A new and simple technique for vectorization of Finite Element Method in electromagnetics

C. H. Veloso [1], A. M. F. Frasson [2] and K. Z. Nobrega [*]

[1,*] DEE-Instituto Federal do Maranhão/ [2] DEE-Universidade Federal do Espírito Santo

[1,*] Av. Getúlio Vargas 4, Monte Castelo 65030-005 (São Luis-Ma, Brazil), [2] Av. Fernando Ferrari 514, Goiabeiras 29075-910 (Vitória-Es, Brazil)

[1] henri_5_veloso@hotmail.com; [2] antonio@denise.ele.ufes.br; [*] bzuza@ifma.edu.br

***Abstract*— Here it will be present a new approach on programming the finite element method that turns programs faster, simpler and more effective in languages like MATLAB or SCILAB, for example, when compared to the traditional way to program and that uses built-in function. Through this new approach, Laplace's Equation in a 2D domain has been solved getting a better performance up to 2.5 for a linear mesh with triangles and more than 1,000,000 nodes. Finally, some remarks will also be done about other possible uses of the technique.**

***Index Terms*—Finite element methods; Laplace equation; MATLAB.**

I. INTRODUCTION

During the last twenty years, MATLAB has been widely used as a potential tool for scientific programming, especially for solving of differential equation either in the industry or in the classroom [1]-[6]. Once it is an interpreted language, MATLAB is usually slower than compiled languages like C or FORTRAN although it won't be true for all kind of application. On reality, Matlab can be well adapted to numerical computation since the underlying algorithms for built-in functions and supplied m-files are based on the standard libraries LINPACK and EISPACK, usually compiled in C.

Here, it will be reported an effective, simple, vectorized and easily adaptive way to program the finite element method (FEM) in MATLAB using built-in functions only (except for one) with no loops. In fact, spent times of simulation comparing a code with loop and the one proposed will be shown. The case of study was Laplace's equation in a 2D domain with nodal formulation. Although not described here due to page limitations, a simple approach to generate the element matrices applied into the proposed code is also discussed and it will be explained on the next section.

II. NODAL FORMULATION FOR 2D DOMAIN

As it is well-known, the FEM formulation with triangular elements can be described in four simple steps:

1. in the local coordinates (ξ, η), the shape functions $\{N_i\}$ are written as

$$\begin{Bmatrix} N_1 \\ N_2 \\ N_3 \end{Bmatrix} = \begin{Bmatrix} L_1 \\ L_2 \\ L_3 \end{Bmatrix} \text{ or } \begin{Bmatrix} N_1 \\ N_2 \\ N_3 \\ N_4 \\ N_5 \\ N_6 \end{Bmatrix} = \begin{Bmatrix} L_1.(2L_1 - 0.5) \\ L_2.(2L_2 - 0.5) \\ L_3.(2L_3 - 0.5) \\ 4L_1L_2 \\ 4L_2L_3 \\ 4L_3L_1 \end{Bmatrix} \tag{1}$$

with $L_1 = 1 - \xi - \eta \quad , L_2 = \xi, \quad L_3 = \eta$.

2. The coordinates *x*, *y* become:

$$\begin{aligned} x &= x_1.L_1 + x_2.L_2 + x_3.L_3 \\ y &= y_1.L_1 + y_2.L_2 + y_3.L_3 \end{aligned} \tag{2}$$

where x_i and y_i are coordinates into node *i* for a given element.

3. An integration in global coordinates can be transformed in local coordinates as:

$$\iint_{\Omega e} f(x,y)dxdy = \iint_{\Lambda} f(\xi,\eta)\left|J(\xi,\eta)\right|d\xi d\eta \tag{3}$$

where $\left|J(\xi,\eta)\right|$ is the determinant of Jacobian matrix [*J*], given by

$$[J] = \begin{bmatrix} \partial x/\partial\xi & \partial y/\partial\xi \\ \partial x/\partial\eta & \partial y/\partial\eta \end{bmatrix}^e . \tag{4}$$

4. Finally, the last step concerns differential operations, it means:

$$\begin{bmatrix} \partial\{N\}/\partial x \\ \partial\{N\{/\partial y \end{bmatrix} = [J]^{-1}\begin{bmatrix} \partial\{N\}/\partial\xi \\ \partial\{N\}/\partial\eta \end{bmatrix}. \tag{5}$$

The idea to program these four steps is to easily and automatically produce the element matrices to be introduced into the main program discussed in next section. In fact, the relations expressed by Eqs (1) to (5) already contemplate the linear or quadratic elements and they can be easily modified to 3D domain.

III. CONCEPT OF THE VECTORIZED CODE

At first, it is assumed that mesh data have already been loaded. For the Laplace's problem those data are saved on three variables: *nodes*, *topol* and *bound*. *nodes* is a matrix containing coordinates of all points of discretization and its dimension will be n_n x *2.* The variable *topol* contains mesh's topology and it is a matrix n_e x *ie* (*ie*=3,6 for linear or quadratic elements, respectively). Finally, *bound* has dimension n_b x *2* where the first and the second columns corresponds to number of the node and the value associated to it, respectively. It will be used to assign tension's value in the already known nodes. Following, one has the possible code for implementation, explained line by line.

As someone knows, MATLAB has to find and to assign a contiguous piece of free RAM memory for a given variable. For this reason, the following coordinates x_i and y_i have not used into the same variable but for different ones. Actually, they demand the same RAM memory size but in the process of allocation it is faster writing in this way because different variables are smaller if compared to a single one. Thus, until the line of *Ae* one will have the area of all elements that is based on x_i and y_i only.

```
ie=size(topol,2);
x1=nodes(topol(:,1),1);
x2=nodes(topol(:,2),1);
x3=nodes(topol(:,3),1);
y1=nodes(topol(:,1),2);
y2=nodes(topol(:,2),2);
y3=nodes(topol(:,3),2);
Ae=(x2-x1).*(y3-y1)-(x3-x1).*(y2-y1);
```

Next, the line code has intention to create two vectors *col* and *lin* associated to a matrix *ie* x *ie*. The idea is to create two vectors containing the position of a matrix *ie* x *ie* and use the indices to pipe all local nodes from *topol*. For example, the following line code produces *lin =[1 1 1 2 2 2 3 3 3]* and *col =[1 2 3 1 2 3 1 2 3]* where *lin* makes the rule of changing the weight function in the FEM while *col* the tension (unknowns). It must be emphasized that `ind2sub` or any equivalence is the only non built-in function along this whole code, it means, the only command line slower if compared to C or FORTRAN but its idea is too simple and truly fast even for 3D domain.

```
[col lin]=ind2sub([ie,ie],1:ie^2);
```

After that, all those combinations (for all elements at the same time) will be stored into variable *biunivoc* which keeps the position where the element matrices (here assembled for all elements) must be pre-allocated into the global matrix.

```
BIUNIVOC= zeros(nel*ie^2,2);
BIUNIVOC=[reshape(topol(:,lin),nel*ie^2,1) reshape(to
pol(:,col),nel*ie^2,1)];
```

Then, taking the expression for the element matrix generated by any routine following steps of section 2, one assign *biunivoc* as the pseudo global matrix to finally assemble the global matriz, *A*, through the `sparse` function.

```
A_aux=[
    - (x2 - x3).^2./(4.*Ae) - (y2 - y3).^2./(4.*Ae);
    ((x1 - x3).*(x2 - x3))./(4.*Ae) + ((y1 - y3).*(y2 -
y3))./(4.*Ae);
    - ((x1 - x2).*(x2 - x3))./(4.*Ae) - ((y1 - y2).*(y2 -
y3))./(4.*Ae);
    ((x1 - x3).*(x2 - x3))./(4.*Ae) + ((y1 - y3).*(y2 -
y3))./(4.*Ae);
    - (x1 - x3).^2./(4.*Ae) - (y1 - y3).^2./(4.*Ae);
    ((x1 - x2).*(x1 - x3))./(4.*Ae) + ((y1 - y2).*(y1 -
y3))./(4.*Ae);
    - ((x1 - x2).*(x2 - x3))./(4.*Ae) - ((y1 - y2).*(y2 -
y3))./(4.*Ae);
    ((x1 - x2).*(x1 - x3))./(4.*Ae) + ((y1 - y2).*(y1 -
y3))./(4.*Ae);
    - (x1 - x2).^2./(4.*Ae) - (y1 - y2).^2./(4.*Ae);
    ];
A = sparse(BIUNIVOC(:,1),BIUNIVOC(:,2),A_aux);
```

Finally, the last command lines are self-explained.

```
V=zeros(nn,1); % Create with 0 to pre-allocate memory
V_known=V;     % Vector of the known values
V_known(bound(:,1),1)=bound(:,2); % apply the values
b= -A*V_known;                    % create the vector b

A(bound(:,1),:)=[]; % remove the already known position from
A(:,bound(:,1))=[]; % matrix A and
b(bound(:,1))=[];   % b

Venx= A\b; %Solve the system and to find the unknowns
```

IV. Results

To illustrate potential use of this approach, it was developed a traditional FEM program under the same circumstances as well as the vectorized code to compare then. Also, it must be pointed that subroutines were running using an Intel i7 with only 8GB of RAM's memory with no parallel code. Table 1 illustrates the results.

TABLE I
Time to solve Laplace's Equation for linear elements.

Nº of nodes	t (s) without loop	t (s) with loop	t_{loop}/t_{noloop}
221	0.020	0.033	1.663
455	0.031	0.059	1.936
495	0.035	0.090	2.525
704	0.045	0.250	5.566
1,193	0.084	0.122	1.460
2,398	0.087	0.200	2.307
3,215	0.102	0.272	2.654
3,994	0.151	0.324	2.141
9,586	0.265	0.730	2.753
9,677	0.263	0.738	2.806
16,655	0.480	1.263	2.633
22,005	0.566	1.665	2.943
49,292	1.360	3.817	2.807
99,508	2.917	7.695	2.638
201,601	5.985	15.885	2.654
419,093	13.750	33.337	2.425
876,682	28.531	71.364	2.501
971,448	32.885	83.829	2.549
1,081,843	36.860	99.554	2.701
1,210,588	41.175	107.161	2.603
1,554,142	55.865	143.670	2.572
1,788,404	65.283	163.417	2.503

According to Table 1, one can see how the vectorized code is more efficient than the conventional one for a factor about 2.5 even for a number of nodes above 1M. Also, one should remember that only built-in function have been used, which turns the code with a processing similar to language C. Although not shown, similar results are also obtained with quadratic elements.

Above all, notice that for quadratic elements the only line code that changes is the element matrix *A_aux*, turning this approach very simple. Besides, for the best of our knowledge all vectorized codes still have a loop, which is completely absent in this one. As a last remark, this idea can be easily extended for other kinds of problems like mode analysis or beam propagation no matter dimension of the mesh and even considering conditions like PML, for example.

Finally, authors would like to thank you both FAPEMA and IFMA for the financial support of this research project.

References

[1] J. Alberty, C. Carstensen, S. A. Funken, R. Klose, “MATLAB implementation of the finite element method in elasticity,” *Computing*, vol.69, pp. 236-263, 2002.

[2] M. S. Gockenbach, *Understanding and implementing the Finite Element Method*, SIAM, 2006.

[3] J. Koko, “Vectorized MATLAB codes for linear two-dimensional elasticity,” *Sci. Program*, vol.15, no. 3, pp. 157-172.

[4] I. M. Smith, D. V. Griffiths, *Programming the Finite Element Method*, 4th ed., John Wiley & Sons, 2004.

[5] J. Alberty, C. Carstensen, S. A. Funken, R. Klose, “MATLAB implementation of the finite element method in elasticity,” *Computing*, vol.69, pp. 236-263, 2002.

[6] Talal Rahman, Jan Valdman, “Fast MATLAB assembly of FEM matrices in 2D and 3D: Nodal elements,” *Applied Mathematics and Computation*, to be published.