note $\boldsymbol{T_w}$ le tableau de dimension $1 \times \mathrm{n_q}$ défini par $\boldsymbol{T_w}(i) = w(\mathrm{q}^i)$, $\forall i \in \{1, \ldots, \mathrm{n_q}\}$ et $\boldsymbol{W}_1$, $\boldsymbol{W}_2$, $\boldsymbol{W}_3$ trois tableaux de dimension $1 \times \mathrm{n_{me}}$ définis pour tout $k \in \{1, \ldots, \mathrm{n_{me}}\}$ par

$$\boldsymbol{W}_1(k) = \frac{|T_k|}{30}\boldsymbol{T_w}(\mathrm{me}(1,k)), \ \boldsymbol{W}_2(k) = \frac{|T_k|}{30}\boldsymbol{T_w}(\mathrm{me}(2,k)) \text{ et } \boldsymbol{W}_3(k) = \frac{|T_k|}{30}\boldsymbol{T_w}(\mathrm{me}(3,k)).$$

Avec ces notations, on a

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

Pour le calcul de ces trois tableaux, on donne un code non vectorisé (à gauche) et une version vectorisée (au milieu) qui peut être réduite à une ligne (à droite) :

```
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
```

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

```
W=Tw(me).*(ones(3,1)*areas/30);
```

Ici W est une matrice de taille $3 \times \mathrm{n_{me}}$, dont la ℓ-ème ligne est $\boldsymbol{W}_\ell$, $1 \leqslant \ell \leqslant 3$.
On reprend le mécanisme de la Figure **??** pour la construction de la matrice $\mathbb{K}_g$. On doit donc vectoriser le code suivant :

```
Kg=zeros(9,nme);
for k=1:nme
  Me=ElemMassWMat(areas(k),Tw(me(:,k)));
  Kg(:,k)=Me(:);
end
```

Pour cela, soient $\boldsymbol{K}_1$, $\boldsymbol{K}_2$, $\boldsymbol{K}_3$, $\boldsymbol{K}_5$, $\boldsymbol{K}_6$, $\boldsymbol{K}_9$ les six tableaux de dimension $1 \times \mathrm{n_{me}}$ définis par

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

La matrice élémentaire de masse avec poids et la colonne k de la matrice $\mathbb{K}_g$ sont donc respectivement :

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

On en déduit la version vectorisée du calcul de $\mathbb{K}_g$:

```
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];
```

Une représentation de cette technique est donnée en Figure **??**.

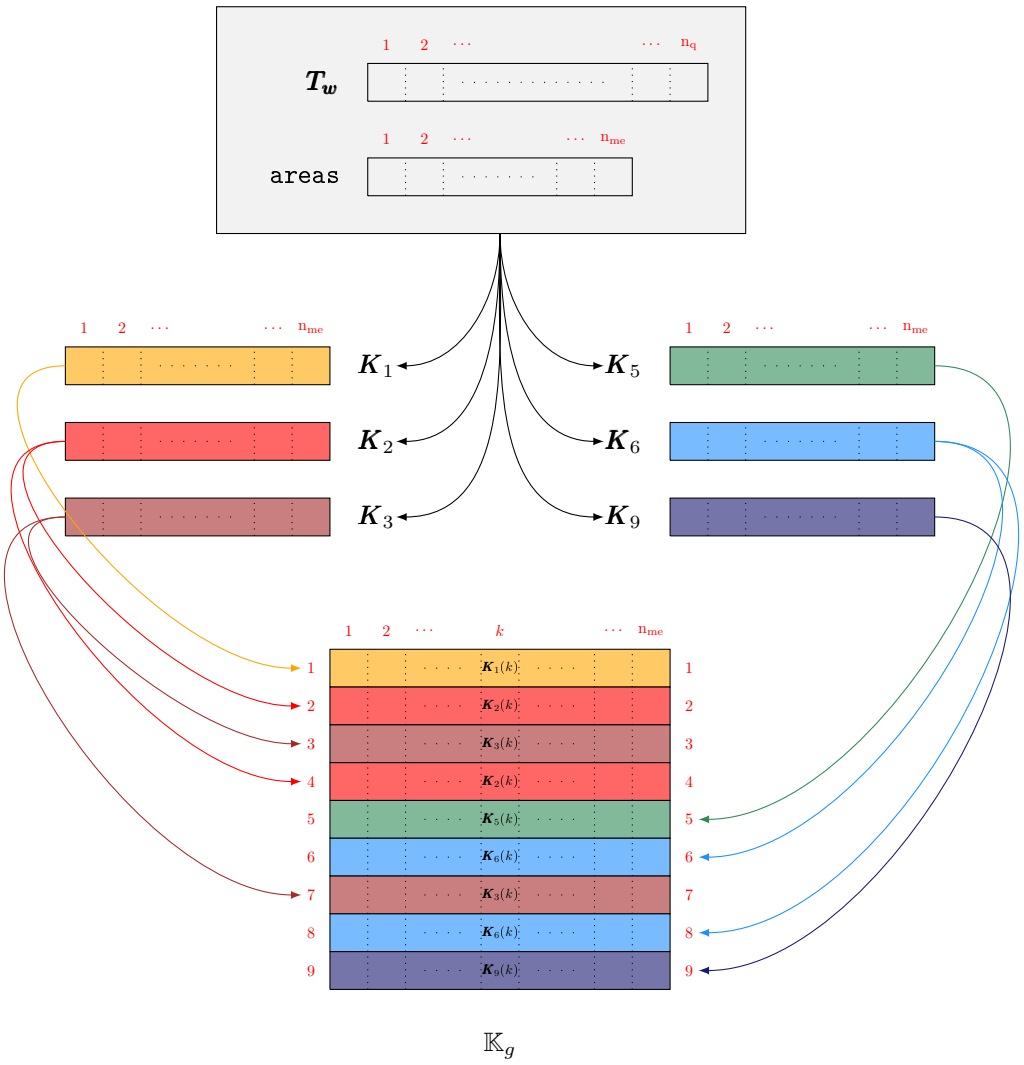

FIGURE 6.3. *Assemblage de la matrice de masse avec poids - Version 2*

Au final, voici la fonction complète vectorisée utilisant la symétrie de la matrice élémentaire :

LISTING 6
MassWAssemblingP1OptV2.m

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

6.3. Matrice de rigidité. Les trois sommets du triangle T_k sont $\mathrm{q}^{\mathrm{me}(1,k)}, \mathrm{q}^{\mathrm{me}(2,k)}$ et $\mathrm{q}^{\mathrm{me}(3,k)}$. On note $\boldsymbol{u}^k = \mathrm{q}^{\mathrm{me}(2,k)} - \mathrm{q}^{\mathrm{me}(3,k)}$, $\boldsymbol{v}^k = \mathrm{q}^{\mathrm{me}(3,k)} - \mathrm{q}^{\mathrm{me}(1,k)}$ et $\boldsymbol{w}^k = \mathrm{q}^{\mathrm{me}(1,k)} - \mathrm{q}^{\mathrm{me}(2,k)}$. Avec ces notations, la matrice

élémentaire de rigidité associée au triangle T_k est donnée par

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

On note $\boldsymbol{K}_1$, $\boldsymbol{K}_2$, $\boldsymbol{K}_3$, $\boldsymbol{K}_5$, $\boldsymbol{K}_6$ et $\boldsymbol{K}_9$ les six tableaux de dimension $1 \times \mathrm{n_{me}}$ tels que, $\forall k \in \{1, \ldots, \mathrm{n_{me}}\}$,

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

Si l'on construit ces tableaux, alors le code d'assemblage vectorisé correspond à la Figure **??** et s'écrit

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

On décrit maintenant la construction vectorisée de ces six tableaux. Pour cela, on introduit les tableaux $\boldsymbol{q}_\alpha \in \mathcal{M}_{2,\mathrm{n_{me}}}(\mathbb{R})$, $\alpha \in \{1, \ldots, 3\}$, contenant l'ensemble des coordonnées des sommets α des triangles T_k :

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

On présente à gauche une version non vectorisée de création de ces trois tableaux et à droite la version vectorisée :

```
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,:));
```

On obtient alors facilement les trois tableaux $\boldsymbol{u}$, $\boldsymbol{v}$ et $\boldsymbol{w}$ de dimension $2 \times \mathrm{n_{me}}$ ayant respectivement pour colonne k, $\mathrm{q}^{\mathrm{me}(2,k)} - \mathrm{q}^{\mathrm{me}(3,k)}$, $\mathrm{q}^{\mathrm{me}(3,k)} - \mathrm{q}^{\mathrm{me}(1,k)}$ et $\mathrm{q}^{\mathrm{me}(1,k)} - \mathrm{q}^{\mathrm{me}(2,k)}$.

Le code correspondant est

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

Ensuite, l'opérateur $.*$ (multiplication éléments par éléments entre tableaux) et la fonction **sum**(.,1) (sommation uniquement suivant les lignes) permettent le calcul de différents tableaux. Par exemple, le calcul de $\boldsymbol{K}_2$ s'effectue vectoriellement avec le code suivant

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

Au final, voici la fonction complète vectorisée utilisant la symétrie de la matrice élémentaire :

Listing 7
StiffAssemblingP1OptV2.m

```
function R=StiffAssemblingP1OptV2(nq,nme,q,me,areas)
Ig = me([1 2 3 1 2 3 1 2 3],:);
Jg = me([1 1 1 2 2 2 3 3 3],:);

q1 =q(:,me(1,:)); q2 =q(:,me(2,:)); q3 =q(:,me(3,:));
u = q2-q3; v=q3-q1; w=q1-q2;
clear q1 q2 q3
areas4=4*areas;
Kg=zeros(9,nme);
Kg(1,:)=sum(u.*u,1)./areas4; % K1 ou G11
Kg(2,:)=sum(v.*u,1)./areas4; % K2 ou G12
Kg(3,:)=sum(w.*u,1)./areas4; % K3 ou G13
Kg(5,:)=sum(v.*v,1)./areas4; % K5 ou G22
Kg(6,:)=sum(w.*v,1)./areas4; % K6 ou G23
Kg(9,:)=sum(w.*w,1)./areas4; % K9 ou G33
Kg([4, 7, 8],:)=Kg([2, 3, 6],:);
R = sparse(Ig(:),Jg(:),Kg(:),nq,nq);
```

6.4. Résultats numériques. Dans cette partie on compare les performances des codes `OptV2` avec celle de FreeFEM++ et celle des codes donnés dans [?, ?, ?, ?]. Le domaine Ω est le disque unité.

6.4.1. Comparaison avec FreeFEM++. Sur la Figure ??, on compare les temps de calcul des codes `OptV2` sous Matlab et Octave avec leurs analogues sous FreeFEM++, en fonction de la taille du maillage. On

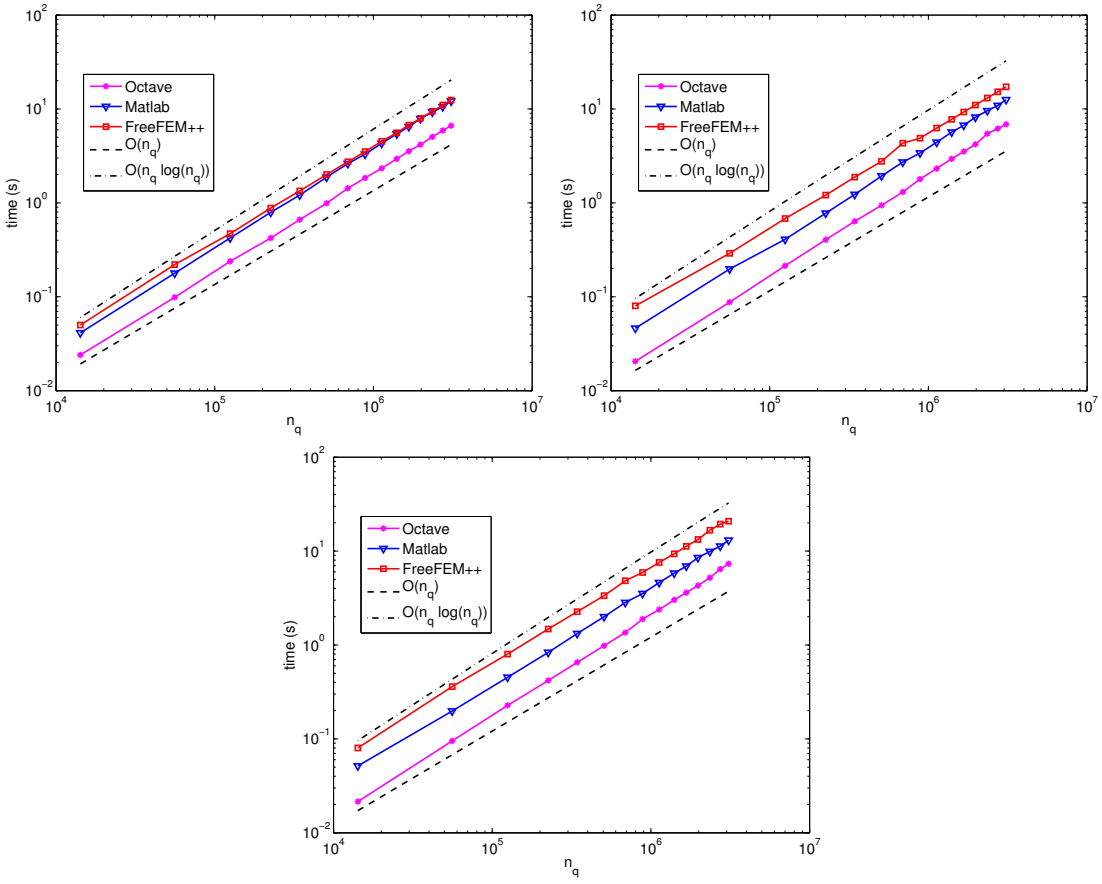

Figure 6.4. *Comparaison des codes d'assemblage : `OptV2` en Matlab/Octave et FreeFEM++, pour les matrices de masse (en haut à gauche), masse avec poids (en haut à droite) et rigidité (en bas).*

donne en Annexe ?? les tableaux correspondant à la Figure ??.
Les performances des codes Matlab/Octave restent en $\mathcal{O}(n_q)$ et elles sont légèrement meilleures que celle de FreeFEM++. De plus, et uniquement avec le code `OptV2`, Octave prend le dessus sur Matlab. Dans les autres

versions des codes, non totalement vectorisées, le JIT-Accelerator (Just-In-Time) de Matlab permet des performances nettement supérieures à Octave.

En outre, on peut améliorer les performances sous Matlab en utilisant la **SuiteSparse** de T. Davis [?], sachant que celle-ci est nativement utilisée sous Octave. Dans nos codes, l'utilisation de la fonction `cs_sparse` de la **SuiteSparse** à la place de la fonction `sparse` de Matlab permet d'améliorer les performances d'environ un facteur 1.1 pour la version `OptV1` et 2.5 pour la version `OptV2`.

6.4.2. Comparaison avec les codes de [?, ?, ?, ?]. On compare ici les codes d'assemblage des matrices de Masse et Rigidité proposés par T. Rahman et J. Valdman [?], par A. Hannukainen et M. Juntunen [?] et par L. Chen [?, ?] avec la version `OptV2` développée dans cet article. En Figure ?? (avec Matlab) et en Figure ?? (avec Octave), on représente les temps calcul en fonction du nombre de sommets du maillage (disque unité), pour ces différents codes. Les valeurs associées sont données dans les Tables ?? à ??. Pour de grandes matrices creuses, la version `OptV2` permet des gains de performance de l'ordre de 5% à 20%, par rapport aux autres codes vectorisés (pour des maillages suffisamment grands).

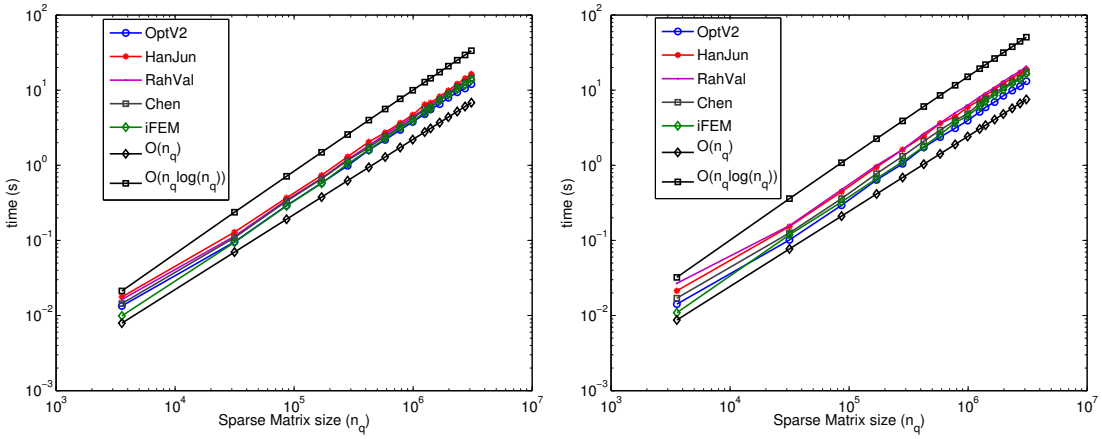

FIGURE 6.5. *Comparaison des codes d'assemblage sous Matlab R2012b : `OptV2` et [?, ?, ?, ?], pour les matrices de masse (à gauche) et de rigidité (à droite).*

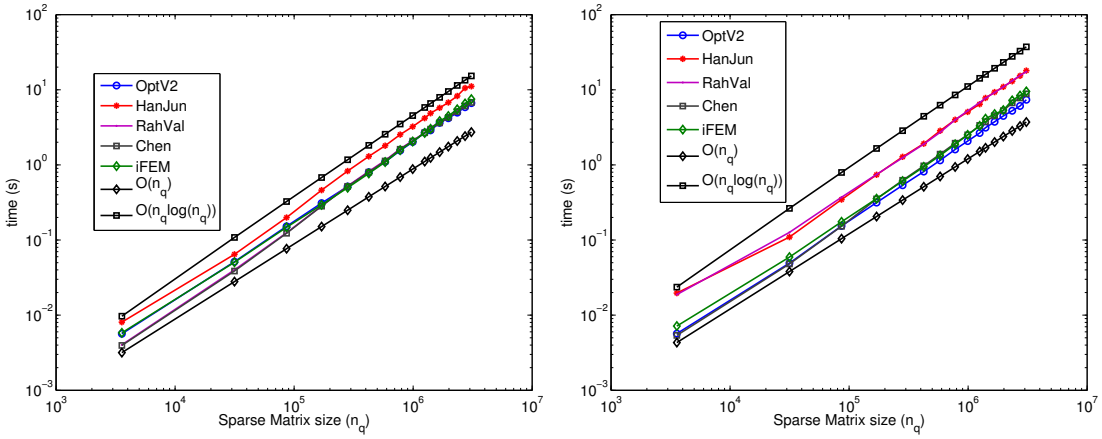

FIGURE 6.6. *Comparaison des codes d'assemblage sous Octave 3.6.3 : `OptV2` et [?, ?, ?, ?], pour les matrices de masse (à gauche) et de rigidité (à droite).*

7. Conclusion. On a progressivement construit, pour trois exemples de matrices, les codes d'assemblage totalement vectorisés à partir d'un code basique et décrit, pour chaque version, les techniques s'y affairant et les gains obtenus. L'assemblage de matrices d'ordre 10^6, sur notre machine de référence, est alors obtenu en moins de 4 secondes (resp. environ 2 secondes) avec Matlab (resp. avec Octave).

Ces techniques d'optimisations sous Matlab/Octave peuvent s'étendre à d'autres types de matrices, d'éléments finis (P_k, Q_k, ...) et en 3D. En mécanique, les mêmes techniques ont été utilisées pour l'assemblage de matrices de rigidité élastique en dimension 2 et les gains obtenus sont du même ordre de grandeur.

De plus, sous Matlab, il est possible d'augmenter les performances des codes `OptV2` en utilisant une carte GPU. Les premiers essais effectués indiquent des temps de calculs divisés par 6 (par rapport à `OptV2` sans 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 [?, ?, ?, ?] (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 [?, ?, ?, ?] (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 [?, ?, ?, ?] (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 [?, ?, ?, ?] (columns 3-6) : time in seconds (top value) and speedup (bottom value). The speedup reference is* `OptV2` *version.*

Annexe A. Comparaison des performances de codes Matlab/Octave avec FreeFEM++.

A.1. Code classique 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. Code OptV0 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. Code OptV1 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. Code OptV2 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* 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.*

Annexe B. Codes.

B.1. Matrices élémentaires.

LISTING 8
ElemMassMatP1.m

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

LISTING 9
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 10
ElemStiffMatP1.m

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

B.2. Code classique.

LISTING 11
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 12
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 13
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. Codes optimisés - Version 0.

Listing 14
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 15
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 16
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. Codes optimisés - Version 1.

Listing 17
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 18
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 19
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);
```

B.5. Codes optimisés - Version 2.

Listing 20
MassAssemblingP1OptV2.m

```
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);
```

Listing 21
MassWAssemblingP1OptV2.m

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

Listing 22
StiffAssemblingP1OptV2.m

```
function R=StiffAssemblingP1OptV2(nq,nme,q,me,areas)
Ig = me([1 2 3 1 2 3 1 2 3],:);
Jg = me([1 1 1 2 2 2 3 3 3],:);

q1 =q(:,me(1,:)); q2 =q(:,me(2,:)); q3 =q(:,me(3,:));
u = q2-q3; v=q3-q1; w=q1-q2;
clear q1 q2 q3
areas4=4*areas;
Kg=zeros(9,nme);
Kg(1,:)=sum(u.*u,1)./areas4;
Kg(2,:)=sum(v.*u,1)./areas4;
Kg(3,:)=sum(w.*u,1)./areas4;
Kg(5,:)=sum(v.*v,1)./areas4;
Kg(6,:)=sum(w.*v,1)./areas4;
Kg(9,:)=sum(w.*w,1)./areas4;
Kg([4, 7, 8],:)=Kg([2, 3, 6],:);
R = sparse(Ig,Jg,Kg,nq,nq);
```

Annexe C. Matlab sparse trouble. Dans cette partie, on illustre un problème rencontré lors du développement de nos codes : baisse des performances de l'assemblage par les fonctions de base et optimisées OptV0 lors du passage du release R2011b au release R2012a ou R2012b. En fait, ceci provient de l'usage de la commande M = **sparse**(nq,nq). Pour l'illustrer, on donne, dans le tableau **??**, les temps cpu d'assemblage de la matrice de masse par la fonction MassAssemblingP1OptV0 pour différentes versions de Matlab.

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

Ce problème a été notifié à l'équipe de développement de MathWorks :

```
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.
```

Pour corriger ce problème dans les versions R2012a et R2012b, il est préconisé, par le support technique de Matlab, d'utiliser la fonction **spalloc** à la place de la fonction **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);
```