Contents lists available at SciVerse ScienceDirect

Applied Mathematics and Computation

journal homepage: www.elsevier.com/locate/amc

Fast MATLAB assembly of FEM matrices in 2D and 3D: Nodal elements

Talal Rahman [a], Jan Valdman [b,*]

[a] Faculty of Engineering, Bergen University College, Nygårdsgaten 112, 5020 Bergen, Norway
[b] IT4Innovations Centre of Excellence of VŠB-TU Ostrava, tř. 17. listopadu 15, 708 33 Ostrava-Poruba, Czech Republic

A R T I C L E I N F O

Keywords:
MATLAB code vectorization
Finite elements
Stiffness
Mass matrices

A B S T R A C T

We propose an effective and flexible way to assemble finite element stiffness and mass matrices in MATLAB. The major loops in the code have been vectorized using the so called array operation in MATLAB, and no low level languages like the C or Fortran has been used for the purpose. The implementation is based on having the vectorization part separated, in other words hidden, from the original code thereby preserving its original structure, and its flexibility as a finite element code. The code is fast and scalable with respect to time.

1. Introduction

MATLAB has been for years one of the most reliable providers of the environment for computing with finite elements, whether in the classroom or in the industry. Several papers have been written in recent years focusing on the use of MATLAB for solving partial differential equations, cf. e.g. [1–6]. It is however known that MATLAB becomes extremely slow when it comes to executing codes with for-loops as compared to other languages like C or Fortran. Unless the for-loops are vectorized using the so called *array operation*, codes written in MATLAB cannot compete in speed with codes written in C or Fortran. In the process of vectorization, however, it is often the case that the code looses its original structure and becomes less flexible.

In this paper, we propose an effective implementation of the finite element assembly in MATLAB, where all loops over the finite elements are vectorized. In order to preserve its original code structure and its flexibility as a finite element code, attempts are made to keep the vectorization separate from the original code. The idea is to be able to develop an effective MATLAB code without having to think of the vectorization.

This concept was first used inside SERF2DMATLAB [7], a MATLAB finite element code for the simulation of electrorheological fluid. The present paper is an attempt to make this concept available for the first time to a broader community, and to lay a foundation for further development of the concept.

2. Example of non-vectorized and vectorized code: areas and volumes computation

For simplicity, let a 2D geometry be subdivided into triangles as shown in Fig. 1. Coordinates of the triangle nodes are collected in the matrix

```
coordinates = [0 0; 1 0; 2 0; 2 1; 1 1; 0 1; 0 2; 1 2],
```

and the connections between the nodes in the triangles are provided by the matrix

* Corresponding author.
E-mail addresses: talal.rahman@hib.no (T. Rahman), jan.valdman@vsb.cz, Jan.Valdman@gmail.com (J. Valdman).

0096-3003/$ - see front matter © 2011 Elsevier Inc. All rights reserved.
doi:10.1016/j.amc.2011.08.043

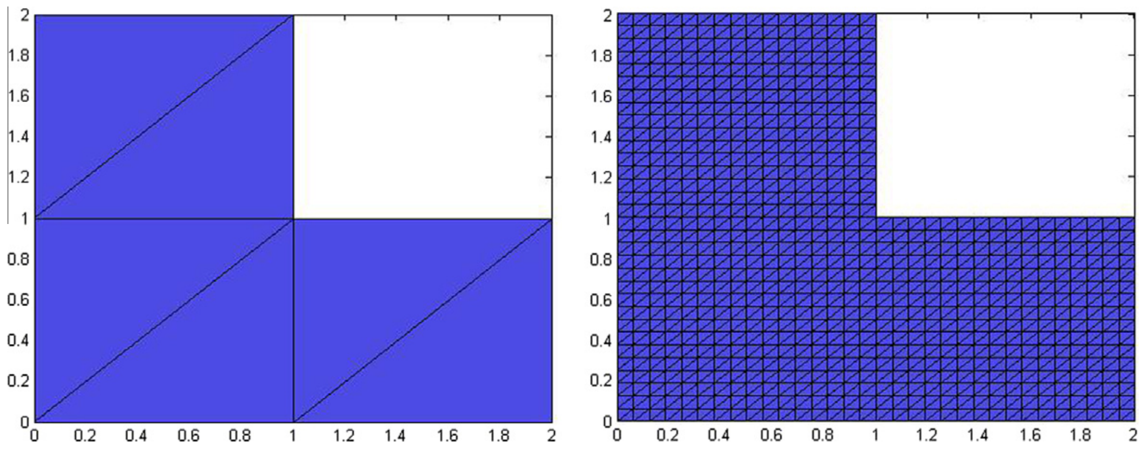

Fig. 1. Triangular coarse mesh (left) and level 4 refined mesh (right) of L-shape geometry.

```
elements = [1 2 5; 5 6 1; 2 3 4; 2 4 5; 6 5 8; 6 8 7]
```

If a triangle consists of three nodes with coordinates $(a,d),(b,e),(c,f)$, its area A_T can be computed as

$$A_T = \left|\det\begin{pmatrix}1 & a & d\\ 1 & b & e\\ 1 & c & f\end{pmatrix}\right| \Big/ 2 = \left|\det\begin{pmatrix}c-a & f-d\\ c-b & f-e\end{pmatrix}\right| \Big/ 2 = |a\cdot e + b\cdot f + c\cdot d - a\cdot f - b\cdot d - c\cdot e|/2.$$

Computation of all areas results in a loop over all triangles and in the following code.

Non-vectorized code to compute triangular areas:

```
tic
areas = zeros (size (elements,1),1);
for i = 1:size (elements,1)
a = coordinates (elements (i,1),1);
b = coordinates (elements (i,2),1);
c = coordinates (elements (i,3),1);
d = coordinates (elements (i,1),2);
e = coordinates (elements (i,2),2);
f = coordinates (elements (i,3),2);
areas (i) = abs (a * e + b * f + c * d - a * f - b * d - c * e)/2;
end
toc
```

Since the geometrical information needed for all triangles is known at once, the area computation can be done in the following loop-free code.

Vectorized code to compute triangular areas:

```
tic
a = coordinates (elements (:,1),1);
b = coordinates (elements (:,2),1);
c = coordinates (elements (:,3),1);
d = coordinates (elements (:,1),2);
e = coordinates (elements (:,2),2);
f = coordinates (elements (:,3),2);
areas = abs (a.*e + b.*f + c.*d - a.*f - b.*d - c.*e)/2;
toc
```

Both MATLAB codes were included in the file

```
start_compute_areas
```

which is a part of a MATLAB package described at the end of Section 4. Similar concept of vectorization can be applied to 3D. The volume V_T of a tetrahedron defined by vertices $(a,e,i),(b,f,j),(c,g,k),(d,h,l)$ is given as

$$V_T = \left|\det\begin{pmatrix}1 & a & e & i\\ 1 & b & f & j\\ 1 & c & g & k\\ 1 & d & h & l\end{pmatrix}\right| \Big/ 6 = \left|\det\begin{pmatrix} d-a & h-e & l-i\\ d-b & h-f & l-j\\ d-c & h-g & l-k\end{pmatrix}\right| \Big/ 6.$$

The last determinant involves 24 products of three factors and its computation would be therefore less efficient. By introducing new variables

$$\begin{aligned} &da := d-a, \quad he = h-e, \quad li = l-i,\\ &db := d-b, \quad hf = h-f, \quad lj = l-j,\\ &de := d-e, \quad hg = h-g, \quad lk = l = k \end{aligned}$$

and we rewrite the volume as

$$V_T = |da\cdot hf\cdot lk + db\cdot li\cdot hg + dc\cdot he\cdot lj - da\cdot lj\cdot hg - db\cdot he\cdot lk - dc\cdot li\cdot hf|/6,$$

i.e., only 6 products of three factors are required.

3. A concept of vectorization

MATLAB has two different types of arithmetic operations: *matrix operations*, defined by the rules of linear algebra, and *array operations*, carried out element by element.

Our implementation is based on extending the element-wise array operation into a matrix-wise array operation, calling it a *matrix-array operation*, where the array elements are matrices rather than scalars, and the operations are defined by the rules of linear algebra. These are handled by a set of MATLAB functions. Through this generalization of the array operation, it is now possible to keep the vectorization hidden from the original code.

The complete FE assembly is formulated in terms of matrix operations locally on each finite element. These are the standard FE assembly operations, see [8]. Once we have those formulations in hand, the task is then simply to implement them using those matrix-array operations, and finally to assemble the locally obtained results into one global result.

The class of finite elements, where it is quite easy to apply our concept, are the ones that are *iso-parametric*, e.g. the nodal elements. The basic feature of an iso-parametric element is that the same shape functions can be used to represent both the unknown variables and the geometry variables. Subsequently, if Φ_i are the shape functions defined on the *reference element*, then a mapping between the global and the reference coordinate systems can be given by the following relation called the iso-parametric property,

$$x = \sum_i \Phi_i(\xi,\eta)x_i, \quad y = \sum_i \Phi_i(\xi,\eta)y_i,$$

where (x,y) is a point on an element corresponding to the point (ξ,η) on the reference element. The pair (x_i,y_i) stands for the global coordinates of the node corresponding to the shape function Φ_i. Derivatives with respect to the global coordinates are easily calculated from the derivatives with respect to the reference coordinates using the Jacobian matrix.

In order to calculate the integrals for the element stiffness matrix, one needs to evaluate the shape function derivatives in the (ξ,η) coordinate system only at quadrature points on the reference element. These values are then converted into their counterparts in the (x,y) coordinate system through an application of the Jacobian matrix inversion. Using the iso-parametric property and an appropriate quadrature rule for the integration, it is in fact quite straight forward to represent the assembly of the local stiffness, and the local mass matrix, in terms of (local) matrix operations (e.g. matrix–vector multiplication, Jacobian matrix inversion), see [8] for details. Each of these (local) matrix operations are translated into one global matrix-array operation, as mentioned above, in order to be performed on all finite elements.

Let us explain our concepts on an assembly of a stiffness matrix for linear elements in 3D. Given a triangulation by matrices ‘elements’ and ‘coordinates’, we first create an array ‘coord’ of all coordinates corresponding to every element by

```
NE = size (elements,1);
coord = zeros (3,4,NE);
for d = 1:3
   for i = 1:4
      coord (d,i,:) = coordinates (elements (:,i),d);
   end
end
```

It should be noticed that the loops only run over indices 'i' and 'd' and not over the number of elements 'NE'. The above generated array of matrices 'coord' serves as an input argument of the function

```
[dphi,jac] = phider (coord,IP,'P1');
```

which provides derivatives of all shape functions defined on every tetrahedron and collected in an array of matrices 'dphi'. The size of 'dphi' is

$$3 \times 4 \times 1 \times \mathrm{NE},$$

which means that 3 partial derivatives of 4 linear basic functions defined on each tetrahedron are computed in one integration point. It is clear that the gradient of a linear basic function is a constant function and only one integration point is sufficient for an exact integration. The integration point 'IP' is declared globally

```
IP = [1/4 1/4 1/4]';
```

as the center of mass of the reference tetrahedron with vertices $(0,0,0)$, $(1,0,0)$, $(0,1,0)$, $(0,0,1)$. Obviously, an implementation of higher order elements requires more integration points, but it is feasible within this framework. Apart from all derivatives, the determinant of the Jacobian matrix of the affine mapping between the reference and the actual element is stored in an array of matrices 'jac' and the command

```
volumes = abs (squeeze (jac))/6;
```

provides a column vector of 'NE' elements containing volumes of all tetrahedral elements. After the third redundant dimension of the array 'dphi' is removed by

```
dphi = squeeze (dphi);
```

and a new array 'dphi' is of a size $3 \times 4 \times NE$, the global stiffness matrix 'K' is generated once by the command

```
K = sparse (X (:),Y (:),Z (:));
```

Here, arrays of matrices 'X','Y','Z' are created by

```
Y = reshape (repmat (elements,1,4)',4,4,NE);
X = permute (Y,[2 1 3]);
Z = astam (volumes',amtam (dphi,dphi));
```

Note that new functions working on arrays of matrices are needed:

– amtam – it inputs two arrays of matrices A, B of the same size and outputs the array of matrices C of the same size, such that

$$C(:,:,i) = A(:,:,i)' * B(:,:,i) \quad \text{for all } i.$$

– astam – it inputs a vector a of the size 'NE' and the matrix B and outputs the array of matrices C of the same size as B, such that

$$C(:,:,i) = a(i) * B(:,:,i) \quad \text{for all } i$$

Obviously, these functions are a part of a vectorization interface and are located in a directory called 'library_vectorization'. They have been implemented in a vectorized way and will be further optimized.

4. Example: linear elements for a scalar problem

We assume a 3D discretization of a unit cube depicted in Fig. 2 and consider only linear nodal shape functions Φ_i. This is the simplest choice, but the proposed technology works also for higher order shape functions. In discretization of second order elliptic problems, we typically need to construct a stiffness matrix K and a mass matrix M defined as

$$\mathrm{K}_{ij} = \int_\Omega \nabla\Phi_i \cdot \nabla\Phi_j \, dx,$$
$$M_{ij} = \int_\Omega \Phi_i \Phi_j \, dx,$$

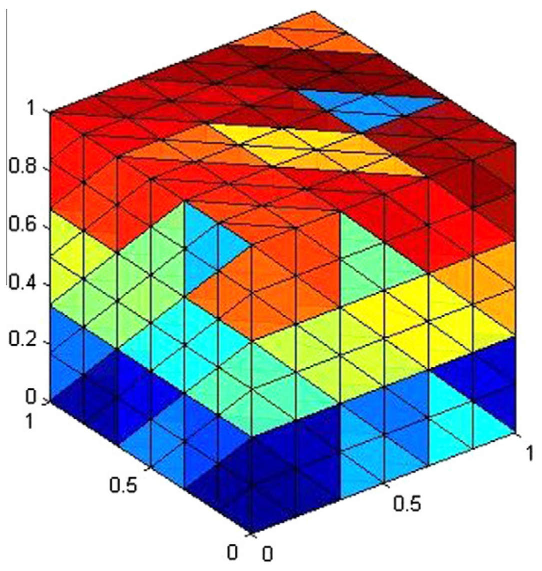

Fig. 2. Tetrahedral mesh of unit cube containing 1296 tetrahedra and 343 vertices.

where Ω is the domain of computation (the cube domain in this case) and ∇ denotes the gradient operator. There is one shape function per vertex, so the size of K and M is equal to the total number of vertices. We assemble both matrices for a sequence of nested meshes obtained by a uniform refinement in order to study the asymptotic behavior of our approach.

Assembly times (in seconds) are shown in Table 1 and were obtained on x4600-3.mis.mpg.de cluster (located at MPI MIS in Leipzig, Germany) with 256 Gb memory using one of 16 CPUs running on 2.8 GHz. We notice that the size of the matrices increases after every refinement approximately by factor of 8, and the time to assemble the matrices increases by approximately the same factor. This demonstrates an almost (linear) optimal time-scaling of our MATLAB implementation.

The MATLAB software is available for testing at MATLAB Central at http://www.mathworks.com/matlabcentral/fileexchange/authors/37756 as a package 'Fast assembly of stiffness and matrices in finite element method'. It contains two starting files:

```
start_assembly_P1_3D
```

Table 1
3D assembly of stiffness matrix K and mass matrix M using $P1$ tetrahedral elements.

Refinement level	Size of K and M	Assembly of K time (s)	Assembly of M time (s)
1	343	0,12	0,05
2	2.197	0,27	0,08
3	15.625	1,66	0,65
4	117.649	12,49	5,73
5	912.673	105,49	48,62
6	7.189.057	1.119,98	539,68

Table 2
2D assembly of stiffness matrix K and mass matrix M using $P1$ triangular elements.

Refinement level	Size of K and M	Assembly of K time (s)	Assembly of M time (s)
6	12.545	0,24	0,10
7	49.665	0,87	0,52
8	197.633	4,12	2,82
9	788.481	17,20	13,75
10	3.149.825	77,60	56,42
11	12.591.105	303,39	223,79
12	50.348.033	1.785,54	1.391,09

Table 3
3D assembly of **elastic** stiffness matrix *K* using *P*1 tetrahedral elements.

Refinement level	Size of *K*	Assembly time (s)	Storage of sparse matrix *K* (MB)	Total storage of X, Y, Z (MB)
1	1.029	0,27	0,54	4,27
2	6.591	2,56	3,84	34,17
3	46.875	23,71	29,22	273,38
4	352.947	201,75	227,48	2.187,00
5	2.738.019	1.838,54	1.795,66	17.496,00

and

```
start_assembly_P1_2D
```

The first file generates 3D results for Table 1 and the second file assembles the matrices K and *M* for a 2D L-shaped geometry using linear nodal elements. The performance is explained in Table 2. The matrix size increases after every refinement approximately by the factor of 4 and the time to assemble the matrices increases again by the same factor (with the exception of the last level, where the time factor is higher). Later, we plan to implement quadratic nodal elements in both 2D and 3D. Application of matrices K and *M* to numerical approximation of Friedrichs' constant in theory of Sobolev spaces is explained in paper [9].

5. Example: extension to linear elasticity

Ideas for the scalar problem explained in the last section can be easily extended to a vector problem such as the linear elasticity problem. Our implementation is directly based on ideas from [2] and allows for vectorized implementation of the software provided there. Here we only discuss the generation of the stiffness matrix K,

$$\mathrm{K}_{ij} = \int_\Omega \varepsilon(\eta_i) : \mathbb{C}\varepsilon(\eta_j)\, dx,$$

for linear elements on a tetrahedra mesh. Displacement of any tetrahedral node is described by 3 degrees of freedom, so the displacement basis functions are vector functions of the form

$$\begin{aligned}&\eta_1 = (\phi_1, 0, 0), \quad \eta_2 = (0, \phi_1, 0), \quad \eta_3 = (0, 0, \phi_1),\\ &\eta_4 = (\phi_2, 0, 0), \quad \eta_5 = (0, \phi_2, 0), \quad \eta_6 = (0, 0, \phi_2),\\ &\text{etc.}\end{aligned}$$

The ε represents a linearized strain tensor

$$\varepsilon(\boldsymbol{u}) = (\nabla \boldsymbol{u} + (\nabla u)^T)/2,$$

where $\boldsymbol{u}$ is a piecewise linear vector function expressed as the linear combination of the functions $\eta_1, \eta_2, \ldots$. An elasticity operator $\mathbb{C}$ corresponds to the linear Hook's law $\sigma = \mathbb{C}\varepsilon(\boldsymbol{u})$, where σ is the stress tensor and $\mathbb{C}$ can be expressed in terms of the Lamé parameters λ and μ.

The vectorization of the code

```
fem_lame3d
```

from [2] is straightforward. The strain tensors will be now stored in an array of matrices

```
R = zeros (6,12,NE);
R ([1,4,5],1:3:10,:) = dphi;
R ([4,2,6],2:3:11,:) = dphi;
R ([5,6,3],3:3:12,:) = dphi;
```

where 'dphi' is generated for the scalar problem in Section 4. With the elasticity matrix at hand

```
C = mu *diag ([2 2 2 1 1 1]) + lambda *kron ([1 0; 0 0],ones (3));
```

the global stiffness matrix will be generated again by

```
K = sparse (X (:),Y (:),Z (:));
```

Table 4
2D assembly of **elastic** stiffness matrix K using $P1$ triangular elements.

Refinement level	Size of K	Assembly time (s)	Storage of sparse matrix K (MB)	Total storage of X, Y, Z (MB)
6	25.090	1,44	4,74	20,25
7	99.330	4,99	18,85	81,00
8	395.266	23,54	75,20	324,00
9	1.576.962	90,18	300,41	1.296,00
10	6.299.650	381,54	1.200,81	5.184,00
11	25.182.210	1.598,47	4.801,62	20.736,00

where arrays of matrices 'X','Y','Z' are created by

```
Y = reshape (repmat (Elements,1,12)',12,12,NE);
X = permute (Y,[2 1 3]);
Z = astam (volumes',amtam (R,smamt (C,permute (R,[2 1 3]))));
```

Note that a new matrix

```
Elements = 3*elements (:,kron (1:4,[1 1 1]))...
-kron (ones (NE,1),kron ([1 1 1 1],[2 1 0]));
```

provides a local–global nodes numbering for the vector problem. In addition to the function 'amtam' and 'astam' used in the scalar problem, a new function from the directory 'library_vectorization' is needed:

– smamt – it inputs a matrix A and an array of matrices B, and outputs the array of matrices C, such that

$$C(:,:,i) = A * B(:,:,i)' \quad \text{for all } i.$$

The performance of our vectorized approach can be tested by running the file

```
start_assembly_P1_3D_elasticity
```

For the convenience, a 2D implementation has also been included in the file

```
start_assembly_P1_2D_elasticity
```

Tables 3 and 4 demonstrate again a proper (linear) scalability with respect to the time. We also provide memory requirements for the total storage of all three arrays of matrices X, Y, Z and for the stiffness matrix K after the assembly using the 'sparse' command. Note that X, Y are integer arrays and Z is a double array with the same dimension equal number of elements 'NE' times the size of the local stiffness matrix which is 6×6 (12×12) in 2D (3D) for linear elasticity using linear nodal elements on triangles (tetrahedra). In our benchmarks, the memory needed to store X, Y, Z is about 5 times larger in 2D and about 10 times larger in 3D. This is a typical redundancy for FEM assembling procedures drive and it cannot be avoided in other C or Fortran based implementations that compute all local (stiffness of mass) matrices at once.

5.1. Possible extensions in future

We are primarily interested in extending the functionality of our code to nodal rectangular elements in 2D and hexahedral elements in 3D. Another focus will be nonnodal elements such as Raviart–Thomas Elements from Hdiv spaces for mixed formulations and Nedelec elements from Hcurl space for computations of Maxwell equations.

Acknowledgment

The second author would like to express his thanks to Max Planck Institute for Mathematics in the Sciences (MPI MIS) for providing computer resources during his stay in Leipzig in 2011.

References

[1] J. Alberty, C. Carstensen, S.A. Funken, Remarks around 50 lines of MATLAB: short finite element implementation, Numer. Algorithms 20 (1999) 117–137.
[2] J. Alberty, C. Carstensen, S.A. Funken, R. Klose, MATLAB implementation of the finite element method in elasticity, Computing 69 (2002) 236–263.
[3] S. Funken, D. Praetorius, P. Wissgott, Efficient implementation of adaptive P1-FEM in MATLAB, ASC Report 19/2008, Institute for Analysis and Scientific Computing, Vienna University of Technology, Wien, 2008.
[4] M.S. Gockenbach, Understanding and Implementing the Finite Element Method, SIAM, 2006.

[5] J. Koko, *Vectorized MATLAB codes for linear two-dimensional elasticity*, Sci. Program. 15 (3) (2007) 157–172.
[6] P.-O. Persson, G. Strang, *A simple mesh generation in MATLAB*, SIAM Rev. 42 (2004) 329–345.
[7] T. Rahman, SERF2D-MATLAB (Ver. 1.1) – Documentation, University of Augsburg, 2003.
[8] I.M. Smith, D.V. Griffiths, Programming the Finite Element Method, fourth ed., John Wiley & Sons, 2004.
[9] J. Valdman, Minimization of Functional Majorant in A Posteriori Error Analysis based on H(div) Multigrid-Preconditioned CG Method, Advances in Numerical Analysis, vol. 2009, Article ID 164519, 2009.