Assembling sparse matrices in MATLAB

Sergio Zlotnik[1] and Pedro Díez[2,*,†]

[1]*Group of Dynamics of the Lithosphere (GDL) Institute of Earth Sciences 'Jaume Almera', CSIC Lluís Solé i Sabarís s/n, 08028 Barcelona, Spain*
[2]*Laboratori de Càlcul Numèric, Departament de Matemàtica Aplicada III, Universitat Politècnica de Catalunya Campus Nord UPC, 08034 Barcelona, Spain*

SUMMARY

The assembly of sparse matrices is a key operation in finite element methods. In this study we analyze several factors that may have an influence on the efficiency of the assembly procedure.

Different insertion strategies are compared using two metrics: a Cost function (the number of memory movements) and actual computing time. An improved algorithm implemented in MATLAB is proposed. It reduces both memory operations and computing time for all tested cases.

The efficiency of the assembly process is found to be highly dependent on node and element numbering. The effect of the classic reverse Cuthill–McKee algorithm is, in most cases, positive and reduces computation costs.

Finally, the case where a sparse matrix has to be re-assembled at each time step is studied. The efficiency of the assembly is improved if the matrix pattern is entirely or partially inherited from previous steps.

KEY WORDS: sparse matrix; finite elements; assembly

1. INTRODUCTION

The implementation of finite element methods involves assembling elemental matrices into global sparse matrices. Despite the great effort that has been done to optimize sparse algorithms and operations, details of how to set a sparse matrix are usually not mentioned in the literature. Nevertheless, the matrix assembly is a time-consuming process, which under unfavorable conditions, can overcome the computing time of the sparse solver.

One of the scenarios, where the assembly is time consuming, is the solution of transient multiphase Stokes flow. In these kind of problems, matrices have to be assembled in each time step and the complete pattern of the matrices is unknown. When partition of the unity methods are used to enrich the discretization, degrees of freedom are created and destroyed dynamically at each time step. In practice, this increases the assembly time dramatically. In fact, the present study is motivated by the problem encountered in the implementation of one of these methods [1].

MATLAB is a programming platform frequently used to implement finite element codes for both research and engineering practice. The remainder of this paper is devoted to analyze both

*Correspondence to: Pedro Díez, Laboratori de Càlcul Numèric, Departament de Matemàtica Aplicada III, Universitat Politècnica de Catalunya Campus Nord UPC, 08034 Barcelona, Spain.
†E-mail: pedro.diez@upc.es

Contract/grant sponsor: Ministerio de Educación y Ciencia; contract/grant numbers: DPI2004-03000, CTM2005-08071-C03-03/MAR, CSD2006-00041

from the theoretical and practical viewpoint the assembly procedure in MATLAB. Moreover, the best option found is provided as open code that reduces the assembly time in all the cases.

2. STORAGE AND CREATION OF SPARSE MATRICES

In MATLAB, sparse matrices are stored in the classical compressed sparse row (CSR) format [2]. The CSR data structure is based on three arrays: `pr`, `ir`, and `jc`. A matrix K composed of m rows and n columns, with `nnz` non-zero coefficients is stored in three arrays: (1) the real array `pr` with length `nnz` containing the non-zero values a_{ij} stored by column, (2) the integer array `ir` with length `nnz` containing the row indices of the elements of `pr`, and (3) the integer array `jc` with length $n+1$ containing the pointers to the beginning of each row in the arrays `pr` and `ir`. The last position of `jc` contains the number `nnz`.

This means that, in CSR storage, the non-zero values a_{ij} are stored in `pr` sorted by columns. When a new value is added it has to be inserted in its corresponding place. This can lead to moving large chunks of memory if the insertion is done at the beginning of the array. Despite the memory operations being fast, the large number of insertions during the assembling process is time consuming.

To study the efficiency of insertions during sparse matrix assembly we define the *Cost* of an insertion in a sparse matrix as the number of coefficients moved in `pr` to add a new value. Both the insertion at the end of `pr` and the modification of an existing coefficient have Cost zero. For example, a full $n \times n$ matrix filled sequentially following the storage sorting has zero assembly Cost. If the same matrix is filled in the reverse order, the Cost would be $n^2(n^2+1)/2$.

Two different alternatives are analyzed to minimize the Cost of the assembly process: (1) reduce the Cost of each insertion by node and element renumbering and (2) group insertions to make place for several new coefficients in a unique operation.

In Sections 3 and 4 we assume that the pattern of the matrix, that is all its non-zero positions, is completely unknown before the assembling process. The case of a known, or partially known, pattern is analyzed in Section 5.

3. INFLUENCE OF NODE NUMBERING

The Cost of the insertion of each coefficient during assembly process depends on the order in which degrees of freedom are added to the matrix. If degrees of freedom are added in reverse order, each insertion must move all previously inserted values, maximizing the Cost of the assembly and wasting computation time. Therefore, the Cost will be highly influenced by the numbering of the mesh nodes and elements.

Node renumbering procedures have been proposed in the literature to minimize the bandwidth of a sparse matrix (e.g. [3–9]). A lower bandwidth implies that the difference of numbering between related degrees of freedom is minimal. A deep analysis of different renumbering procedures is out of the scope of this study (we refer the reader, for example, to [10]). The efficiency of different ordering algorithms in finite element models was studied by Kaveh [7, 9, 11].

The classic reverse Cuthill–McKee (RCM) algorithm is used here as a reference tool to study the influence of the renumbering into the Cost of the assembly procedure.

The following assembly procedure in MATLAB is considered:

```
for e = 1:numberOfElements
  Te = T(e,:);
  Ke = calculateElementalMatrix( e );
  K(Te,Te) = K(Te,Te) + Ke;
end
```

where `Ke` is the elemental matrix, `K` the global matrix, `T` the connectivity array, and `Te` contains the global numbering of the nodes of element `e`. The Cost of this assembly procedure is evaluated for a set of meshes, with and without renumbering.

Figure 1 shows the Cost of each insertion during the assembly of the global matrix corresponding to an unstructured mesh of 104 linear triangular elements. The assembly Cost for renumbered and non-renumbered case is shown. The total Cost of the assembly is the sum of all those values (the area below the curve). During assembly of the non-renumbered case of the mesh, 32 606 coefficients were moved, while in the renumbered case only 4730 coefficients were moved. In this case, node and element renumbering allows reducing the Cost by a factor of seven.

Table I shows the assembly Cost for different meshes. The examples correspond to different element types (triangle/quadrangle, linear/quadratic). As expected, the node numbering has a high influence on the efficiency of the assembly process. In most cases, renumbering reduces the Cost of the assembly. Nevertheless, in two cases the RCM algorithm increases the bandwidth and the Cost (meshes 3 and 4 in Table I). This is due to the fact that these two meshes are structured and, consequently, the initial numbering of the nodes is somehow optimized. The heuristic RCM algorithm is designed for general meshes and does not account for the particular features of every concrete mesh. RCM is, therefore, expected to perform well for a general mesh but it is not ensuring a lower bandwidth for all meshes. In particular, this remark applies to meshes following a very special pattern as, for instance, structured meshes.

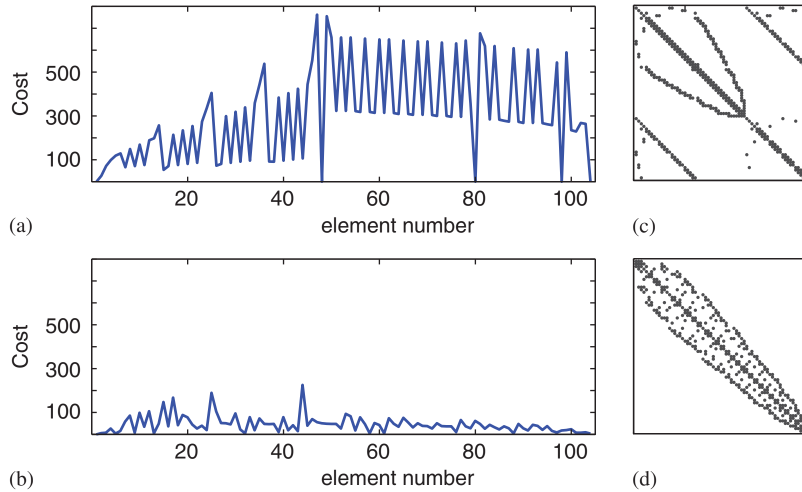

Figure 1. Cost of the assembly. The Cost of the insertion of each element during assembling process is shown for non-renumbered (a) and renumbered (b) cases. The total Cost is the area below the curve. The pattern of the matrices, associated with a mesh of 104 linear triangles, are shown in (c) and (d) and have bandwidths 65 and 14, respectively.

Table I. Comparison of Cost of assembly with and without renumbering for several meshes. Cost and Cost R are the total assembly Cost without and with renumbering, respectively.

Mesh	Structured	Elem shape	Num nodes	Num elem	Cost ($\times 10^4$)	Cost R ($\times 10^4$)	Cost/Cost R
1	No	△	3	104	3.2	0.4	8
2	No	△	3	4312	2104.7	109.9	19.15
3	Yes	□	4	400	14.2	19.1	0.7
4	Yes	□	9	400	670.0	945.7	0.7
5	Yes	△	3	800	8.0	5.4	1.5
6	Yes	△	4	800	871.7	55.3	15.7
7	Yes	△	6	200	41.7	33.4	1.24

4. INSERTING MATRIX COMPONENTS BY PACKAGES

A strategy of inserting matrix components by packages consists of inserting several coefficients in the global sparse matrix in a unique high-level operation. If the location of several coefficients in the `pr` array is known, it is possible to optimize the insertion algorithm reducing the number of memory movements. In particular, we compare the single-insertion of every component, and the possibility of inserting all the coefficients of every elementary matrix in a unique high-level operation.

4.1. Single-insertion versus package-insertion

The two extreme grouping cases are: *single-insertion* when the position of only one coefficient is known and *fully ordered-insertion* when the complete pattern of the matrix is known and all coefficients (the complete matrix) are inserted in one operation. The assembling Cost using the single-insertions is the sum of each individual insertion and the Cost of the fully ordered-insertion is zero. In the later case, the complete pattern of the matrix is known, thus the position of each coefficient in `pr` can be calculated in advance and no memory movements are necessary.

Both cases can be easily implemented in MATLAB. The widely used assignment instruction using matrix indexing

```
K(Te,Te) = K(Te,Te) + Ke                                    (1)
```

implements the single-insertion case.[‡] On the other hand, the instruction `sparse(i,j,s,m,n)` implements the fully ordered-insert case. This instruction uses `i`, `j`, and `s` to generate an m-by-n sparse matrix `S` such that `S(i(k),j(k)) = s(k)`. The main drawback using `sparse` is that all elementary matrices and all indices must be stored in memory before calling the `sparse` function. The memory needed to store all elemental matrices is, in average, six times larger than `nnz` for a triangular mesh, four times for a quadrilateral mesh, 24 times for a tetrahedral mesh, and eight times for an hexahedral mesh. Therefore, this procedure can only be used if memory restrictions are not critical.

4.2. The EbE-insertion algorithm

Between these two previous extreme cases is the insertion of all the components of every elementary matrix in a unique operation. Packing the information in a element-by-element basis is a natural option in finite element context, where one element is processed at time. Additionally, the package at element level, leads to minimal changes in existing codes to include the proposed procedure.

We call element-by-element insertion or *EbE–insertion* to the insertion of an element matrix in a package. The idea of the algorithm is first locate all the positions corresponding to the inserted elements in `pr`. Some positions may correspond to the already allocated positions in the global matrix. The modification of existing values is straightforward. The other positions need a modification of `pr` to create the places for the new coefficients. As several new places are created in a single operation, the movements of chunks of `pr` are reduced.

In other words, inserting every coefficient individually results in moving some components of the `pr` array several times, especially the components at the end of the array. Instead, we determine the space needed to insert all values in `pr` and, then, only one movement of each already allocated coefficient in `pr` is performed. Each coefficient is moved as many places as the number of new values is inserted before it, but just once.

Algorithm 1 lists the pseudocode of the EbE-insertion algorithm used to insert an element matrix K_e into the global matrix K at global positions T_e.

[‡]The algorithm used in MATLAB to implement matrix indexing is not available in MATLAB documentation. However, an accurate analysis of execution times indicates that the strategy is likely based on a single-insertion and, hence, not efficient. A brief discussion is given in Section 4.4.2.

Our implementation in C language of this algorithm can be downloaded from http://www.ija.csic.es/gt/sergioz/. It is ready to build a MATLAB external file (*mex file*) and it is designed to be used it in any MATLAB previous codes with minimal code changes: only one line of code has to be replaced (the assembly). This code has been successfully tested in Windows and Linux environments, and in 32 and 64 bits processors.

4.3. Counting operations

The complexity of the insertion algorithms (its order) is analyzed by counting the operations needed to insert a square element matrix of size m_{e} in a global square sparse matrix of size m. The worst case of the single-insertion algorithm uses

$$\mathcal{O}(m_{\mathrm{e}}^2\,(b+m+n_{\mathrm{nz}}))$$

operations. Where b and n_{nz} are, respectively, the max bandwidth and the number of non-zero coefficients of the global matrix. The worst case of the EbE-insertion algorithm as implemented

Algorithm 1. Pseudocode of EbE–algorithm

Input: global sparse matrix K (stored in the arrays pr, ir, and jc), element matrix K_{e}, insertion indices T_{e}
Output: updated matrix K
Sort: *sort the indices T_{e} and accordingly the element K_{e} matrix*
1: sort T_{e}
2: sort K_{e}
Find positions: *find the position in K where each coefficient of K_{e} will be stored. Each position may be already allocated or not. If not, memory movements are needed to store the new coefficient in its corresponding place. The position of the coefficients to be inserted is stored in the $newPositions$ array. The columns of the K matrix corresponding to the new positions are stored in the $columnNewPositions$ array.*
3: **for all** coefficient c in K_{e} **do**
4: p = find position of c in pr
5: **if** the position p is not already allocated **then**
6: add p to the $newPositions$ list
7: add the column to the $columnNewPositions$ list
8: **end if**
9: **end for**
Fix jc: *fix the jc array by increasing all values from the columns where the new coefficients are inserted to the end of the array.*
10: **for** i from 1 to size($modifiedColumns$) **do**
11: increase all coefficients of jc between $modifiedColumns[i]$ and $modifiedColumns[i+1]$
12: **end for**
Fix pr: *make place in the array pr (and ir) to store the new coefficients.*
13: add the size of pr to the end of $newPositions$ array
14: **for** i from size($newPositions$) to 1 step -1 **do**
15: move i places the segment of pr from $newPositions[i]-1$ to $newPositions[i-1]$
16: **end for**
Insert: *at this point all position in pr were created. Finally, the coefficients of K_{e} have to be inserted (or added) to their corresponding locations in pr. Only rest inserting (or adding) the coefficients of K_{e} in its corresponding position.*
17: **for all** coefficient c in K_{e} **do**
18: insert c
19: **end for**

in this work uses

$$\mathcal{O}(m_{\mathrm{e}}^2 b + m + n_{\mathrm{nz}})$$

operations. The average case for both algorithms modifies the term n_{nz} changing it by the average number of `pr` coefficients reallocated. This number is difficult to estimate in a general case.

4.4. Numerical test

4.4.1. Analysis of the insertion Cost. To test the EbE-insertion algorithm proposed in Section 4.2 we compare the assembling Cost using both single- and EbE-insertion procedures. The same meshes of Section 3 are used here. For each mesh, the node numbering with least possible bandwidth has been used (the RCM renumbered version of the triangular meshes and the non-renumbered version of the square structured meshes). Note that the worst is the node numbering, the larger is the difference between EbE- and single-insertion algorithms (favoring EbE-insertion). Thus, using a different numbering would result in an even better performance of the EbE-insertion algorithm.

As expected, the assembly using EbE-insertion has lower Cost for all the tested meshes. The improvement for high-order elements is much greater than in low-order elements. The reduction in Cost obtained using second-order squares and EbE-insertion is 95.9% (the Cost is divided by more than 20). For the linear triangles, the Cost reduction is between the 45 and 28%. Figure 2 shows Cost plots for both algorithms. In each panel the mesh and the pattern of the resulting matrix is also shown.

This analysis corresponds to a scalar problem with only one degree of freedom per node. For vectorial unknowns, with more than one degree of freedom per node, the gain using EbE-insertion is even larger.

4.4.2. Analysis of the assembly computing time. The ultimate goal of the proposed algorithm is to reduce assembling time. The Cost, as defined in this work, is a theoretical measure that is used as an indicator but it cannot be directly related to computing time.

A finite element global matrix is assembled and the time of insertion of each elementary matrix is registered for both the EbE- and single-insertion algorithms. The single-insertion is implemented with the matrix indexing MATLAB assignment and the EbE-insertion is implemented in a *mex* file.

Figure 3 shows the measured times for several meshes. The total assembly time is the sum of all elementary insertions or, graphically, the area below the curve. The EbE-insertion algorithm reduces the computation time in all tested cases. The same behavior as with the Cost indicator is obtained: higher-order elements have a larger improvement compared with simpler elements. Assembly time for mesh 3 (nine-noded quadratic squares) has been reduced by 90% compared with single-insertion time, even for a small size mesh with 400 elements.

The reduction in assembly time is more significant if the number of elements or the complexity of element type increases. This is confirmed by comparing the results corresponding to the meshes with the same element type. See, for instance, meshes 2, 6, and 1 (composed by mini element) with 4300, 800, and 104, respectively. The gain obtained using this strategy is, however, also important for simple elements. In fact, even for the simplest 2D elements, the linear triangles, the assembly time is reduced by 26%.

Since single-insertion is implemented in a MATLAB built-in function and EbE-insertion is a mex file, the time measured can be biased by different factors, such as interpreter time, dynamic link library calls, etc. To fairly compare single- and EbE-insertion algorithms, a single-insertion routine was implemented in an mex file and compared with the built-in MATLAB assigned function. The mex implementation is observed experimentally to be of about 10% faster than the MATLAB assignment, probably due to parameter checking. The reduction percentages displayed in Figure 3 refer to the standard MATLAB built-in function, which is the natural choice for a standard MATLAB user. If expressed with respect to the mex implementation of the single-insertion, the gain obtained by the EbE-insertion would be slightly lower.

The fact that the relation between the built-in MATLAB insertion and the mex single-insertion time is roughly a constant (it does not depend on the element order or the number of elements)

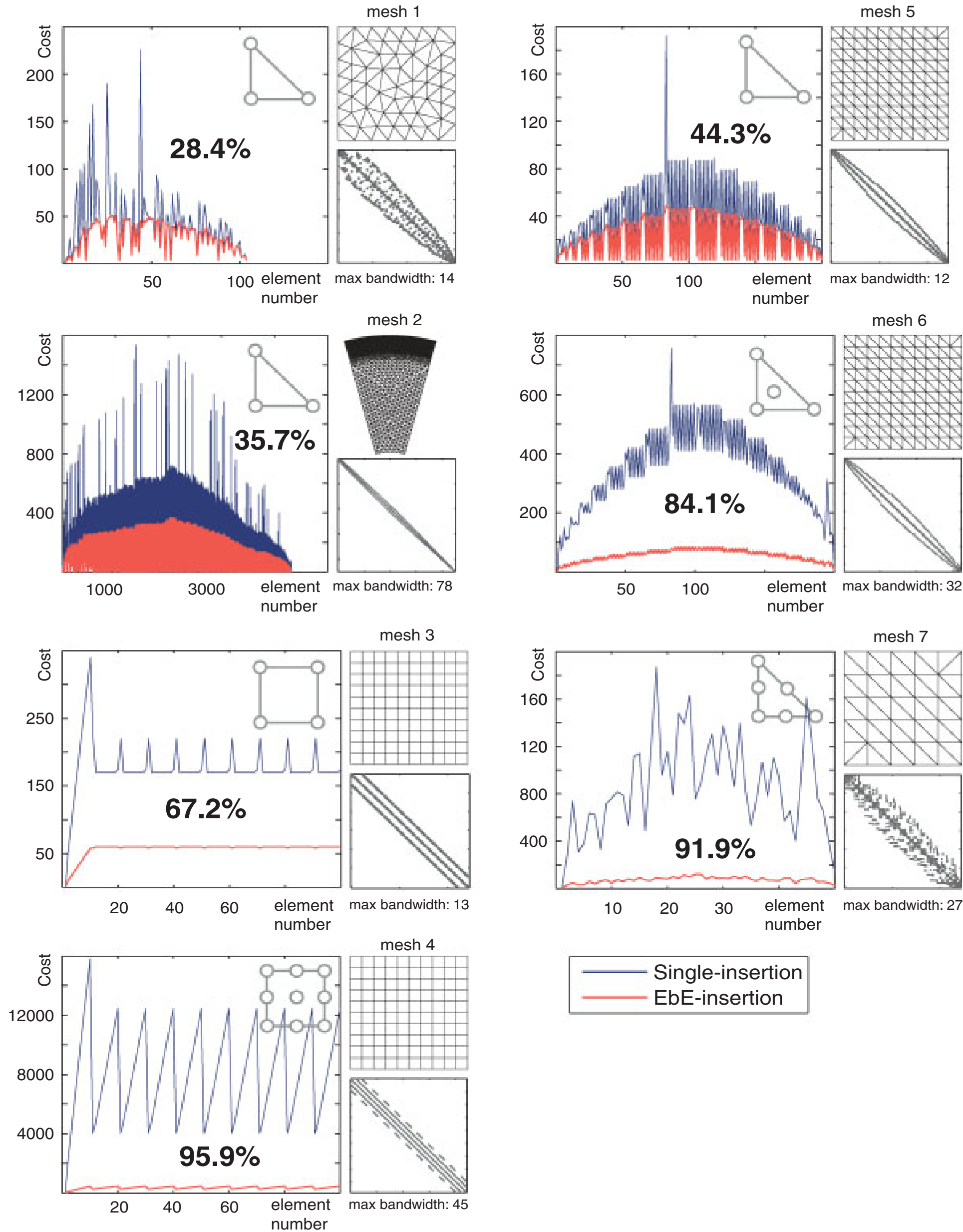

Figure 2. Comparison of the assembly Cost for single- and EbE-insertion strategies in different meshes. The mesh and the position of non-zero coefficients in the assembled matrix are shown at the right of each Cost plot. The element type is displayed inside each Cost plot. The reduction in Cost obtained using the EbE-insertion is displayed in boldface.

indicates that the number of operations done by the algorithms is of the same order. This suggests that the MATLAB assignment instruction uses this algorithm.

5. USING (PREVIOUS) MATRIX PATTERN IN A TIME MARCHING SCHEME

In a time marching procedure, where matrices with similar pattern have to be assembled at each time step, the efficiency of the assembly may be improved using the pattern of the matrix at the previous step. If the pattern of the matrix completely changes from step-to-step, as if a complete

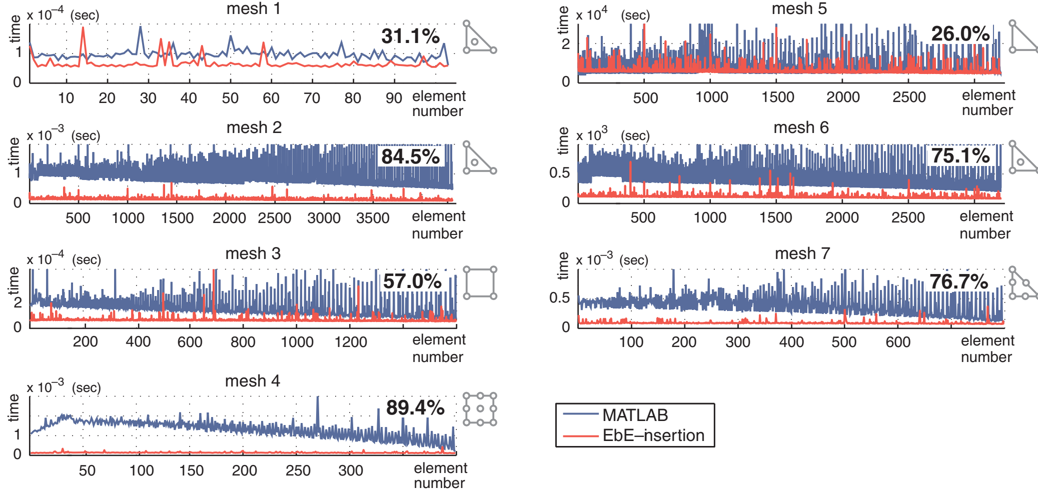

Figure 3. Comparison of the computing time used to assemble a global matrix with different insertion algorithms. The time used to insert each element in the global matrix is measured. The total assembly time is represented by the area below the plotted curve. Same meshes as in Section 4.4.1 are used. In meshes number one and two a central bubble node is included. Element type is displayed at the right of each panel. The reduction in the global matrix assembly time obtained using the EbE-insertion is displayed in boldface.

Algorithm 2. Scheme of the pattern-reutilization algorithm

Output: global K matrix

Initialization: *Set all coefficients in* pr *to be zero and left* `ir` *and* `jc` *unchanged. Note that after this step* K *is not a valid sparse matrix because there are zeros stored (in fact, all stored values are zero)*

1: **for all** coefficient c in pr **do**
2: $c \leftarrow 0$
3: **end for**

Assembly: *Assembly the matrix in a traditional way (using EbE- or single-insertion)*

4: **for all** element e **do**
5: compute the element matrix K_{e}
6: insert the K_{e} matrix in the global matrix K
7: **end for**

Fix sparsity: *If, after assembly, some coefficient of* K *remains zero, remove it*

8: **for all** coefficient c of K **do**
9: **if** $c = 0$ **then**
10: remove c from K
11: **end if**
12: **end for**

---

remesh is performed, no piece of information from previous step can be used and the situation is similar as in previous sections. If the pattern of the matrix remains unchanged, all information on the positions of the coefficients in `pr` is available, allowing for a zero Cost assembly in all steps after the first one. An intermediate case happens if most of the pattern is fixed and only a few positions change. Examples of this are the mesh updating methodology proposed by Knupp [12] and the procedures adding new degrees of freedom in a partition of the unity framework [1].

A very simple algorithm, taking advantage of the information of the previous pattern, can be implemented by maintaining the `ir` and `jc` vectors from the previous step and resetting the `pr` to zero. The scheme of this algorithm is summarized in Algorithm 2.

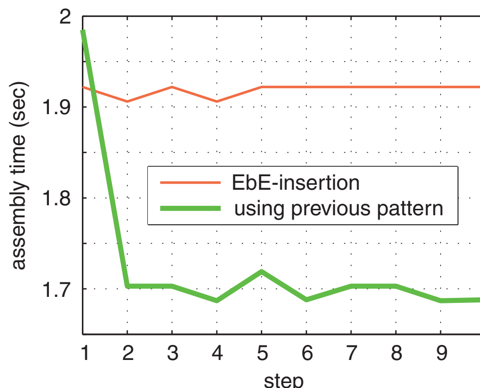

Figure 4. Time of assembly of a sparse global matrix in 10 consecutive time steps. Pure EbE-insertion algorithm and algorithm taking advantage of the pattern of the matrix in the previous step are shown.

In the first step, the sparse matrix K is cleared keeping the pattern in the `ir` and `jc` vectors. Note that after this step K is not a valid sparse matrix because it stores zero coefficients. The second step is a standard assembly process. All inserted coefficients will be added to existing positions in `pr`. As `pr` have been set to zero, the resulting `pr` vector is the same as starting from an empty matrix. If the pattern of the matrix has exactly the same pattern of the previous step, all positions in `pr` will be set to its new value. If a new coefficient needs to be stored in a previously zero position, the insertion algorithm will make the needed place. Finally, when the assembly has been done `pr` need to be checked to assure that no zero coefficients remain. If there are zeros stored, they have to be removed to preserve sparsity. That is done in the third step.

Note that the implementation of this strategy is not as simple as replacing the insertion instruction (1), which was the case of the EbE-insertion algorithm. In fact, this strategy involves intermediate states of non-valid sparse matrices. The first step (initialization) and the third step (fix sparsity) are not standard in MATLAB. This is because they produce (the first) and fix (the third) non-acceptable sparse matrices.

The MATLAB external routine provided here can also be used to perform these two operations (using the proper label as the input). The current implementation is only valid for matrices K with constant size (the sparsity pattern may change but the global dimensions m and n must be kept constant). The gain in computing time obtained by using this strategy in a time marching process has been found to be of about 10%. The outcome of the numerical experiment, with a mesh of 3200 triangular mini elements, is shown in Figure 4.

6. CONCLUSIONS

In this study, the assembly of sparse matrices in finite element-like framework has been analyzed. A new EbE-insertion algorithm is proposed that reduces significantly the assembly computing time. The role of node numbering and the different insertion strategies have been tested based on a theoretical Cost indicator and computing time.

Node and element numbering is an important factor, with a high influence on the efficiency of the assembly process of sparse matrices. Algorithms designed to reduce matrix bandwidth help to reduce the assembly time in most cases. Bad node numbering considerably increases the assembly computing time, eventually overcoming the time of solving the system.

A simple program based on grouping the insertions of an elemental matrix has been proposed and tested. This EbE-insertion strategy is natural in a finite element framework and requires minimal changes to include it in previous MATLAB codes. This program performs better than the MATLAB built-in assignment. In some cases, the assembly time has been reduced by 90%.

In evolutionary problems, stiffness matrices have to be assembled at each time step and the pattern of the previous step can be used to improve the assembly. Using this idea, the assembly time in a standard test is found to be reduced by 10%.

The implementation presented here is a C code that can be compiled to an MATLAB external file (*mex file*) and straightforwardly included in any existing MATLAB code.

REFERENCES

1. Zlotnik S, Díez P, Fernández M, Vergés J. Numerical modelling of tectonic plates subduction using X–FEM. *Computer Methods in Applied Mechanics and Engineering* 2007; **196**:4283–4293.
2. Manmaker L. *MATLAB External Interfaces*. The MathWorks, Natick, MA, release 2006b, for matlab 7.3 edition, September 2006.
3. Akhras G. A new node renumbering algorithm for bandwidth reduction. *International Journal for Numerical Methods in Engineering* 1987; **24**(9):1823–1824.
4. Boutora Y, Takorabet N, Ibtiouen R, Mezani S. A new method for minimizing the bandwidth and profile of square matrices for triangular finite elements mesh. *IEEE Transactions on Magnetics* 2007; **43**(4):1513–1516. (Boutora, Youcef Takorabet, Noureddine Ibtiouen, Rachid Mezani, Smail Sp. Iss. SI).
5. Cuthill E, McKee J. Reducing the bandwidth of sparse symmetric matrices. *Proceedings of the 1969 24th National Conference*. ACM Press: New York, 1969; 157–172.
6. Lai YC. A three-step renumbering procedure for high-order finite element analysis. *International Journal for Numerical Methods in Engineering* 1998; **41**(1):127–135.
7. Kaveh A. *Structural Mechanics*: *Graph and Matrix Methods* (3rd edn). Research Studies Press: Somerset, U.K., 2004.
8. Kaveh A. Multiple use of a shortest route tree for ordering. *Communications in Applied Numerical Methods* 2005; **2**(2):213–215.
9. Kaveh A. *Optimal Structural Analysis* (2nd edn). Wiley: Somerset, U.K., 2006.
10. Lohner R. Some useful renumbering strategies for unstructured grids. *International Journal for Numerical Methods in Engineering* 1993; **36**(19):3259–3270.
11. Kaveh A, Behfar SMR. Finite element nodal ordering algorithms. *Communications in Numerical Methods in Engineering* 1995; **11**(12):995–1003.
12. Knupp P. Updating meshes on deforming domains: an application of the target-matrix paradigm. *Communications in Numerical Methods in Engineering* 2007; DOI: 10.1002/cnm.1013.