Z. Angew. Math. Phys. (2024) 75:150

0044-2275/24/040001-37
published online July 28, 2024
https://doi.org/10.1007/s00033-024-02293-w

Zeitschrift für angewandte Mathematik und Physik ZAMP

Vectfem: a generalized MATLAB-based vectorized algorithm for the computation of global matrix/force for finite elements of any type and approximation order in linear elasticity

Baurice Sylvain Sadjiep Tchuigwa, Jan Krmela, Jan Pokorny, Vladimíra Krmelová and Petr Jilek

Abstract. In this paper, we introduce a new vectorized MATLAB-based algorithm for efficient serial computation of global matrix/force arising from finite element method (FEM) for meshes of any type and approximation order in linear elasticity. Because for-loops in MATLAB are very slow, we propose a modified process that takes advantage of vectorization and sparse assembly to achieve good performance while using the same memory as the standard algorithm. For this purpose, by using good programming practices, the implementation of this scheme is succinctly described and can be integrated into any MATLAB package dealing with FEM. Specifically, attention is paid to the calculation of the triplet (row index, column index, matrix components) as well as the assembly of the global stiffness matrix, mass matrix and force vector. Additionally, an extension of the proposed approach for Mindlin plate theory and functionally graded materials is outlined. Finally, the accuracy of this strategy is verified on selected numerical tests after comparing the obtained results with those of ABAQUS. In terms of performance, the study conducted on a set of meshes considering the standard algorithm and two other well-known MATLAB vectorized algorithms revealed that: (i) for a 2D beam problem meshed with P_1-triangle elements, a speedup of about 8 and 15 is achieved with `sparse` and `fsparse`, respectively. (ii) for a 3D plate problem meshed with P_1-tetrahedral elements, a speedup of about 4 and 8 is achieved with `sparse` and `fsparse`, respectively. When compared to ABAQUS performance, the proposed scheme results in a computational time that is about five times smaller.

Mathematics Subject Classification. 74S05, 74B05, 65-04.

Keywords. Finite element method, Algorithm, Vectorization, Elasticity, Sparse, MATLAB.

1. Introduction

Over the decades since the finite element method (FEM) was introduced initially by A. Hrennikoff in the work [1], in which he discretized the domain as lattice structure, then treated using a variational method by R. Courant [2], significant advancements have been achieved so far thanks to several discoveries in mathematics, particularly in variational methods [4,6], numerical analysis [5] and the development of computers [3]. Thereby, the simulation of more complex problems, which was yesterday a dream, is progressively coming true today. However, despite these major strides observed in FEM [4], new challenges have emerged, such as sustainability [7] and integrative design [8], which imply multi-physics simulations [9,10,18], primordial for the optimal and rational use of natural resources. The concepts of sustainability and integrative design impose the development of new models capable of bringing together all the physical phenomena and driving forces behind the behavior of materials or systems, leading to more realistic results. Of course, this comes with a wide range of bottlenecks, like the complexity of the model [14,16]. Failing to lighten models without depriving them of all their consistency and accuracy, one can improve their implementation by easing their interpretability and execution on computing resources.

This work was funded by the University of Pardubice (grant No. SGS_2023 and SGS_2024) and the Cultural and Educational Grant Agency of the Slovak Republic (project No. KEGA 003TnUAD-4/2022)..

At the early stage of the development of simulation tools in university research centers, MATLAB [11] has become one of the most widely used programming languages, not only for teaching purposes in STEM fields, but also for the deep-end research industry since MATLAB is a high-level vector-based programming language that enables to operate on vectors or matrices instead of basic elements. More concretely, this feature makes it possible to get rid of for-loops by transforming the intervening expressions so as to execute operations in fewer instructions. Thus, using this vector feature in the context of FEM implementation [17,19] is a tremendous asset for developing more advanced packages designed for engineers dealing with simulations or researchers/lecturers to introduce students to the programmatic aspects of FEM in a practical way.

In this regard, one of the first attempts to vectorize FEM codes in MATLAB is due to work by Koko [12], in which the author presented a strategy on how to vectorize the computation global matrix/forces for the discretized weak form deriving from the strong equation in linear elasticity. However, this strategy is restricted to finite elements in 2D. Aside from that, Dabrowski et al. [13] introduced an optimized implementation of FEM in MATLAB called MILAMIN for large problems. Although vectorization strategy is not particularly explored in this work, the authors developed a new scheme to ease the construction of the triplet (column index, row index, matrix or vector components) and assembly of global matrix/force. Unlike what is commonly believed, authors demonstrated that in linear elasticity, the scheme outperforms many C++ or Fortran-based packages like OOFEM or FEAPpv.

A few years later, Cecka et al. [20] came out with an efficient strategy for the implementation on GPUs with an optimal use of memory. The authors in [22] proposed a vectorized scheme limited to problems meshed with P_1 triangle elements (in 2D) and P_1-tetrahedral elements (in 3D) in linear elasticity. Although this strategy is free from the for-loop over elements, its scheme is computationally expensive since many smaller for-loops are introduced in the functions. Based on this work, [23] extended it to elasticity problems meshed with linear quadrilateral elements. Later on, [15] introduced the first MATLAB-based vectorized algorithm for triangular and tetrahedral elements of any order in linear elasticity. Besides its performance over the standard assembly method, this approach requires less than a quarter of the memory (RAM) needed by the former.

Marcinkowski et al. in [21] investigated the issue by taking advantage of multidimensional arrays to compute and store element matrices (stiffness and mass) at once and then use an iterative process to solve the global equation without assembling. However, when it comes to dealing with topologies with a high number of elements, the iterative feature of this approach turns out to be a big drawback and limiting factor.

In the work [24], the author paid particular attention to good programming styles in MATLAB to vectorize matrix/vector computation for the Poisson equation in a domain discretized with P_1 triangle elements with one nodal degree freedom. Notwithstanding the fact that invoking sparse representation [25–27] with vectorization technique represents a fantastic asset, it is essential to have a good balance between the gained speed and memory needed for the computation since an optimal performance demands a good memory allocation extensively discussed in [28,29].

In the recent version of MATLAB, starting from R2020b, new built-in functions such as `pagetimes, mtimes, pagemldivide, and pagetranspose` make it possible to efficiently multiply or transpose multidimensional matrices.

After underlining all these points, in this paper we present a generalized MATLAB-based vectorized algorithm for the computation of global matrix/force for finite element of any type and any order in linear elasticity. In this scheme, the following new features are addressed:

- With help of MATLAB built-in functions operating on multidimensional matrices, the *for-loop over elements is removed*;
- At each integration point, *all element matrix/vector* of the mesh are computed *at once* and then summed up to get the contribution of all integration points;

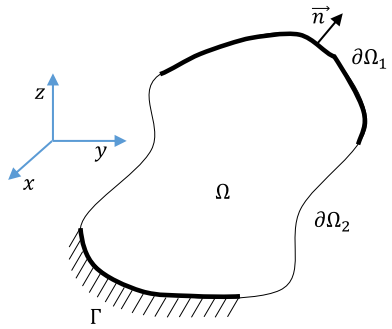


FIG. 1. Schematic drawing of the continuum body

- For any type of elements and nodal degree of freedom, the computation of *column indices and row indices* is obtained from degrees of freedom connectivity matrix using *only basic operations and a single for-loop*;
- *Generalization to elements of any types* in 2D (triangle and quadrilateral) and 3D (tetrahedral, hexahedral, pentahedral, etc.) and for *any approximation order* (linear, quadratic, cubic, etc.);
- Extension to Mindlin plate theory problem and functionally graded materials.

The structure of this paper is as follows: First, a brief classical and discretized variational formulation for general boundary value problems in elastodynamics is provided in Sect. 2, along with the standard algorithm for computing the global stiffness/mass matrix and force vector. After examining the shortcomings of the standard scheme, the methodology used to develop and implement in MATLAB the present algorithm is then succinctly described in the next section (Sect. 3). In the final section (Sect. 4), an L-shaped elastic structure is simulated using the proposed algorithm and the results validated after comparison with ABAQUS results of the same structure. Subsequently, a performance analysis is carried on a set of meshes for a 2D beam problem and a 3D plate problem to validate the computational cost and memory utilization of the proposed scheme over the standard algorithm and the vectorized algorithms by Anjam et al. [29] and Cuvelier et al. [15].

2. Problem formulation

For the sake of clarity, three types of notations are purposely used in the remainder to differentiate MATLAB's commands and syntax from variables and expression defined in this study. Words in monospace typewriter font (`pagetimes`, `zeros`, `transpose`, `none`, etc.) refer to MATLAB's built-in functions or syntax. Bold camel-case words are name of defined functions (**GaussQuadrature**, **ShapeFunctions**,etc.), while italic characters and words are variables or operators (n, u,etc.).

Let us consider a continuum body (see Fig. 1) that initially (at $t = 0$) occupies the domain $\Omega_0 \subset \mathbb{R}^d{}_{|d\in(1,2,3)}$ in the reference configuration and $\Omega_t \subset \mathbb{R}^d{}_{|d\in(1,2,3)}$ in the current configuration and is subjected to body forces f_v per unit mass and a stress vector t applied on the surface $\partial\Omega_1$.

At any time in the Cartesian coordinates, any point X of the body is described according to the repeated indices rule by $u = u_i E_i$ where $i \in < 1, 2, 3 >$ and E_i are unit orthogonal vectors of the basis. Let also the boundary of the bounded set Ω (see Fig. 1) be defined by the union of three nonoverlapping partitions:

$$\partial\Omega = \partial\Omega_1 \cup \partial\Omega_2 \cup \Gamma \tag{2.1}$$

Wherein $\partial\Omega_1$ is the boundary part where traction vector $t = \sigma\mathbf{n}$ is acting, Γ is the set of points where boundary conditions are enforced ($u = u_p$) and $\partial\Omega_2$ the remaining subset of $\partial\Omega$. For completeness, we

precise that $\mathbf{n}$ is the outward normal unit vector to the surface $\partial\Omega_1$, σ is Cauchy stress tensor and u_p are prescribed or known displacement values. Meanwhile, we precise that in the scope of the present work, Dirichlet boundary condition is sufficient to lay down the aim of this paper, and we emphasize that other types of boundary conditions (Newmann, Robin or mixed) can be enforced as well since the enforcement of boundary conditions is posterior to the assembly of global matrices/Vectors. Under all the above conditions, the equilibrium of the body is governed by the boundary value problem of the form (strong form):

$$\begin{cases} \operatorname{div}(\sigma) + f = \rho\ddot{u} & \text{in } [0,T]\times\Omega \\ \sigma\mathbf{n} = t & \text{on } \partial\Omega_1 \\ u = u_p & \text{on } \Gamma \end{cases} \tag{2.2}$$

We assume for simplicity to be in small strain (ε) and in linear elasticity, such a way that the constitutive law reads as

$$\sigma = \mathcal{C} : \varepsilon \tag{2.3}$$

where $\mathcal{C}$ is a fourth-order tensor called elasticity tensor and ε is the small strain tensor given by the expression

$$\varepsilon = \frac{1}{2}(\nabla u + {}^T\nabla u) \tag{2.4}$$

Variational formulation and discretization of the problem:
Before going any further, we define the space of kinematically admissible displacements as

$$V = \left\{ u \in H^1(\Omega) \mid u = u_p \text{ on } \Gamma \right\} \tag{2.5}$$

where $H^1(\Omega)$ is the Sobolev space of functions whose first-order derivatives are bounded in the energy norm. Considering a virtual displacement δu, we multiply both sides of the first line of Eq.(2.2) and then integrate over the volume $\delta u \in V_d$

$$\int_\Omega div(\sigma) \bullet \delta u \, \mathrm{d}V + \int_\Omega f \bullet \delta u \mathrm{d}V = \int_\Omega \rho\ddot{u} \bullet \delta u \, \mathrm{d}S \tag{2.6}$$

In the previous equation, we precise that the symbol ($\bullet$) is the scalar product. Now focusing our attention on the first expression on the left-hand side of Eq.(2.6) and making use of the Stokes–Green–Ostrogradski formula, we rewrite it into

$$\int_\Omega div(\sigma) \bullet \delta u \, \mathrm{d}V = \int_{\partial\Omega_1} t \bullet \delta u \, \mathrm{d}S - \int_\Omega \sigma : \nabla \delta u \, \mathrm{d}V \tag{2.7}$$

where ∇ denotes the gradient operator. Owing to the symmetricity of the strain tensor $\varepsilon = \frac{1}{2}\left(\nabla u + {}^T\nabla u\right)$ and after transformation, the variational formulation of the boundary value reads as follows:

$$\int_\Omega \rho\ddot{u} \bullet \delta u \, \mathrm{d}V + \int_\Omega \sigma : \delta\varepsilon \, \mathrm{d}V = \int_\Omega f \bullet \delta u \, \mathrm{d}V + \int_{\partial\Omega_1} t \bullet \delta u \, \mathrm{d}S \tag{2.8}$$

Let $\Omega_h = \bigcup_{e=1}^{nel} \Omega_h^{(e)}$ be the discretized domain in the discretized space and described with data in Table 1. In MATLAB language, we make use of the function `struct` to create a structure array that groups mesh parameters and their values into a single variable called *FE_model* of type structure, each of whose fields can be retrieved using dot notation syntax (e.g., `nel = FE_model.nel`)
Let also V_h be the space of kinematically admissible displacements associated with Ω_h. The variational formulation of the problem is defined in the discretized space by

$$\begin{cases} \text{find } u_h \in V_h \text{ such that} \\ \int_{\Omega_h} \rho\ddot{u}_h \bullet \delta u_h \, \mathrm{d}V + \int_{\Omega_h} \sigma_h : \delta\varepsilon_h \, \mathrm{d}V = \int_{\Omega_h} f \bullet \delta u_h \mathrm{d}V + \int_{\partial\Omega_1} t \bullet \delta u_h \, \mathrm{d}S \end{cases} \tag{2.9}$$

Table 1. *Mesh data structure*

Name	Type	Size	Description
nel	Integer	1	Number of finite elements
m	Integer	1	Number of degrees of freedom per node
n	Integer	1	Number of nodes or vertices per element
nn	Integer	1	Number of nodes or vertices of the mesh
d	Integer	1	Dimension of the problem
q	Integer	1	Number of integration points
ndof	Integer	1	Number of degrees of freedom of all nodes
Coord	Double	(*nn*×*d*)	Array of vertex coordinates
Connect	Double	(*nel*×*n*)	Connectivity array
p	Integer	1	Shape function order

In the discretized space, the approximate global displacement of the element $e \in < 1, \ldots, nel >$ can be written in terms of the nodal displacements $\overline{u}_{i\ \in<1,\ldots,n>}^{(e)}$ and nodal shape functions $N_{i\ \in<1,\ldots,n>}$

$$u_h^{(e)} = \sum_{i=1}^{n} N_i \overline{u}_i^{(e)} = \mathcal{N}_e u_e, \tag{2.10}$$

where the shape function $\mathcal{N}_e$ of the element e is given by

$$\mathcal{N}_e = \begin{bmatrix} N_1 & 0 & 0 & N_2 & 0 & 0 & \ldots & N_n & 0 & 0 \\ 0 & N_1 & 0 & 0 & N_2 & 0 & \ldots & 0 & N_n & 0 \\ 0 & 0 & N_1 & 0 & 0 & N_2 & \ldots & 0 & 0 & N_n \end{bmatrix} \text{ in 3D}$$
$$\text{and } \mathcal{N}_e = \begin{bmatrix} N_1 & 0 & N_2 & 0 & \ldots & N_n & 0 \\ 0 & N_1 & 0 & N_2 & \ldots & 0 & N_n \end{bmatrix} \text{ in 2D} \tag{2.11}$$

With the definition of u in Eq.(2.10), it is convenient for implementation purposes to use Voigt notation to reduce the order of tensors involved by one order. As such, $\tilde{\sigma}$ and $\tilde{\varepsilon}$ are respectively Cauchy stress vector and strain vector associated with σ and ε

$$\tilde{\sigma}^T = \begin{bmatrix} \sigma_{xx} & \sigma_{yy} & \sigma_{zz} & \sigma_{yz} & \sigma_{zx} & \sigma_{xy} \end{bmatrix} \text{ in 3D}, \tilde{\sigma}^T = \begin{bmatrix} \sigma_{xx} & \sigma_{yy} & \sigma_{xy} \end{bmatrix} \text{ in 2D}$$
$$\tilde{\varepsilon}^T = \begin{bmatrix} \varepsilon_{xx} & \varepsilon_{yy} & \varepsilon_{zz} & \varepsilon_{yz} & \varepsilon_{zx} & \varepsilon_{xy} \end{bmatrix} \text{ in 3D}, \tilde{\varepsilon}^T = \begin{bmatrix} \varepsilon_{xx} & \varepsilon_{yy} & \varepsilon_{xy} \end{bmatrix} \text{ in 2D} \tag{2.12}$$

Under this notation, the strain vector of each element e can be written in terms of the nodal displacement vector as follows:

$$\tilde{\varepsilon}_e = B_e u_e \tag{2.13}$$

where B_e is the strain matrix of the element and has for components the following

$$B_e = \begin{bmatrix} N_{1,x} & 0 & 0 & N_{2,x} & 0 & 0 & \ldots & N_{n,x} & 0 & 0 \\ 0 & N_{1,y} & 0 & 0 & N_{2,y} & 0 & \ldots & 0 & N_{n,y} & 0 \\ 0 & 0 & N_{1,z} & 0 & 0 & N_{2,z} & \ldots & 0 & 0 & N_{n,z} \\ N_{1,y} & 0 & N_{1,z} & N_{2,y} & 0 & N_{2,z} & \ldots & N_{n,y} & 0 & N_{n,z} \\ N_{1,x} & N_{1,z} & 0 & N_{2,x} & N_{2,z} & 0 & \ldots & N_{n,x} & N_{n,z} & 0 \\ 0 & N_{1,y} & N_{1,x} & 0 & N_{2,y} & N_{2,x} & \ldots & 0 & N_{n,y} & N_{n,x} \end{bmatrix} \text{ in 3D}$$
$$\text{and } B_e = \begin{bmatrix} N_{1,x} & 0 & N_{2,x} & 0 & \ldots & N_{n,x} & 0 \\ 0 & N_{1,y} & 0 & N_{2,y} & \ldots & 0 & N_{n,y} \\ N_{1,y} & N_{1,x} & N_{2,y} & N_{2,x} & \ldots & N_{n,y} & N_{n,x} \end{bmatrix} \text{ in 2D} \tag{2.14}$$

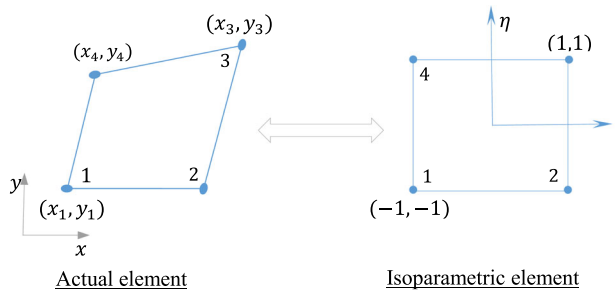


FIG. 2. Isoparametric mapping for a quadrilateral element

After replacing Eqs.(3.11), (2.10) and (2.13) into Eq.(2.9) and simplifying everything, we end up with Eq.(2.15)

$$\mathop{\mathrm{A}}_{e=1}^{nel}\left\{\int_{\Omega_e}\rho\mathcal{N}_e{}^T\mathcal{N}_e\,\mathrm{d}V\right\}\ddot{u}_e+\mathop{\mathrm{A}}_{e=1}^{nel}\left\{\int_{\Omega_e}B_e{}^T\mathcal{C}B_e\,\mathrm{d}V\right\}u_e\;=\;\mathop{\mathrm{A}}_{e=1}^{nel}\left\{\int_{\Omega_h}\mathcal{N}_e{}^T f\,\mathrm{d}V\right\}+\mathop{\mathrm{A}}_{e=1}^{nel}\left\{\int_{\partial\Omega_1}\mathcal{N}_e{}^T t\,\mathrm{d}S\right\}\tag{2.15}$$

where A is the assembly operator and is preferred over the union operator since it makes more sense in regards to the fact that the assembly is done at nodal level. Subsequently, Eq.(2.15) can be recast in matrix form as

$$\mathop{\mathrm{A}}_{e=1}^{nel}M_e\ddot{u}_e+\mathop{\mathrm{A}}_{e=1}^{nel}K_e u_e=\mathop{\mathrm{A}}_{e=1}^{nel}f_e+\mathop{\mathrm{A}}_{e=1}^{nel}t_e\tag{2.16}$$

where the expressions M_e, K_e, f_e and t_e are the element mass matrix, stiffness matrix, body force vector and applied force vector, respectively. It should also be noted that in Eq.(2.14), integrals over elements are usually approximated numerically using the Gauss quadrature rule based on the transformation between parent space and the isoparametric space as illustrated in Fig. 2 for linear quadrilateral elements, for instance.

Let $\mathcal{U}$ be the global nodal displacement vector of all the nodes (nn) of the mesh such that

$$\mathcal{U}^T=\left[\,u_1\;u_2\;u_3\;\ldots\;u_{m*nn}\,\right]^T,\;\mathcal{U}\in\mathbb{R}^{m\times nn}\tag{2.17}$$

After rearranging and assembling contributions of all elements of the mesh, we end up with the resulting global algebraic system of equations of the boundary value problem as follows:

$$M_g\,\ddot{\mathcal{U}}+K_g\,\mathcal{U}=F_{ext}\;\;\mathcal{U}\in\mathbb{R}^{m\times nn}\tag{2.18}$$

where F_{ext} is the global external force vector (sum of body force and traction force), while K_g and M_g are, respectively, the global stiffness matrix and mass matrix of the body. For further details about this type of variational formulation, the reader is referred to works in [30,31]. In the conventional computer-based calculation [17,32–35], K_g, M_g or F_{ext} are obtained by sequentially computing element matrices or vectors for each element and storing their row indices $IndexI$, column indices $IndexJ$ and values K_{values}. Once all the elements have been calculated, global matrices or vectors are assembled using MATLAB's built-in function `sparse` as shown in the workflow in Fig. 3.

The standard algorithm for computing the stiffness matrix according to the workflow shown in Fig. 3 is outlined in algorithm 1 and called **StandardGlobalStiffness**

Taking a look at the structure of algorithm 1, two main limitations emerge:

Algorithm 1 standard algorithm for computing Kg

Require: FE_model ▷ FE_model is a structure array
function $[\mathcal{K}_g]$ = **StandardGlobalStiffness**(FE_model)
  $IndexI \leftarrow IndexJ \leftarrow$ `int32(zeros(`$(m*n)^2*nel, 1$`))`
  $K_{values} \leftarrow$ `zeros(`$(m*n)^2*nel, 1$`)`
  $s \leftarrow 1$
  for $e \leftarrow 1$ to nel **do**
    $K_e \leftarrow LocalMatrices(FE\_model, \dots)$ ▷ element stiffness matrix
    $me \leftarrow FE\_model.connect(e, :)$
    for $j \leftarrow 1$ to n **do**
      for $\alpha \leftarrow 1$ to m **do**
        for $i \leftarrow 1$ to n **do**
          for $\beta \leftarrow 1$ to m **do**
            $IndexI(s) \leftarrow m*(me(i)-1)+\alpha$
            $IndexJ(s) \leftarrow m*(me(j)-1)+\beta$
            $v \leftarrow m*(i-1)+\alpha$
            $l \leftarrow m*(j-1)+\beta$
            $K_{values}(s) \leftarrow K_e(v,l)$
            $s \leftarrow s+1$
          end for
        end for
      end for
    end for
  end for
  $\mathcal{K}_g \leftarrow$ `sparse(`$IndexI, IndexJ, K_{values}$`)`; $\mathcal{K}_{values} \leftarrow [\,]$
end function

- The non-vectorized for-loop over elements of the mesh: The elementwise-based computation of element stiffness matrices and their indices ($IndexI$ and $IndexJ$) is computationally inefficient since it does not make optimal use of MATLAB's vector operation capabilities;
- Apart from the outermost for-loop, four nested for-loops are needed to compute row and column indices.

Several assembly procedures have been proposed like those in [15,17,29] to get rid of at least one of the above-mentioned limitations. Precisely, while authors in the former use matrix rearrangements to reduce memory usage and achieve a better performance, the authors in the latter take advantage of affine transformations to formulate their algorithm.

3. Proposed vectorized algorithm

3.1. Formulation

First and foremost, it is important to keep in mind that serial calculation of finite element terms can be classified into two categories: non-vectorized serial calculation and vectorized serial calculation. Due to integrals involved in the calculation of finite element terms, each method can be performed either implicitly, by means of numerical integration schemes like Gauss quadrature rule, or explicitly (exact integration) as shown in Fig. 4. However, the latter approach is tedious and irrelevant for finite elements with approximation order higher than 1 ($p > 1$).

Now, we are going to introduce an implicit vectorized algorithm based on the branch highlighted in orange in Fig. 4. The workflow of this algorithm in linear elasticity is depicted in Fig. 5.

In the following, we describe the whole procedure and the steps it comprises.

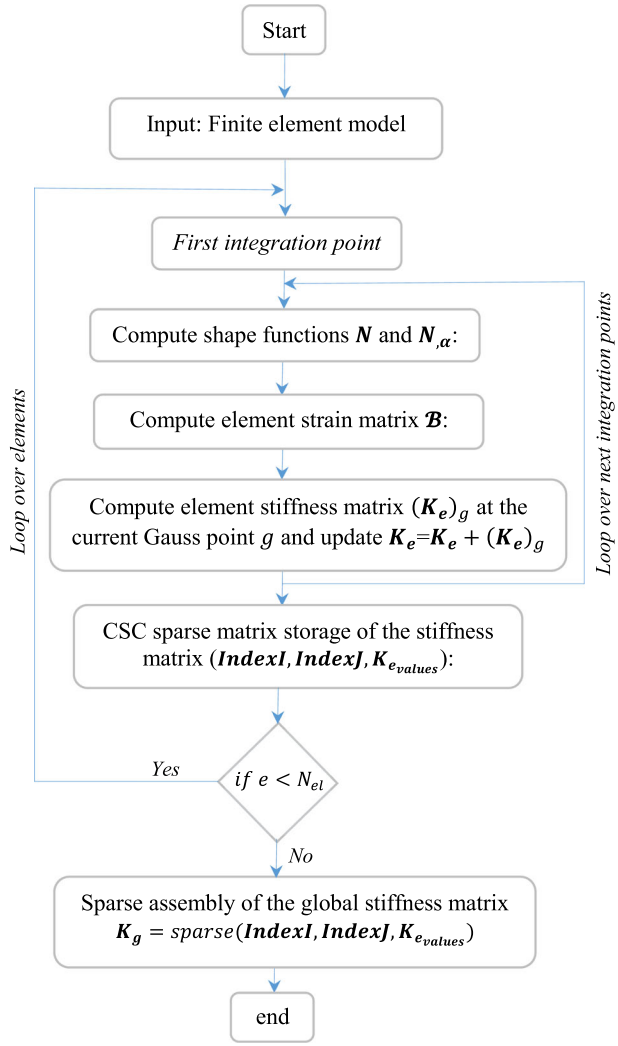


FIG. 3. Standard workflow for computing the stiffness matrix

(a) Firstly, we need to gather values of nodal shape functions and their derivatives in the isoparametric space so that they are computed only once. Hence in the subroutine **GaussQuadrature**, Gauss quadrature data (number of Gauss points, their respective coordinates and weights) are respectively calculated in the isoparametric space and stored in the expressions $nPoint$, $weight$ and Gq,with

$$Gq = \begin{bmatrix} \xi_1 & \xi_2 & \dots & \xi_q \\ \eta_1 & \eta_2 & \dots & \eta_q \\ \kappa_1 & \kappa_2 & \dots & \kappa_q \end{bmatrix} \text{ in 3D} \tag{3.1}$$

where Gq is of size $d \times q$. When it comes to 1D elements, Gq is reduced to only the first line, whereas only the first two lines are relevant for 2D elements.

(b) With Gauss quadrature data at hand, we proceed with the computation of values of shape functions and their derivatives at each integration point, then store them as NI and $dN\_d\theta$, respectively. In the subroutine **ShapeFunction**, basis functions of some basic 2D and 3D isoparametric elements are

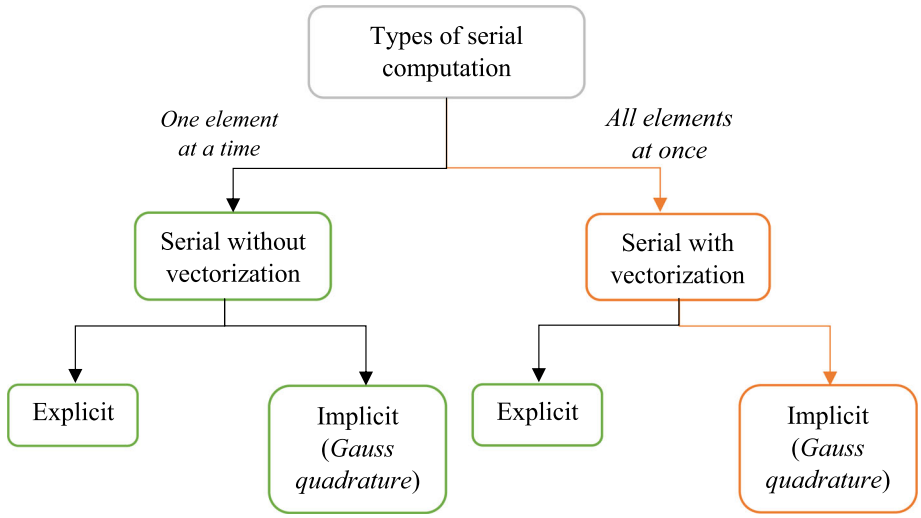


FIG. 4. Types of serial computation in FEM

defined with help of symbolic operations in MATLAB, which enables to play around with derivatives as well as corresponding values at any given integration point without much effort. Outputs read as

$$NI = \begin{bmatrix} N_1(\theta_1) & \dots & N_1(\theta_q) \\ N_2(\theta_1) & \dots & N_2(\theta_q) \\ \dots & \dots & \dots \\ N_n(\theta_1) & \dots & N_n(\theta_q) \end{bmatrix} \tag{3.2}$$

where q is the number of Gauss quadrature points, $dim\,(NI) = d{\times}q$ and $\theta_i \in {}_{<1,2,\dots,\ q>}$ are the coordinates of quadrature points in the isoparametric space

$$\theta_i = \begin{cases} \xi_i & \text{in 1D} \\ (\xi,\eta)_i & \text{in 2D} \\ (\xi,\eta,\kappa)_i & \text{in 3D} \end{cases} \tag{3.3}$$

Similarly, $dN\_d\theta$ is a multidimensional matrix that stores values of derivative of basis functions (with respect to the isoparametric space) at all quadrature points in such a way that values at each integration point i are stored in the slice i of the multidimensional matrix $dN\_d\theta$ and are calculated from the expression (for 3D problems) below

$$dN\_d\theta(:,:,i) = \begin{bmatrix} \frac{\partial N_1}{\partial \xi}(\theta_i) & \frac{\partial N_1}{\partial \eta}(\theta_i) & \frac{\partial N_1}{\partial \eta}(\theta_i) \\ \frac{\partial N_2}{\partial \xi}(\theta_i) & \frac{\partial N_2}{\partial \eta}(\theta_i) & \frac{\partial N_2}{\partial \eta}(\theta_i) \\ \dots & \dots & \dots \\ \frac{\partial N_n}{\partial \xi}(\theta_i) & \frac{\partial N_n}{\partial \eta}(\theta_i) & \frac{\partial N_n}{\partial \eta}(\theta_i) \end{bmatrix} \tag{3.4}$$

So $dim(dN\_d\theta) = n{\times}d{\times}q$, wherein n is the number of basis functions at each finite element and d the dimension of the problem. From the above form, one can deduce $dN\_d\theta$ in 1D and 2D.

(c) The next step is to get rid of the for-loop over elements. To this end, we define the multidimensional strain matrix $\mathcal{B}$ that contains strain matrices of every single element stored in a specific slice or page with index i.

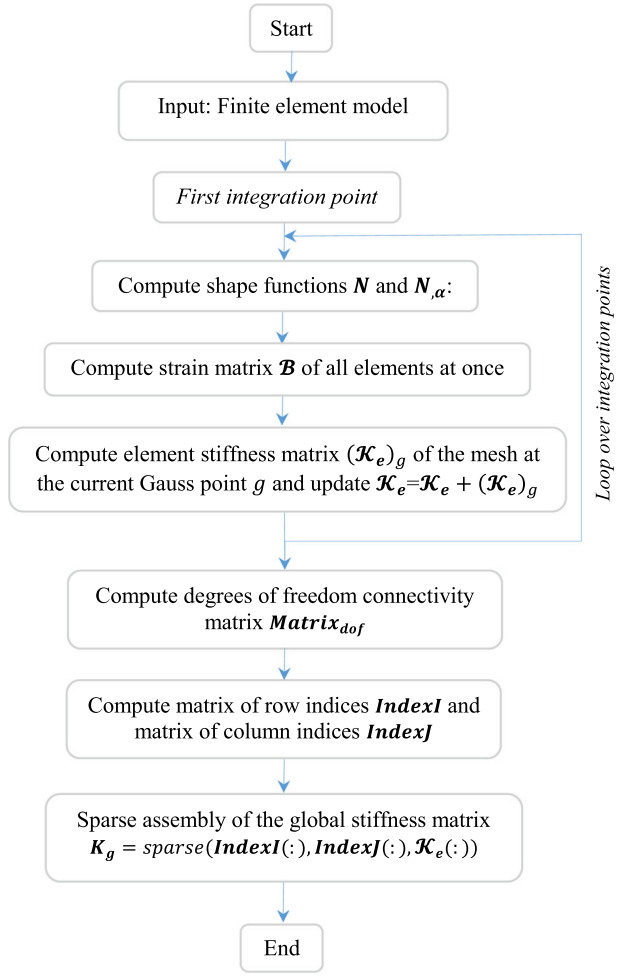


FIG. 5. Workflow of the proposed algorithm

Since $dN\_d\theta$ is the derivative of the nodal basis functions in the isoparametric space, we have to compute the Jacobian J of the transformation.

For every integration point, the following steps are defined to compute $\mathcal{B}$ and the corresponding multidimensional element stiffness matrix $\mathcal{K}_e$:

(i) From $dN\_d\theta$, we extract the derivative of the shape function at the current Gauss point and call MATLAB built-in function *pagetimes* to calculate the Jacobian of the transformation J by the formula

$$J = \texttt{pagetimes}(dN\_d\theta(:,:,i), \texttt{'none'}, Pts, \texttt{'transpose'}) \tag{3.5}$$

where J is of dimension $d{\times}d{\times}nel$.

(ii) Afterward, the derivative of the basis function matrix dN_dX in the element space is computed

$$dN\_dX = \texttt{pagemldivide}(J, dN\_d\theta) \tag{3.6}$$

where dN_dX is of dimension $d{\times}n{\times}nel$.

(iii) Then after pre-allocating memory for storing $\mathcal{B}$, its values are computed and filled in only one for-loop of length n such that at the node $s \in < 1, 2, \ldots, n >$ of all elements, the component

$\mathcal{B}(:, d*(s-1)+1 : d*s, :)$ is in 3D given by

$$\mathcal{B}(:, d*(s-1)+1 : d*s, :) = \begin{bmatrix} dN\_dX(1,s,:) & 0_{1\times 1\times nel} & 0_{1\times 1\times nel} \\ 0_{1\times 1\times nel} & dN\_dX(2,s,:) & 0_{1\times 1\times nel} \\ 0_{1\times 1\times nel} & 0_{1\times 1\times nel} & dN\_dX(3,s,:) \\ dN\_dX(2,s,:) & dN\_dX(1,s,:) & 0_{1\times 1\times nel} \\ 0_{1\times 1 x nel} & dN\_dX(3,s,:) & dN\_dX(2,s,:) \\ dN\_dX(3,s,:) & 0_{1\times 1\times nel} & dN\_dX(1,s,:) \end{bmatrix} \tag{3.7}$$

The component $\mathcal{B}(:, d*(s-1)+1 : d*s, :)$ for 2D problems is deduced following the same token as in Eq. 3.6.

$$\mathcal{B}(:, d*(s-1)+1 : d*s, :) = \begin{bmatrix} dN\_dX(1,s,:) & 0_{1\times 1\times nel} \\ 0_{1\times 1\times nel} & dN\_dX(2,s,:) \\ dN\_dX(2,s,:) & dN\_dX(1,s,:) \end{bmatrix} \tag{3.8}$$

The matrix $\mathcal{B}$ is fully constructed in a single for-loop by calling the function **Bs** written Algorithm 4.

(iv) Now let $detJ$ be the determinant of the Jacobian matrix J. In order to avoid elementwise computation of $detJ$, we introduce the following vectorized form defined in the function **DetJacobian** of Algorithm 2.

(v) Update the multidimensional element stiffness matrix $\mathcal{K}_e$ given in 3D

$$\mathcal{K}_e = \mathcal{K}_e + \texttt{pagemtimes}(\mathcal{B}, \texttt{'transpose'}, \texttt{pagemtimes}(\mathcal{C}, \mathcal{B}).* detJ * weight(i), \texttt{'transpose'}) \tag{3.9}$$

where the determinant of the Jacobian $detJ$ is of size $1\times 1\times nel$ and $\mathcal{K}_e$ is of size $(m*n)\times(m*n)\times nel$. The row vector $weight$ of size $1\times q$, is the vector of Gauss quadrature weights. In 2D, the second expression on the right-hand side of Eq.(3.9) will be multiplied by the thickness of the elements.

d) Once we obtain stiffness matrix of all elements $\mathcal{K}_e$ for all integration points, the next step is to construct row and column indices ($IndexI$ and $IndexJ$).
While the row and column indices in Algorithm 1 and others existing in the literature are computed concurrently with the element stiffness values, we propose a scheme in Algorithm 3, wherein $IndexI$ and $IndexJ$ are calculated from the degrees of freedom connectivity matrix $Matrix\_dof$, obtained in a single for-loop over m. For memory sake, $Matrix\_dof$, $IndexI$ and $IndexJ$ are defined as 32-bit integers.

It should be noted that in algorithm 3, the operations $Matrix\_dof(:)$ and $Matrix\_dof(:)'$ convert the matrix $Matrix\_dof$ into a column vector and a row vector, respectively.

Remark 3.1. For any element e with domain $\Omega_h^{(e)} \subset \Omega_h \subset \mathbb{R}^d$, row and column indices ($IndexI_e$ and $IndexJ_e$) of its stiffness components can be straightforwardly obtained from its degrees of freedom connectivity matrix after a single $for-loop$ over the number of degrees of freedom per node.

Algorithm 2 Determinant of the Jacobian J

Require: J, nel, d
function $[detJ]$ = **DetJacobian**(J, nel, d)
  $Mat \leftarrow$ `reshape`$(J, [\,], nel)$
  if $d == 2$ **then** ▷ in 2D
    $detJ \leftarrow Mat(1,:).*Mat(4,:) - Mat(2,:).*Mat(3,:)$
  else if $d == 3$ **then** ▷ in 3D
    $detJ \leftarrow Mat(1,:).*(Mat(5,:).*Mat(9,:) - Mat(6,:).*Mat(8,:)) - Mat(4,:).*(Mat(2,:).*Mat(9,:) - Mat(3,:).*Mat(8,:)) + Mat(7,:).*(Mat(2,:).*Mat(6,:) - Mat(3,:).*Mat(5,:))$
  end if
end function

Algorithm 3 Computation of $Matrix\_dof$

Require: FE_model, ▷ model data structure
$Matrix\_dof \leftarrow$ `int32(zeros(`$nel, n*m$`)')` ▷ Initilization of $Matrix\_dof$
$k \leftarrow (1:n)'$
for $a \leftarrow 1$ to m **do**
  $Matrix\_dof(m*(k-1)+a,:) \leftarrow (m*(FE\_model.connect(:,k)-1)+a)'$
end for
$IndexJ \leftarrow$ `int32(repmat(`$Matrix\_dof(:)', m*n, 1$`))`
$IndexI \leftarrow$ `int32(repmat(`$Matrix\_dof, m*n, 1$`))`

e) As a final step, after constructing $IndexI$, $IndexJ$, and $\mathcal{K}_e$, the global stiffness matrix is assembled using MATLAB's built-in function `sparse` through the command below

$$\mathcal{K}_g \leftarrow \texttt{sparse}(IndexI\texttt{(:)}, IndexJ\texttt{(:)}, \texttt{reshape}(\mathcal{K}_e, [\,], 1)) \tag{3.10}$$

To wrap it all up, the summary of the proposed method is outlined in Algorithm 4 under the function **PropGlobalStiffness**. In order to reduce as much as possible memory usage, some variables are emptied at certain points throughout the execution (see lines 13, 15, 18, 26 and 27).

Algorithm 4 Proposed algorithm for computing the stiffness matrix $\mathcal{K}_g$

Input: FE_model ▷ FE_model is a structure array
1: **function** $[\mathcal{K}_g] =$ **PropGlobalStiffness**(FE_model)
2: $[Gq, weight, q] \leftarrow$ **GaussQuadrature**(FE_model) ▷ Get Gauss quadrature data
3: $[NI, dN\_d\theta] \leftarrow$ **ShapeFunction**(FE_model) ▷ Get shape functions and their derivatives
4: $\mathcal{K}_e \leftarrow$ `zeros(`$m*n, m*n, nel$`)` ▷ Initilization of $\mathcal{K}_e$
5: $CoordI \leftarrow$ `@(`x`)[`$FE\_model.Coord((FE\_model.connect(x,:))', :)$`]'`
6: $Pts \leftarrow$ `reshape(`$CoordI(1:nel), ndof, nn, [\,]$`)` ▷ Coordinates of elements' nodes
7: **for** $i \leftarrow 1$ to q **do** ▷ loop over integration points
8: $J \leftarrow$ `pagetimes(`$dN\_d\theta(:,:,i)$`, 'none',` Pts`, 'transpose')`
9: $dN\_dX \leftarrow$ `pagemldivide(`$J, dN\_d\theta(:,:,i)$`)`
10: **for** $s \leftarrow 1$ to n **do**
11: Compute $\mathcal{B}(:, d*(s-1)+1:d*s, :)$ ▷ see Eq.(3.7) and (3.8)
12: **end for**
13: $dN\_dX \leftarrow [\,]$ ▷ clear variable
14: $detJ \leftarrow$ **DetJacobian**(J, nel, d) ▷ see Algorithm 2
15: $J \leftarrow [\,]$ ▷ clear variable
16: Update $\mathcal{K}_e$ ▷ see Eq.(3.9)
17: **end for**
18: $[\mathcal{B}, Pts, detJ] =$ `deal(`$[\,]$`)` ▷ clear variables
19: $Mat\_dof \leftarrow$ `int32(zeros(`$nel, n*m$`)')` ▷ Initilization of Mat_dof
20: $k \leftarrow (1:n)'$
21: **for** $\beta \leftarrow 1$ to m **do**
22: $Mat\_dof(m*(k-1)+\beta,:) \leftarrow (m*(connect(:,k)-1)+\beta)'$
23: **end for**
24: $IndexJ \leftarrow$ `int32(repmat(`$Mat\_dof(:)', m*n, 1$`))`
25: $IndexI \leftarrow$ `int32(repmat(`$Mat\_dof, m*n, 1$`))`
26: $Mat\_dof \leftarrow [\,]$ ▷ clear variable
27: $\mathcal{K}_g \leftarrow$ `sparse(`$IndexI\texttt{(:)}, IndexJ\texttt{(:)}, \texttt{reshape}(\mathcal{K}_e, [\,], 1)$`);` $\mathcal{K}_e \leftarrow [\,]$
28: **end function**

For sake of simplicity, some lines in the above algorithm have intentionally been omitted particularly those related to the retrieval of mesh data (nel, n, d, m, etc.) from the structure array FE_model using dot notation syntax (e.g., `nel = FE_model.nel`).

3.2. Extension of the proposed vectorized algorithm for other types of problems

3.2.1. For functionally graded materials. Recalling that in the case of functionally graded materials, there is a spatial variation of material properties due to spatial changes in the material's composition and/or microstructure. Basically, Young's modulus E and/or Poisson's ratio can be described as a function of the position of the observation point $M \in \Omega$. So, is the elasticity tensor $\mathcal{C}(M)$ such that the constitutive behavior reads as

$$\sigma = \mathcal{C}(M) : \varepsilon \tag{3.11}$$

The numerical calculation of the element stiffness matrix within such materials is quite straightforward after computing the element elasticity tensor at the current Gauss quadrature point g. Knowing that the spatial position of each Gauss quadrature point can be computed from element nodal coordinates using shape functions as given in Eq. (3.12)

$$x_g^{(e)} = \sum_{i=1}^{n} N_i x_i^{(e)} = \mathcal{N}_e x_e; \qquad y_g^{(e)} = \sum_{i=1}^{n} N_i y_i^{(e)} = \mathcal{N}_e y_e; \qquad z_g^{(e)} = \sum_{i=1}^{n} N_i z_i^{(e)} = \mathcal{N}_e z_e \tag{3.12}$$

Owing to the fact that the spatial expression of E and/or ν is a known parameter of the problem, E and/or ν at the current Gauss point g can be computed in 3D as per Eq. (3.13)

$$E(M_g) = E\left(x_g^{(e)}, y_g^{(e)}, z_g^{(e)}\right); \qquad \nu(M_g) = \nu\left(x_g^{(e)}, y_g^{(e)}, z_g^{(e)}\right) \tag{3.13}$$

For a basic illustration of the current vectorized algorithm, let us assume a case with ν constant, $E(M)$ varying with respect to x and considering a state of plane stress, the elasticity tensor is given in 3D by the expression

$$\mathcal{C}(M_g) = \frac{E(M_g)}{(1+\nu)(1-2\nu)} \begin{bmatrix} 1-\nu & \nu & \nu & 0 & 0 & 0 \\ \nu & 1-\nu & \nu & 0 & 0 & 0 \\ \nu & \nu & 1-\nu & 0 & 0 & 0 \\ 0 & 0 & 0 & \frac{1}{2}(1-2\nu) & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{1}{2}(1-2\nu) & 0 \\ 0 & 0 & 0 & 0 & 0 & \frac{1}{2}(1-2\nu) \end{bmatrix} \tag{3.14}$$

or simply

$$\mathcal{C}(x) = E(x)\,\mathcal{C}_b \tag{3.15}$$

where $\mathcal{C}_b$ is the material tensor for a unit Young's modulus. $E(x)$ at the Gauss point g for all elements of the model is computed using a single command

$$E(x_g) = \texttt{pagetimes}(E(\texttt{pagetimes}(Pts(1,:,:), Ng)), \mathcal{C}_b) \tag{3.16}$$

In the previous expression, the user defines the analytical expression of $E(x)$ using MATLAB function handle. Ng is a column vector of the element nodal shape function at the current integration point.

Following this same token, the material tensor of any functionally graded material can be calculated in a vectorized manner without much effort. Provided the analytical expression of $E(x)$ and/or $\nu(x)$ is known in advance.

3.2.2. For plate problems with Mindlin formulation. Mindlin plate formulation proposes an improvement to Kirchhoff–Love plate theory to take into account shear deformations through the thickness of the thick plates. In this formulation, the section initially perpendicular to the middle surface before the deformation

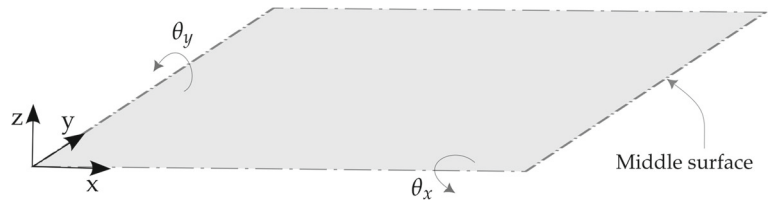


FIG. 6. Illustration of Mindlin plate formulation

does not remain perpendicular to the middle plane after the deformation, such that the displacement field can be defined as follows:

$$u = z\theta_x; \qquad v = z\theta_y; \qquad w = w_0; \tag{3.17}$$

where θ_x and θ_y are the rotations with respect to x and y axes, respectively. Similar to Kirchhoff–Love's plate theory, the study of the problem is described from the middle surface as illustrated in Fig. 6.

With this approximation, bending component ϵ_f of the strain tensor reads as

$$\epsilon_x = \frac{\partial u}{\partial x} = z\frac{\partial \theta_x}{\partial x} \tag{3.18a}$$

$$\epsilon_y = \frac{\partial v}{\partial y} = z\frac{\partial \theta_y}{\partial y} \tag{3.18b}$$

$$\gamma_{xy} = \frac{\partial u}{\partial y} + \frac{\partial v}{\partial x} = z\left(\frac{\partial \theta_y}{\partial x} + \frac{\partial \theta_x}{\partial y}\right), \tag{3.18c}$$

while shear component ϵ_c is given by

$$\gamma_{xz} = \frac{\partial w}{\partial x} + \frac{\partial u}{\partial z} = \frac{\partial w}{\partial x} + z\theta_x \tag{3.19a}$$

$$\gamma_{yz} = \frac{\partial w}{\partial y} + \frac{\partial v}{\partial z} = \frac{\partial w}{\partial y} + z\theta_y \tag{3.19b}$$

In the hypothesis of small strains and homogeneous material, the bending part of Cauchy stress σ reads as

$$\sigma_f = \mathcal{C}_f \epsilon_f \tag{3.20}$$

where $\mathcal{C}_f$ is defined as

$$\mathcal{C}_f = \frac{E}{(1-\nu^2)}\begin{bmatrix} 1 & \nu & \nu \\ \nu & 1 & \nu \\ 0 & 0 & \dfrac{1-\nu}{2} \end{bmatrix} \tag{3.21}$$

Shear stress components are given by

$$\sigma_c = \mathcal{C}_c \epsilon_c \tag{3.22}$$

where $\mathcal{C}_c$ is defined as

$$\mathcal{C}_c = \kappa\frac{E}{2\,(1+\nu)}\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \tag{3.23}$$

where κ is the shear correction factor. Now, since the vector field $U = (u, v, w)$ can be fully characterized for given $d^T = (\theta_x, \theta_y, w)$ thanks to Eq. (3.17), we assume in the discretized space, the following

interpolation

$$\theta_x = \sum_{i=1}^{n} N_i \theta_{xi}; \qquad \theta_y = \sum_{i=1}^{n} N_i \theta_{y_i}; \qquad w = \sum_{i=1}^{n} N_i w_i \tag{3.24}$$

In virtue of this general approximation, the bending B_f and shear B_c strain matrices can be readily written out.

$$B_f = \begin{bmatrix} N_{1,x} & 0 & 0 \dots N_{n,x} & 0 & 0 \\ 0 & N_{1,y} & 0 \dots 0 & N_{n,y} & 0 \\ N_{1,y} & N_{1,x} & 0 \dots N_{n,y} & N_{n,x} & 0 \end{bmatrix}; B_c = \begin{bmatrix} N_1 & 0 & N_{1,x} \dots N_n & 0 & N_{n,x} \\ 0 & N_1 & N_{2,y} \dots 0 & N_n & N_{n,y} \end{bmatrix} \tag{3.25}$$

After replacing Eqs.(3.25), (3.24), (3.22) and Eq.(3.20) into Eq.(2.9) and simplifying everything, we end up with Eq.(3.26)

$$\begin{aligned} \mathop{\mathrm{A}}_{e=1}^{nel} \left\{ \int_{\mathrm{S}_e} \rho \mathcal{N}_e{}^T \overline{M} \mathcal{N}_e \, \mathrm{d}V \right\} \ddot{d}_e + \mathop{\mathrm{A}}_{e=1}^{nel} \left\{ \frac{h^3}{12} \int_{\mathrm{S}_e} B_f{}^T \mathcal{C}_f B_f \, \mathrm{d}S \right\} d_e + \mathop{\mathrm{A}}_{e=1}^{nel} \left\{ \alpha h \int_{\mathrm{S}_e} B_c{}^T \mathcal{C}_c B_f \, \mathrm{d}S \right\} d_e \\ = \mathop{\mathrm{A}}_{e=1}^{nel} \left\{ \int_{\mathrm{S}_e} \mathcal{N}_e{}^T f \, \mathrm{d}S \right\} + \mathop{\mathrm{A}}_{e=1}^{nel} \left\{ \int_{\partial\Omega_1} \mathcal{N}_e{}^T t \, \mathrm{d}S \right\} \end{aligned} \tag{3.26}$$

where $\overline{M}$ reads as

$$\overline{M} = \begin{bmatrix} \frac{h^3}{12} & 0 & 0 \\ 0 & \frac{h^3}{12} & 0 \\ 0 & 0 & h \end{bmatrix} \tag{3.27}$$

After assembling, we end up with the system of partial differential equations below

$$M_g \ddot{d} + K_g d = F_{ext} \quad d \in \mathbb{R}^{nn*m} \tag{3.28}$$

It is important to bear in mind that in Eq.(3.28), the shear-related component of the stiffness matrix is calculated using reduced integration. This prevents shear-locking problems from occurring.

The vectorization of Eq. (3.28) is conducted similarly, as outlined in the general formulation of the proposed algorithm in the previous section, in particular, B_f and B_c. Here, the functions **getBplate**, **PropStiffnessPlate**, **PropGlobalForce** were created to implement Mindlin plate theory. The detailed structure of these functions is provided in the Appendix.

4. Numerical experiments:

This section is devoted to the verification and validation of the proposed vectorized algorithm for linear elastic boundary value problems.

- For verification and performance analysis, the commercial software Simulia ABAQUS [38] is used to model and analyze a linear elastic L-shaped domain (see Sect. 4.1), a clamped plate based on Mindlin theory(see Sect. 4.3), a functionally graded membrane in tension(see Sect. 4.4) and a functionally graded 3D beam (see Sect. 4.5) and compare the results and performances with those obtained by the proposed method.
- In order to numerically demonstrate the novelty of the proposed algorithm over two other existing MATLAB-based vectorized algorithms [15,29], a computational time and memory usage-based performance test is conducted in Sect. 4.2. In each method, the global stiffness matrix of a 2D beam then a 3D plate problem is calculated on a set of meshes (from coarse to fine), and the associated execution time and memory usage are recorded and compared.

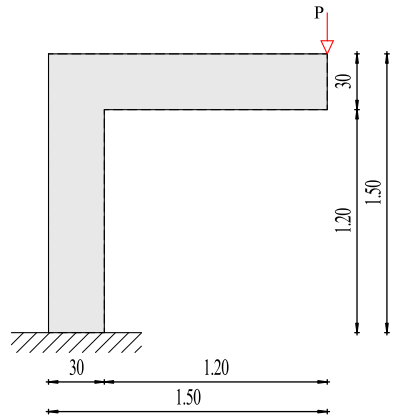


FIG. 7. Schematic drawing of the L-shaped stucture

TABLE 2. *Set of linear triangle element meshes*

Mesh ID	*nel*	*nn*
1	1 386	780
2	16 200	8 401
3	101 250	51 376
4	259 200	130 801
5	720 000	362 001

**nel* and *nn* are the number of elements and nodes, respectively

4.1. Example 1: Elastic L-shaped structure

We consider the 5-cm-thick linear elastic L-shaped structure with the geometry shown in Fig. 7 and whose material properties are Young's modulus E= 210 GPa, Poisson's ratio ν= 0.3 and density ρ = 7850 kg/m^3, respectively. This structure is loaded with a concentric force of intensity 200KN oriented toward the negative y-axis as represented in Fig. 7. For simplicity, we restrict the problem to a plane-stress analysis.

The computing resource used for this simulation is a gaming desktop equipped with an AMD Ryzen 3 1200 processor (running at a clock frequency of 3.1GHz), a total RAM of 24GB, and a Windows 10 Pro 64-bit operating system. On this machine are installed Simulia ABAQUS/CAE 2020 [38] and MATLAB 2024a, which we used to carry out performance comparisons.

First, ABAQUS was used to create the geometry, define material properties and load, enforce boundary condition on the edge y=0, and generate the set of P_1-triangle meshes (linear triangles) with data reported in Table 2.

Subsequently, the meshes created in Simulia ABAQUS were imported into MATLAB as topology inputs. Afterward, the simulations using ABAQUS and the proposed algorithm in MATLAB were carried out, and the result of the vertical displacement U_z based on the first mesh is displayed in Fig. 8.

In both simulations, we note that the vertical displacement is the same. Having verified the accuracy of the proposed method, we move on with the comparison of the computational time in ABAQUS to that of the proposed algorithm based on the set of meshed listed in Table 2.

The results show that by reducing the global matrix construction time by 10 times using the proposed method (see Fig. 9a), the total runtime needed for the FEM analysis is reduced by 5 times (see Fig. 9b).

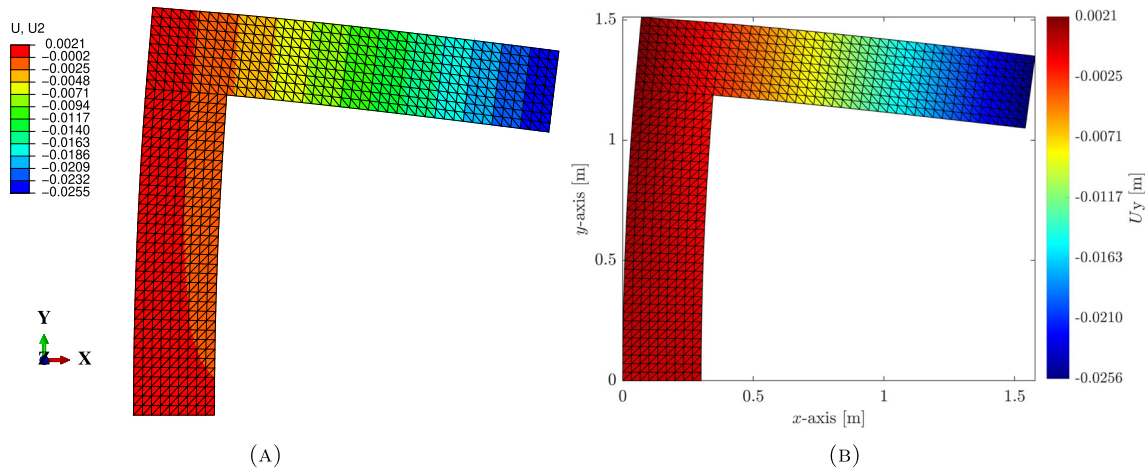


FIG. 8. Visualization of the displacement along y-axis: **a** in ABAQUS/CAE; **b** with proposed algorithm in MATLAB

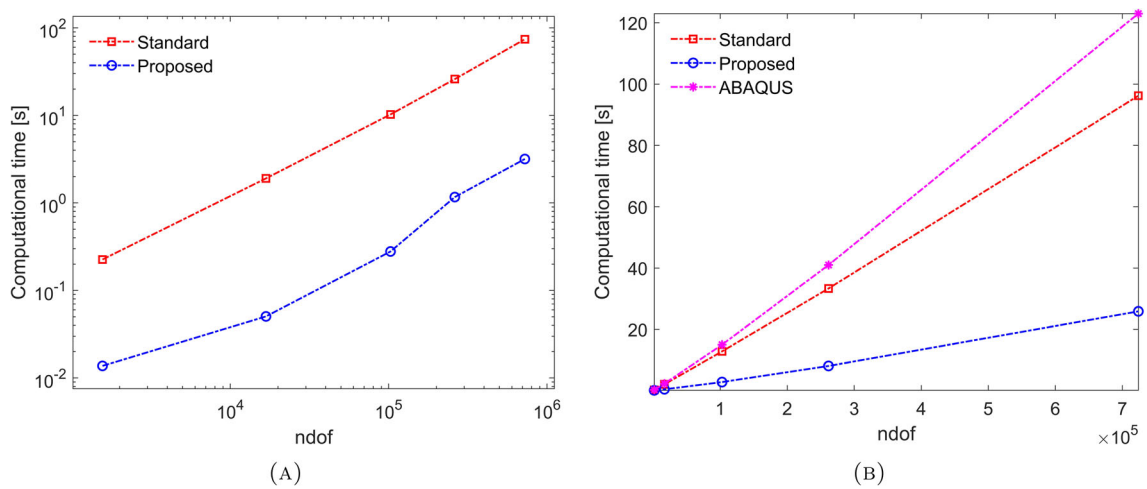


FIG. 9. Runtime performance analysis of the L-shaped structure: **a** CPU time for assembling K_g as per the *Standard* and *proposed* algorithm; **b** Total CPU time of the analysis (global matrix/force construction + solving the system of the equation)

4.2. Example 2: Comparison with the existing vectorized algorithms

In this part, we have only reported information related to the objective of measuring and comparing the execution times for calculating the global stiffness matrix using the algorithms under consideration for a selection of 2D and 3D problems. For both examples, the simulations were conducted on a computer with specifications listed in Table 3 below

4.2.1. Case 1: 2D beam problem. In order to make an assessment of the performance of the proposed vectorized algorithm, we considered two other vectorized algorithms together with the algorithm 1 as comparison basis. For this purpose, a set of 6 P_1 triangle element meshes was considered as topology inputs. The corresponding data are reported in Table 4.

Table 3. *Computer specifications*

Variable	specification
Computer model	Dell G5 15 Gaming (5500)
Processor	
type	Intel Core i7
frequency	2.6/5GHz
generation	Comet Lake -10th generation
model	10750H
number of cores	6
number of threads	12
Memory	
RAM	16GB
Hard drive	1TB SSD
Operating system	Windows 10

Table 4. *Set of linear triangle element meshes*

Mesh ID	*nel*	*nn*
1	278 202	140 000
2	627 302	315 000
3	1 413 452	708 750
4	2 514 602	1 260 000
5	5 661 902	2 835 000
6	10 069 202	5 040 000

*nel and nn are the number of elements and nodes, respectively

We precise that here the number of degrees of freedom per node is $m = 2$; so $ndof = m * nn$ for each mesh ID in Table 4.

The codes for implementing the two other vectorized algorithms [15,29] considered in this study are obtained from authors' platforms [39,40]. Hereafter, these algorithms are referred to respectively as *Cuvelier et al* and *Anjam et al*. Along with that we also used the external function `fsparse` provided with the package `FAST` in [41] and available on the first author's github repository [42]. With this function, a sparse matrix can be constructed from the triplet $IndexI, IndexJ, Kvalues$ more quickly than with MATLAB's built-in `sparse`.

After running the programs using the standard algorithm, the proposed method (first with `sparse` then `fsparse`), *Cuvelier et al* and *Anjam et al* algorithms on the aforementioned set of 2D meshes, the recorded computational times are depicted in Fig. 10.

It is worth noting that the assembly with sparse, named *proposed*, is represented by the cyan dashed line with a star marker.

It can be seen in Fig. 10b that the slope of the line joining the cloud of points for each algorithm is nearly the same. However, it is much more convenient to compare them on the basis of the speedup ratio (see Fig. 11) with respect to the standard algorithm.

In view of the boxplot in Fig. 11b, we find out that the dispersion with the proposed algorithm using the assembly functions `sparse` and `fsparse` is nearly the same. While the proposed scheme with `sparse` is about 9x faster than the standard algorithm, *Cuvelier et al*'s approach is circa 6x faster and *Anjan et al*'s method is about 4x faster. On the other hand, when the present algorithm is executed with `fsparse` instead, the program performance turns out to be 15x faster.

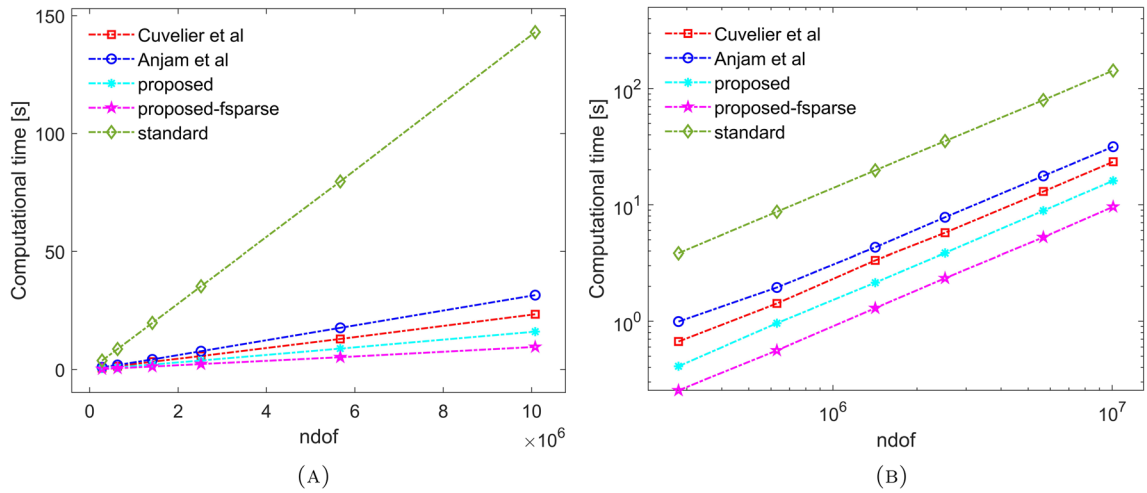


FIG. 10. Visualizaton of $\mathcal{K}_g$ computational time for a set of 2D meshes with P_1 triangle elements: **a** runtime versus the number of degrees of freedom; **b** runtime versus $ndof$ in log–log scale

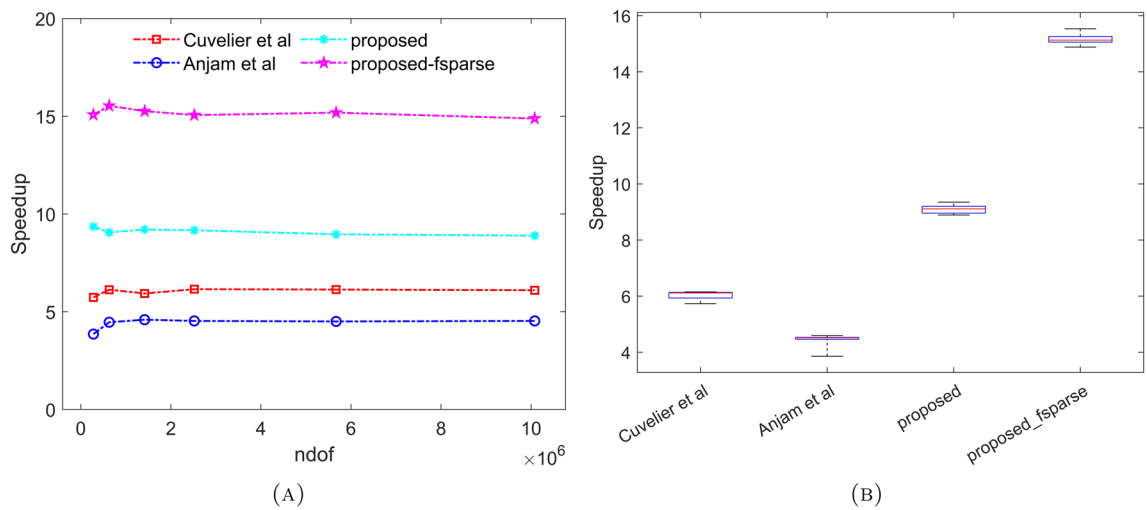


FIG. 11. Visualizaton of the computational speedup with respect to the standard algorithm for a set of 2D meshes with P_1 triangle elements: **a** speedup versus the number of degrees of freedom; **b** boxplot of the speedup of the algorithms

Besides that, the analysis of memory usage in MATLAB profiler reveals that *Cuvelier et al*'s approach is the least RAM-intensive compared to the present method, standard method and *Anjam et al*'s approach as illustrated in Fig. 12.

4.2.2. Case 2: 3D beam problem. Similarly, as we proceeded in section 4.2.1, we consider a simply supported plate discretized with a set of 6 P_1 tetrahedral element meshes whose information are reported in Table 5

We precise that here the number of degrees of freedom per node is $m = 3$; so $ndof = m * nn$ for each mesh ID in Table 5.

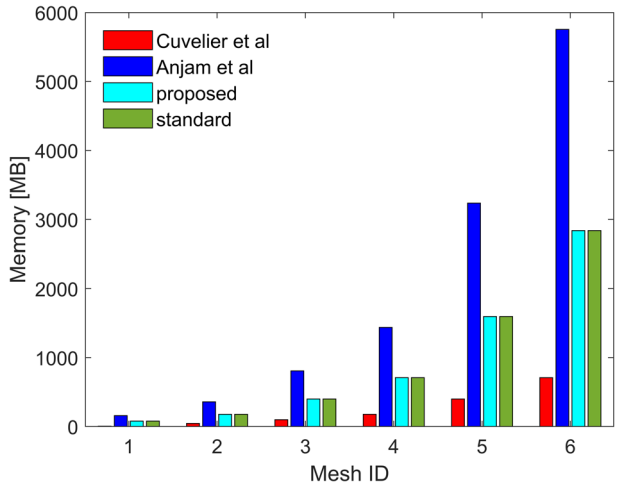


FIG. 12. Memory usage for computing $\mathcal{K}_g$ per algorithm with respect to mesh ID for the selected 2D problem

TABLE 5. *Set of linear tetrahedral element meshes*

Mesh ID	*nel*	*nn*
1	48 114	10 000
2	175 224	33 750
3	431 034	80 000
4	860 544	156 250
5	1 508 754	270 000
6	2 728 674	367 500

*nel and nn are the number of elements and nodes, respectively

After execution with each algorithm as proceeded for the 2D case, we obtained the runtime depicted in Fig. 13.

We note in Fig. 14a that the performance of the proposed algorithm used with `sparse` decreases from mesh 1 to 3 but stabilizes with a speedup of circa 3.8 for larger meshes. Although this behavior is observed, the speedup is at least 1.5x higher than that of *Cuvelier et al* and 2x higher than that of *Anjam et al*, whereas the use of `fsparse` enables to maintain an approximately constant speedup of 8 regardless of mesh size. A more comprehensive visualization of the speedup with respect to the standard algorithm is illustrated with the boxplot in Fig. 14b.

On the memory front, it can be seen in Fig. 15 that the proposed method uses almost as much memory as the standard method.

4.3. Example 3: clamped thick plate

Let us assume the isotropic and homogeneous thick plate depicted Fig. 16 and with mechanical properties $E = 10920$ MPa and $\nu = 0.3$. This plate is subjected to a uniformly distributed transverse pressure of intensity 1MN applied on its top surface. The plate is clamped at its four edges. This example was drawn out from the reference work by Ferreira in [43].

In order to carry out a comprehensive comparison between the performance of the proposed algorithm and that of ABAQUS based on Mindlin plate theory, a group of six meshes of type 4-node elements(QUA4

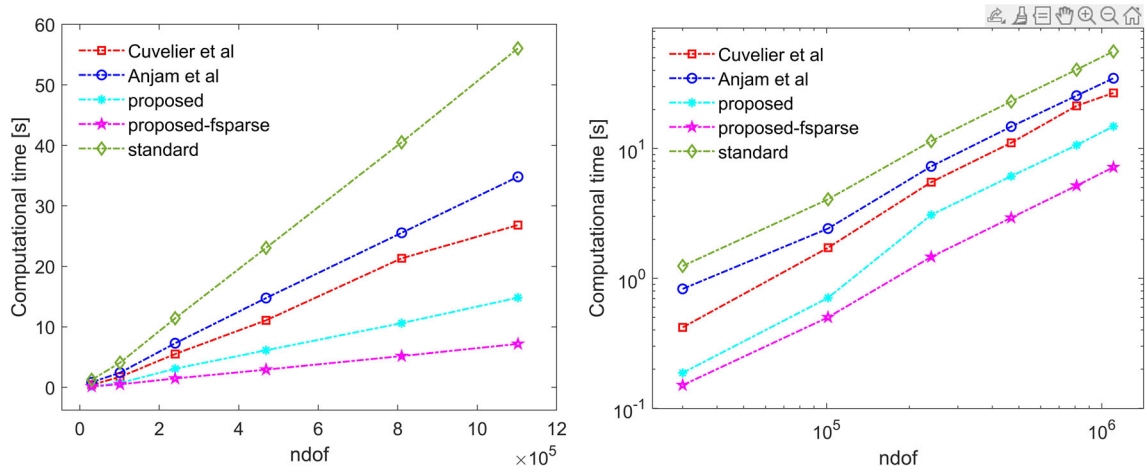


FIG. 13. Visualizaton of computational time for a set of 3D meshes with P_1 tetrahedral elements: **a** runtime versus the number of degrees of freedom; **b** runtime versus $ndof$ in log-log scale

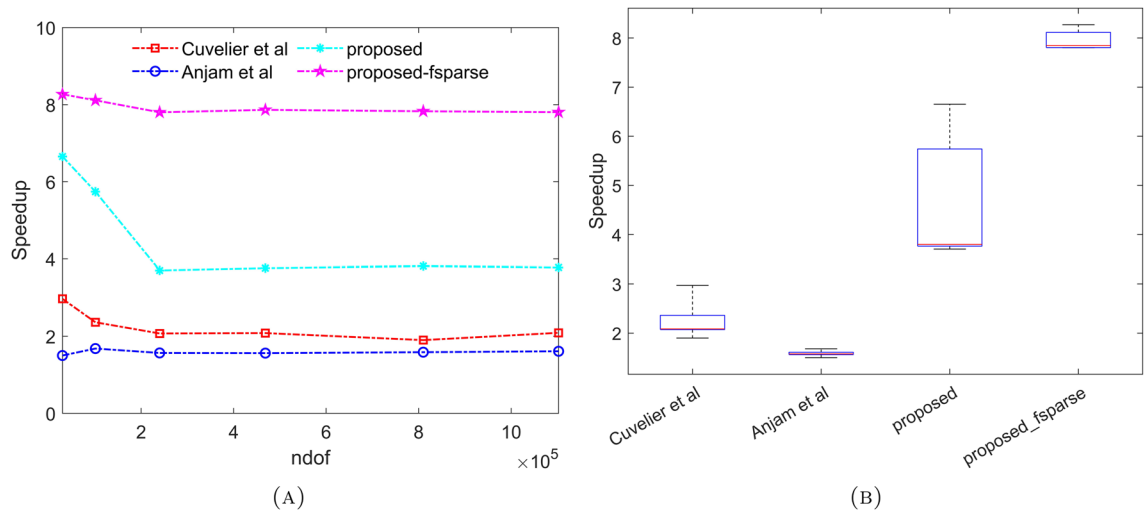


FIG. 14. Visualization of the computational speedup with respect to the standard algorithm for a set of 3D meshes with P_1 tetrahedral elements: **a** speedup versus the number of degrees of freedom; **b** boxplot of the speedup of the algorithms

or Q4) was created in ABAQUS and exported in MATLAB as topological inputs. The properties of the created meshes are summarized in Table 6.

Firstly, mesh 1 was selected for verification of the vectorized algorithm in Appendix 1 based on the results obtained in ABAQUS. Then, simulations were conducted on the same computer specified in Sect. 4.1. Figure 17 visualizes the comparison of the components of the solution vector $d = (\theta_x, \theta_y, w = U_z)$ associated with Mindlin plate theory formulation for the mesh 1.

For simplicity in this example, a shear correction factor of $k = k_x = k_y = \frac{5}{6}$ was considered for implementation in MATLAB. It can be seen in view of both results that the relative error is less that

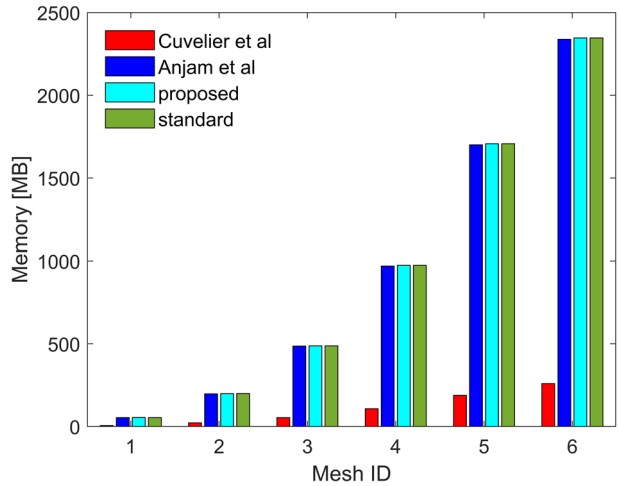


FIG. 15. Memory usage for computing $\mathcal{K}_g$ per algorithm with respect to mesh ID for the selected 3D plate problem

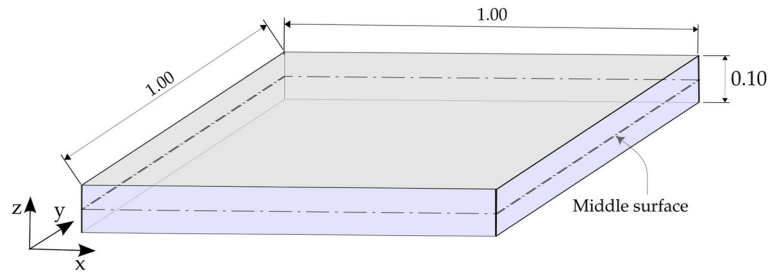


FIG. 16. Sketch of the plate

TABLE 6. *Set of 4-node quadrilateral element meshes*

Mesh ID	*nel*	*nn*
1	400	441
2	22 500	22 801
3	122 500	123 201
4	250 000	251 001
5	562 500	564 001
6	1 000 000	1 002 001

1%. Furthermore, the displacement $U_z = -1.503\ mm$ at the center of the plate is consistent with the one obtained by Ferreira in [43] for the same value of k.

Under the same boundary conditions, we have conducted a modal analysis of the plate (with mesh 2) based on the vectorized form of the Mindlin plate formulation outlined in Eq. (3.26). Table 7 provides a comparison of the six smallest natural frequencies obtained using ABAQUS and our MATLAB algorithm, respectively.

As of the result of the comparison of data in Table 7, it is obvious that the error is still less than 1%. In order to perform modal analysis, the parameter `AnalysisType` is set to `''modal''` using the command line

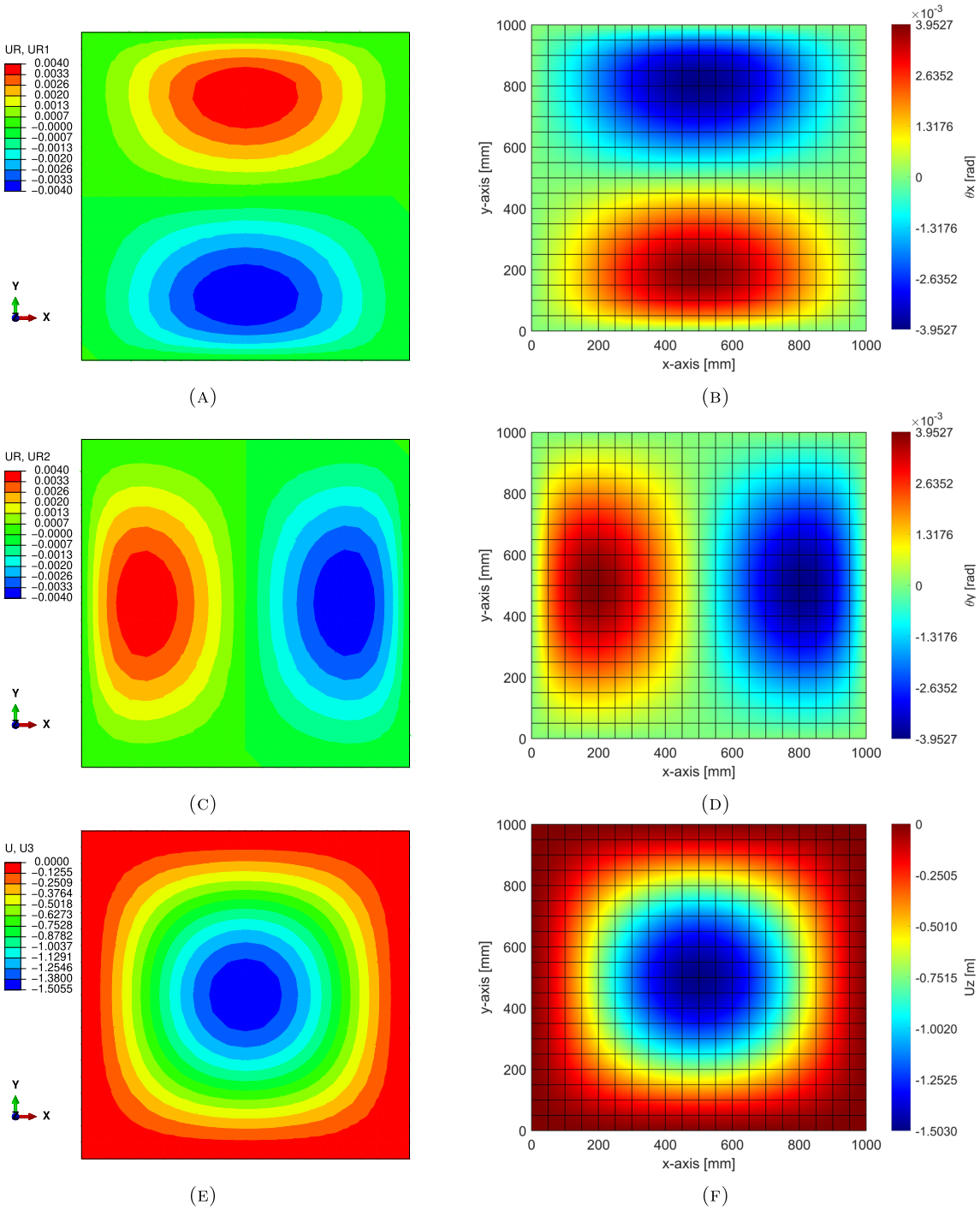
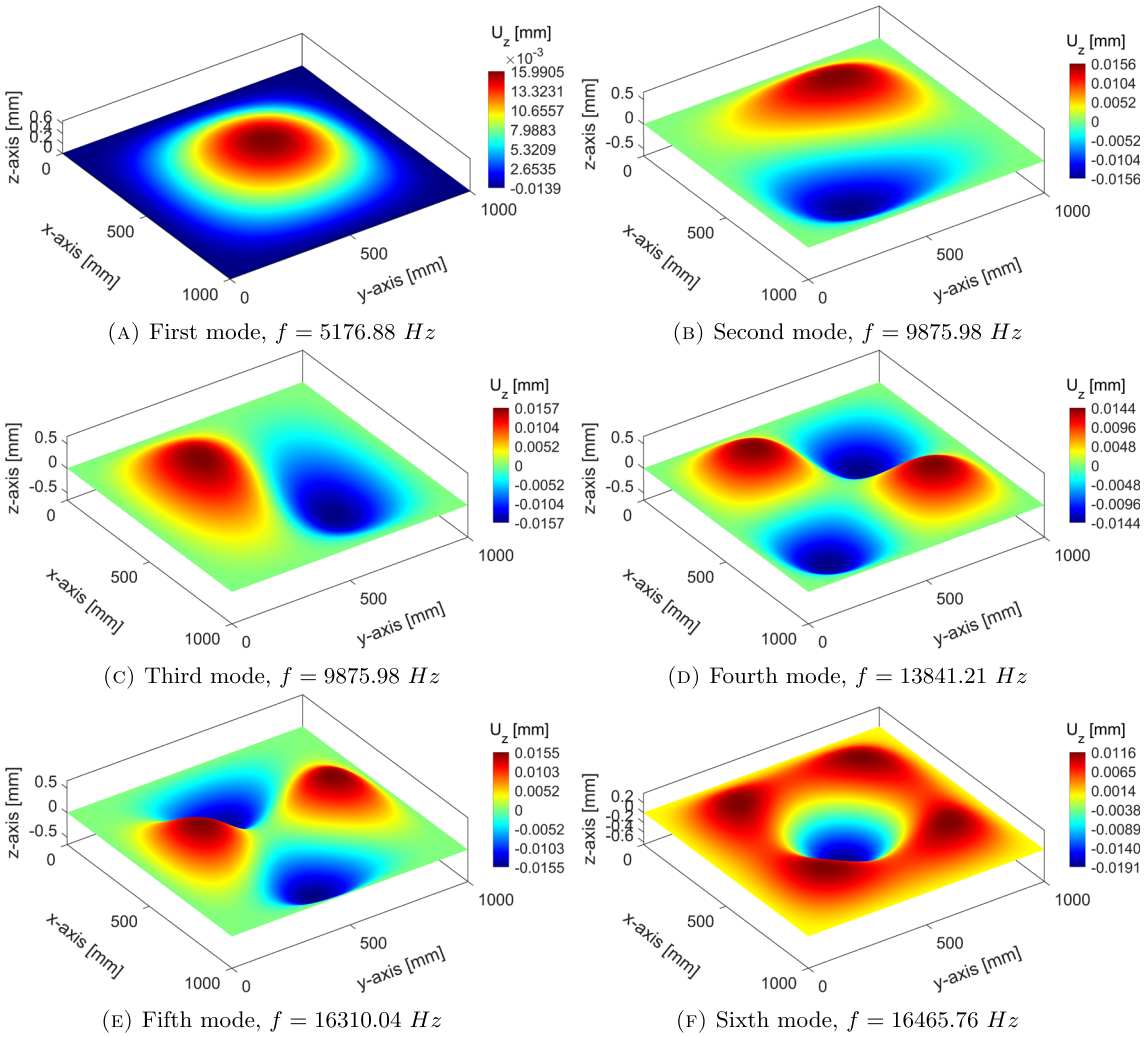


FIG. 17. Static result of the deformation of the plate: in ABAQUS/CAE (on the left-hand side) and in MATLAB with proposed algorithm (on the right-hand side)

TABLE 7. *Comparison of obtained frequencies*

	Mode 1	Mode 2	Mode 3	Mode 4	Mode 5	Mode 6
ABAQUS f_A $[Hz]$	5179.80	9875.80	9875.80	13841.00	16310.00	16466.00
Proposed f_{cal} $[Hz]$	5176.88	9875.98	9875.98	13841.21	16310.04	16465.76



(A) First mode, $f = 5176.88\ Hz$

(B) Second mode, $f = 9875.98\ Hz$

(C) Third mode, $f = 9875.98\ Hz$

(D) Fourth mode, $f = 13841.21\ Hz$

(E) Fifth mode, $f = 16310.04\ Hz$

(F) Sixth mode, $f = 16465.76\ Hz$

FIG. 18. Visualization of the six smallest mode shapes of the square plate

`FE_model.AnalysisType=''modal''` under the "Material parameters, analysis parameters and applied load" in the main file **MainProgram**. Focusing our attention on the six smallest modes, we obtained the mode shapes depicted in Fig. 18 and generated using **ModeShape.m** file created for this specific task.

As in the previous example, we have analyzed the performance of the static problem with the meshes shown in Table 6. Figure 19 depicts the comparison of the computational cost.

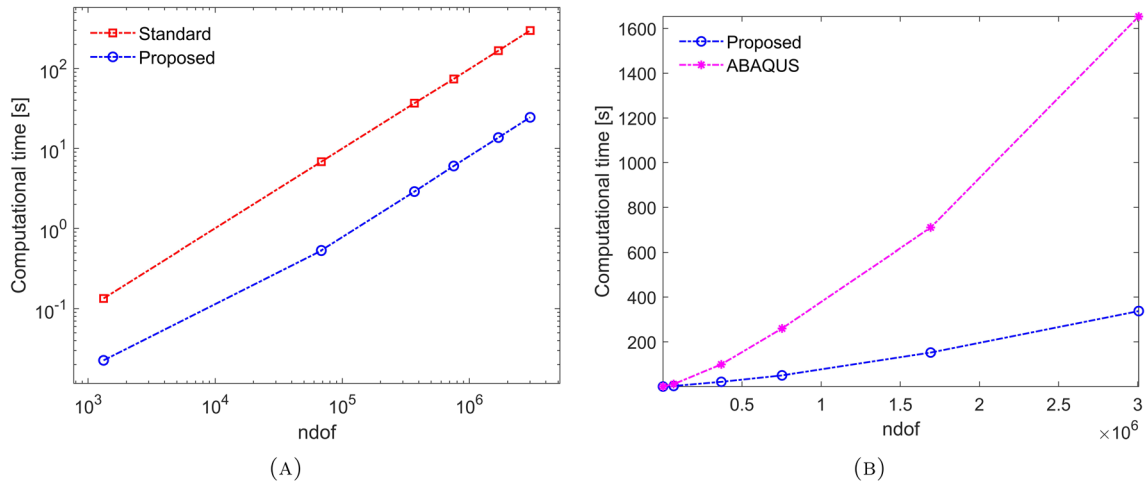


FIG. 19. Runtime performance analysis of the thick square plate: **a** CPU time for assembling K_g as per the *Standard* and *proposed* algorithm; **b** Total CPU time of the analysis (global matrix/force construction + solving the system of the equations)

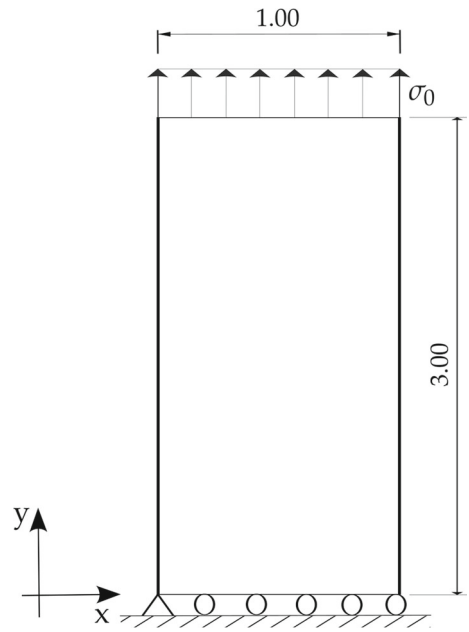
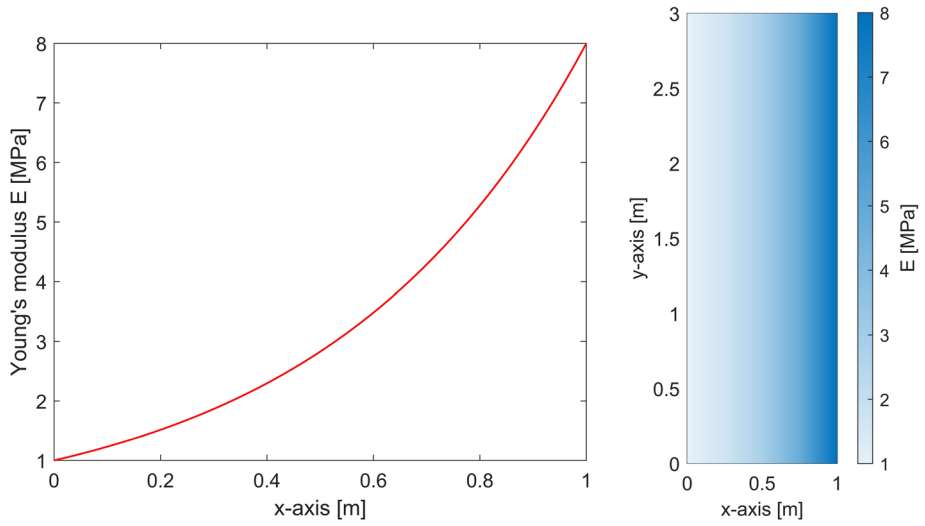


FIG. 20. Sketch of the FG membrane

In view of Fig. 19, it also turns out that in this example, matrix/force construction time is reduced by at least 10 times using the proposed method. Thus, the whole FEM process is accelerated by about 5 times compared to ABAQUS.



(A) (B)

FIG. 21. Exponential variation of Young's modulus E along x-axis: **a** Plotting of E versus x; **b** Distribution of the E within the functionally graded membrane

TABLE 8. *Set of 4-node quadrilateral element meshes*

Mesh ID	*nel*	*nn*
1	48	65
2	7 500	7 701
3	120 000	120 801
4	480 000	481 601
5	750 000	752 001
6	941 360	943 602

4.4. Example 4: functionally graded membrane

In this example, we assess the performance of the proposed method for solving a functionally graded membrane problem. For simplicity, we consider the example investigated by Martínez-Pañeda [44], with dimension sketched in Fig. 20.

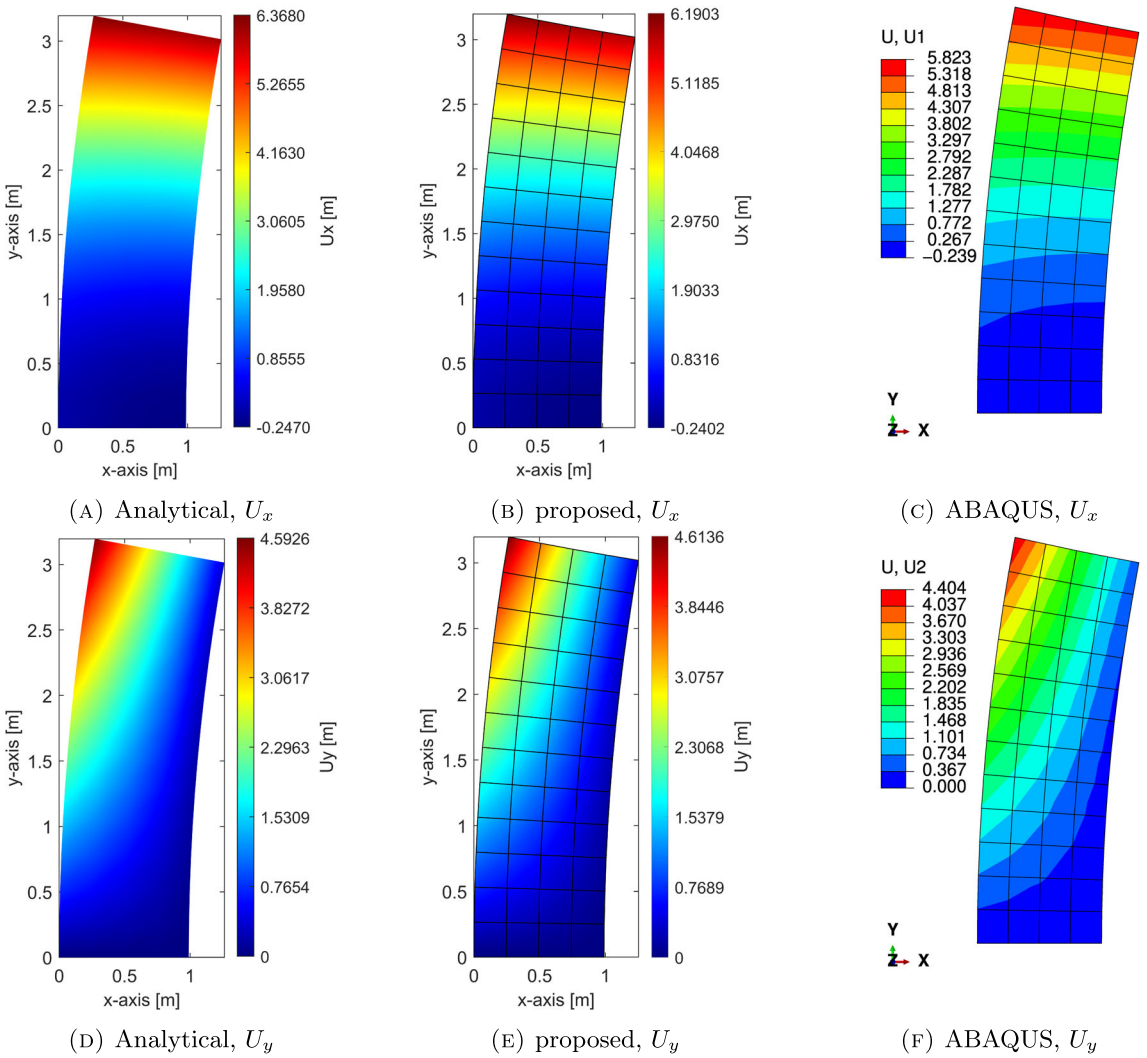
The material's Poisson ratio is set as a constant value $\nu = 0.30$. Meanwhile, Young's modulus E changes in the x-direction as per the function defined in Eq. (4.3).

$$E(x) = E_0 e^{\beta x} \tag{4.1}$$

where $\beta = ln(8)$ and $E_0 = 1$. Also, this membrane is 1 m thick and loaded with a uniform tensile stress of $\sigma_0 = 2$ MPa applied on the top surface of the membrane.

The analytical solution for this problem is given by

$$U_x(x,y) = -\nu\left(\frac{A}{2}x^2 + B\right) - \frac{A}{2}y^2;\ U_y(x,y) = (Ax + B)\,y;\ E_0 e^{\beta x} \tag{4.2}$$



(A) Analytical, U_x (B) proposed, U_x (C) ABAQUS, U_x

(D) Analytical, U_y (E) proposed, U_y (F) ABAQUS, U_y

FIG. 22. Visualization of FG membrane deformation for a scaling factor of 0.0428 according to the analytical equation (left column), the proposed algorithm (middle column) and ABAQUS (right column)

where A and B read as

$$A = \frac{\beta N}{2E_0}\frac{\beta^2 e^\beta - 2\beta e^\beta + \beta^2 + 2\beta}{\beta^2 e^\beta - e^{2\beta} + 2e^\beta - 1}; \quad B = \frac{\beta N}{2E_0}\frac{e^\beta\left[e^\beta\left(-\beta^2 + 3\beta - 4\right) + \beta^2 - 2\beta + 8\right] - \beta - 4}{\left(e^\beta - 1\right)\left(\beta^2 e^\beta - e^{2\beta} + 2e^\beta - 1\right)} \tag{4.3}$$

A set of 6 $P-1$ quadrangle meshes were created in ABAQUS as topological inputs for this analysis. Table 8 presents the size of these meshes.

In order to verify the accuracy of our algorithm, mesh 1 was selected and used for static analysis under the definition provided above. In ABAQUS, we utilized the so-called nodal/temperature-based definition of the young modulus following the x-axis. Using such an approach implies that the value of E at each Gauss integration point is calculated by numerical interpolation from the values of E at the element's nodes. Whereas in the exact method, E_g at each Gauss point is directly computed from the analytical expression of E using the spatial position obtained in Eq. (3.12).

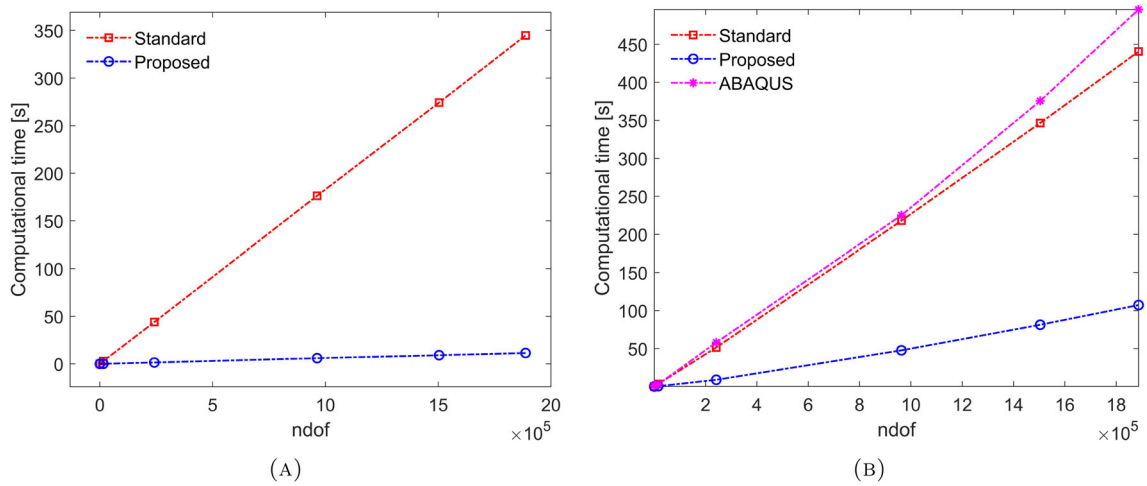


FIG. 23. Runtime performance analysis of the thick square plate: **a** CPU time for assembling K_g as per the *Standard* and *proposed* algorithm; **b** Total CPU time of the analysis (global matrix/force construction + solving the system of the equations)

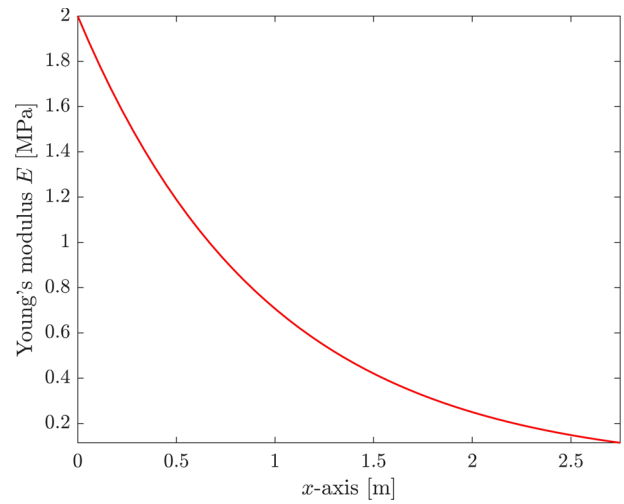


FIG. 24. Young's modulus E as a function x

Simulations were conducted on the same computer specified in Sect. 4.1. Figure 22 depicts the deformations (U_x and U_y) obtained with the analytical equation, proposed algorithm and node/temperature-based approximation method in ABAQUS.

The result of the performance analysis conducted on the set of meshes above is depicted in Fig. 23.

We observe that this example demonstrates a similar performance to the previous ones. The proposed algorithm continues to be at least ten times quicker, and its application for finite element problem analysis results in a fivefold decrease in CPU time compared with ABAQUS performance.

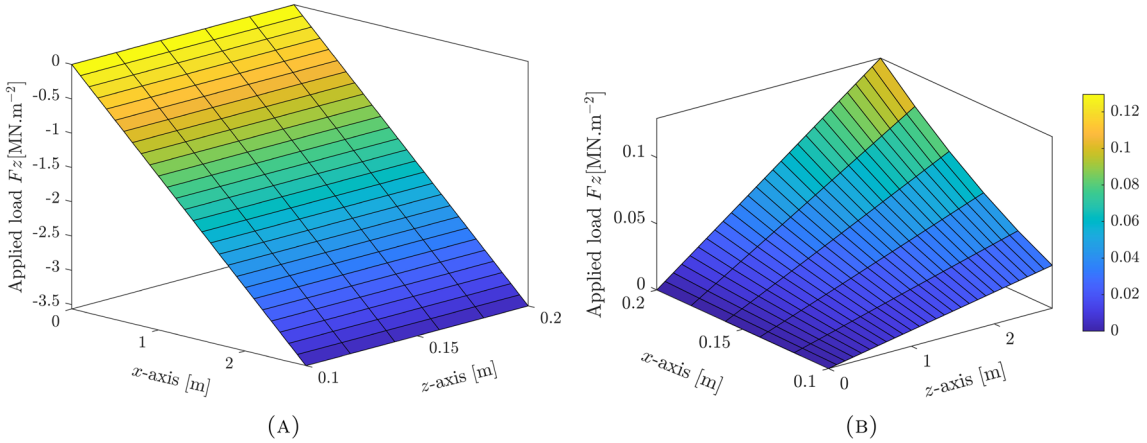


FIG. 25. Components of the applied pressure f_s: **a** representation of the function $f_y\,(x, y)$; **b** representation of the function $f_z\,(x, y)$

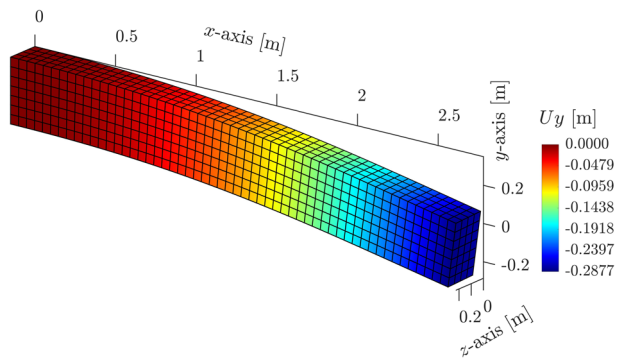


FIG. 26. Vertical displacement U_y of the beam

4.5. Example 5: functionally graded 3D cantilever beam

While the previous example was dedicated to evaluating the performance of the proposed algorithm on a 2D functionally graded membrane, this example focuses on the behavior of the algorithm in the case of a 3D functionally graded problem. The mechanical properties of the material are defined with the Poisson ratio ν equal to 0.3 and an exponentially decreasing Young's modulus E with respect to x-direction as illustrated in Figure[dddddd] and given by

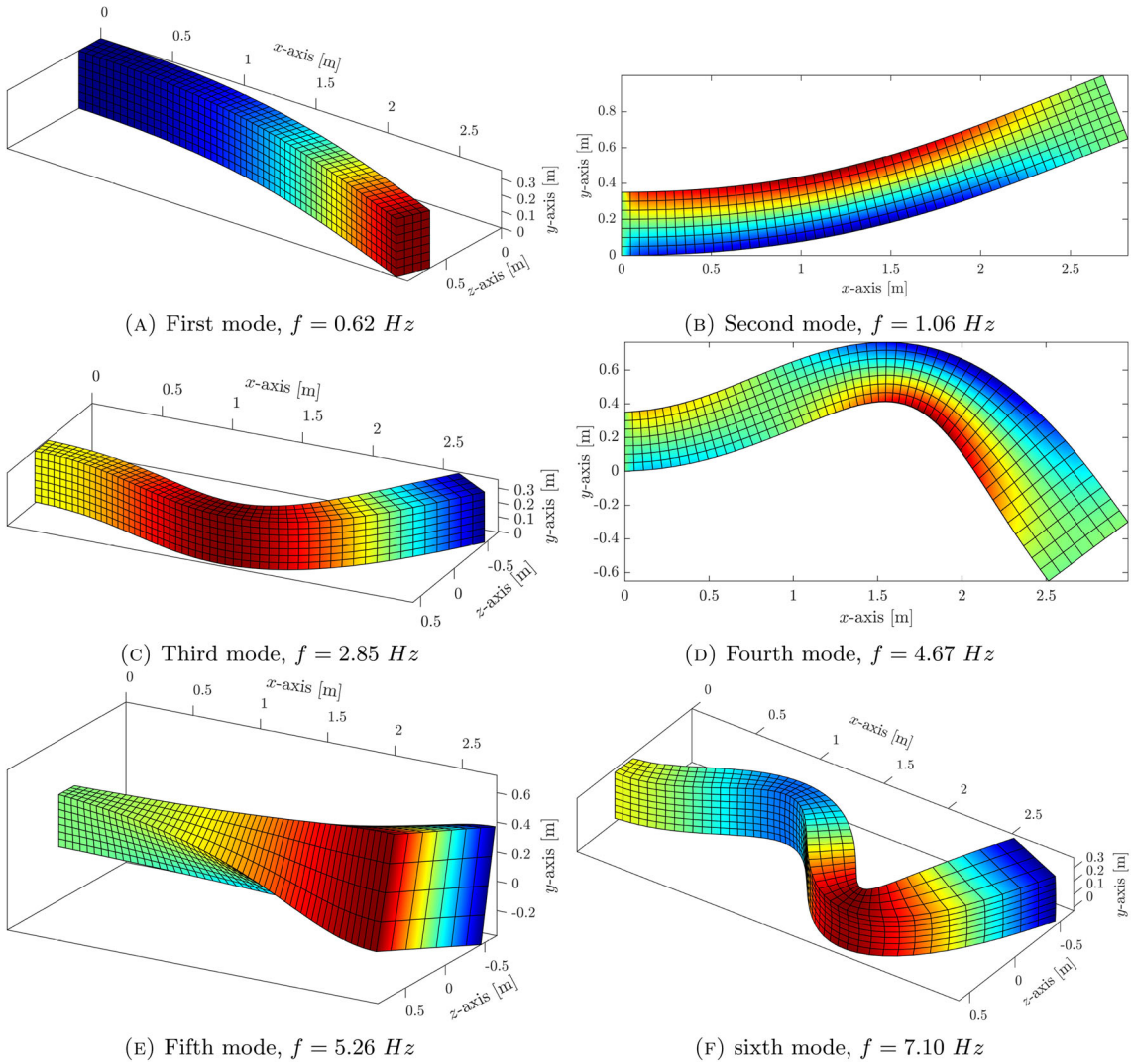
$$E\,(x) = E_0 e^{-\beta x} \tag{4.4}$$

where $\beta = \dfrac{1}{2} ln\,(8)$ and $E_0 = 20000$ MPa. The density of the beam is $\rho = 500$ kg.m^{-3}.

This functionally graded beam is non-symmetrically loaded with a complex pressure $f_s = (0, f_y, f_z)$ with components $f_y = -1.32x$ and $f_z = 1.2xz^2$ depicted in Fig. 25.

Simulations were conducted on the same computer specified in Sect. 4.1. After the analysis, we obtained the deformation depicted in Fig. 26.

We have also performed the modal analysis of the beam and obtained mode shapes associated with the six smallest natural frequencies of the beam as shown in Fig. 27.



(A) First mode, $f = 0.62\ Hz$

(B) Second mode, $f = 1.06\ Hz$

(C) Third mode, $f = 2.85\ Hz$

(D) Fourth mode, $f = 4.67\ Hz$

(E) Fifth mode, $f = 5.26\ Hz$

(F) sixth mode, $f = 7.10\ Hz$

FIG. 27. Visualization of the six smallest mode shapes of the square plate

Finally, we focused our attention on the CPU time (see Fig. 28) for performing the static and modal analysis

It should be noted that in the proposed function in the appendix, only the stiffness matrix is computed in the case of static analysis. In modal analysis, both the stiffness matrix and mass matrix are calculated. We can see here that combining both enables us to reduce the computational cost, though it might not be a good idea when the available memory is a limiting factor.

5. Conclusion

We have proposed in this work, an efficient MATLAB-based vectorized algorithm for computing global matrix/force deriving from the discretized weak form of boundary value problems in linear elasticity.

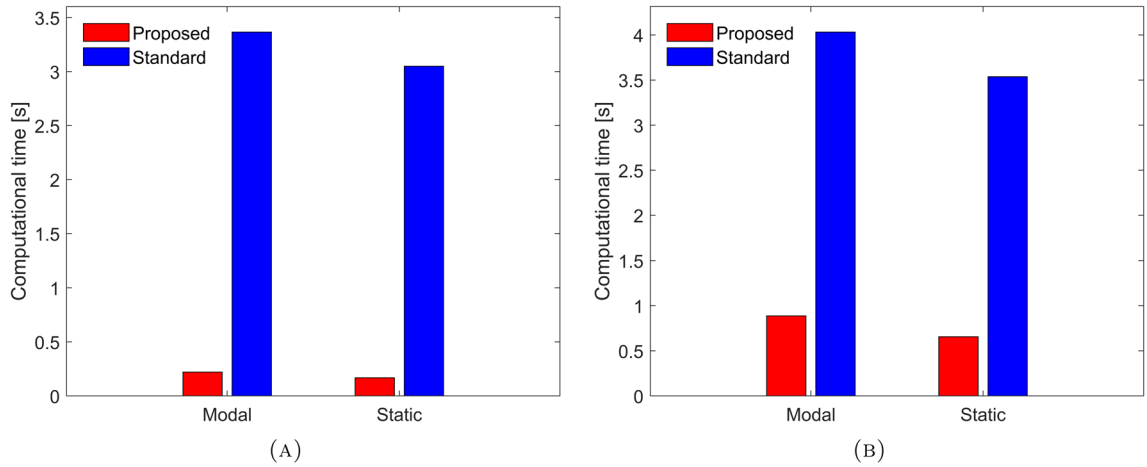


FIG. 28. Performance analysis for static and modal analysis; **a** CPU time taken to compute the matrix; **b** CPU time taken to compute the matrix and solve the system of equations

This scheme introduces a set of new features, which are detailed in section 3. By leveraging the power of vectorization and other optimizations, the algorithm can efficiently handle large-scale problems and deliver accurate results as verified in the numerical experiments in sections 4.1, 4.3, 4.4 and 4.5. Additionally, the performance analysis has revealed that the proposed method is at least ten times faster than the standard approach, and its utilization leads to a decrease of the runtime by at least five times compared to ABAQUS. This is because the approach presented builds on existing vectorized methods in the literature and improves on them in several important ways, including the use of advanced programming techniques, the integration of the latest MATLAB built-in functions to improve computational performance, generalization to meshes with elements of any order and type, and full vectorization of the construction of row indices, column indices and matrix/vector components. Moreover, an extension of the vectorized algorithm to special problems, such as Mindlin plate theory and functionally graded materials, was derived. Alongside its improved performance, it has been shown that this algorithm requires almost the same amount of memory as the standard algorithm, as shown by the performance analysis carried out in sections 4.2.1 and 4.2.2. Even though the proposed scheme requires more memory than that of Cuvelier et al. [15], a reference in the literature, it is at least 1.5x (with `sparse`) and 2.5x (with `fsparse`) faster than the latter, making it a game-changer in a massive simulation.

Overall, this work represents a significant contribution to the field of linear computational mechanics and has important implications for the design of structures and materials in various engineering and scientific applications.

Using the findings of this work, the following aspects will be investigated in forthcoming research:

- Enhancement of memory usage;
- Extension to nonlinear elasticity, J2 plasticity and contact problems;
- Implementation on graphics processing units (GPU).
- Development of an all-in-one fully vectorized MATLAB package intended for practical teaching of FEM to undergraduate and graduate students.

```
>> FE_model

FE_model =

  struct with fields:

          E: 2.100000000000000e+11
         nu: 0.300000000000000
        rho: 7850
  thickness: 0.050000000000000
          d: 2
          m: 2
       type: "T3"
      Coord: [780×2 double]
    connect: [1386×3 int32]
        nel: 1386
         nn: 780
          n: 3
```

FIG. 29. Problem data structure with its fields: FE_model

Acknowledgements

The authors acknowledge the funding provided by the University of Pardubice (through the grant numbers SGS_2023 and SGS_2024) and the Cultural and Educational Grant Agency of the Slovak Republic (KEGA), project No. KEGA 003TnUAD-4/2022 "Simulation of basic and specific experiments of polymers and composites based on experimental data in order to create a virtual computational-experimental laboratory of mechanical tests."

Funding Open access funding provided by The Ministry of Education, Science, Research and Sport of the Slovak Republic in cooperation with Centre for Scientific and Technical Information of the Slovak Republic

Declarations
Conflict of interest: The authors declare no conflict of interest.

Appendix A. MATLAB functions

Before all, it is worth keeping in mind that the data structure containing all the information on the Finite element model is saved under the variable FE_model of type `struct` in MATLAB with the following fields, for instance.

A.A. The general vectorized algorithm

```
function [Kg,Mg]=PropGlobalStiffness(FE_model)
%% PREPARATION OF MODEL DATA
d=FE_model.d; % d= dimension of the problem
n=FE_model.n; % n= number of nodes per element
m=FE_model.m; % m= number of degrees of freedom per node
nel=FE_model.nel; % nel= total number of elements of the mesh
rho=FE_model.rho; % rho= density of the material
AnalysisType=FE_model.AnalysisType;
[Gq,weight,gPoint]=Gauss_quadrature(FE_model); % get  Gauss quadrature data
[NI,dN_dai]=Shapefunction(FE_model,Gq,gPoint); % get shape functions and their deriavtives
Ke=zeros(m*n,m*n,nel); % initialize of element stiffness matrix
if AnalysisType=="modal"
  Me=zeros(FE_model.m*FE_model.n,FE_model.m*FE_model.n,nel);% initialize of element mass matrix
end
Mg=[];
%% CALCULATION OF ELEMENT MATRICES
CoordI=@(x)[FE_model.Coord((FE_model.connect(x,:))',:)]';
Pts=reshape(CoordI(1:nel),m,n,[]);
C=matProp(FE_model);Cb=C;% compute material tensor
for gq=1:gPoint % loop over Gauss quadrature point
   Ng=NI(:,gq);% retrieval of shape function vector at the Gauss point
   J=pagemtimes(dN_dai(:,:,gq),'none',Pts,'transpose');
   dN_dX=pagemldivide(J,dN_dai(:,:,gq));
   [B,N]=getB(dN_dX,nel,d,m,n,Ng,AnalysisType);dN_dX=[];
   detJ=DetJacobean(J,nel,d); J=[];
   if FE_model.MatType=="FGM" % FGM: functionally graded material
      C=pagemtimes(FE_model.Ef(pagemtimes(Pts(1,:,:),Ng)),Cb);
   end
   if d==2 % for 2D problems
      Ke=Ke+pagemtimes(B,'transpose',pagemtimes(C,B).*detJ*weight(gq)*FE_model.thickness,'none');
      if AnalysisType=="modal"
        Me=Me+rho*pagemtimes(N,'transpose',pagemtimes(eye(2,2),N).*detJ*weight(gq)*FE_model.thickness,'none');
      end
   elseif d==3 % for 3D problems
      Ke=Ke+pagemtimes(B,'transpose',pagemtimes(C,B).*detJ*weight(gq),'none');
      if AnalysisType=="modal"
         Me=Me+rho*pagemtimes(N,'transpose',pagemtimes(eye(3,3),N).*detJ*weight(gq),'none');
      end
   end
   B=[];detJ=[];
```

```
end
N=[];Pts=[];
Matrix_dof=int32(zeros(nel,n*m)');% initialization of the degrees of
   freedom connectivity matrix
k=(1:n);
for a=1:m
 Matrix_dof(m*(k-1)+a,:)=(m*(FE_model.connect(1:nel,k)-1)+a)';
end
IndexJ=int32(repmat(Matrix_dof(:)',m*n,1));IndexI=int32(repmat(
   Matrix_dof,m*n,1));Matrix_dof=[];
%% ASSEMBLY OF THE GLOBAL MATRICES
 if AnalysisType=="modal"
     Mg=sparse(IndexI(:),IndexJ(:),reshape(Me,[],1));Me=deal([]); %
        Assembly of the global mass matrix
 end
Kg=sparse(IndexI(:),IndexJ(:),reshape(Ke,[],1));[IndexI,IndexJ,Ke]=deal
   ([]);
```

A.B. For Mindlin plate theory

```
function [Kg,Mg]=PropGlobalStiffnessPlate(FE_model)
%% PREPARATION OF MODEL DATA
d=FE_model.d; % d= dimension of the problem
n=FE_model.n; % n= number of nodes per element
m=FE_model.m; % m= number of degrees of freedom per node
nel=FE_model.nel; % nel= total number of elements of the mesh
rho=FE_model.rho; % rho= density of the material
h=FE_model.thickness;
[Gq,weight,gPoint]=Gauss_quadrature(FE_model); % get Gauss quadrature
   data
[NI,dN_dai]=Shapefunction(FE_model,Gq,gPoint); % get shape functions and
    their deriavtives
C=matProp(FE_model); % compute material tensor
detJ=zeros(1,1,nel); % initialize the determinant of the jocobean
Ke=zeros(FE_model.m*FE_model.n,FE_model.m*FE_model.n,nel); % initialize
   of element mass matrix
AnalysisType=FE_model.AnalysisType;
if FE_model.AnalysisType=="modal"
    Me=zeros(FE_model.m*FE_model.n,FE_model.m*FE_model.n,nel);%
       initialize of element mass matrix
end
Mg=[];
%% CALCULATION OF ELEMENT MATRICES
CoordI=@(x)[FE_model.Coord((FE_model.connect(x,:))',:)]';
Pts=reshape(CoordI(1:nel),m,n,[]);
% Bending component
for gq=1:gPoint
```

```
    Ng=NI(:,gq);% retrieval of shape function vector at the Gauss
        quadrature point
    J=pagemtimes(dN_dai(:,:,gq),'none',Pts(1:2,:,:),'transpose');
    dN_dX=pagemldivide(J,dN_dai(:,:,gq));
    comp="bending";[Bf,~,N]=getB_plate(dN_dX,Ng,nel,m,n,comp,AnalysisType
        );dN_dX=[];%dN_dai=[];
    detJ=DetJacobean(J,nel,d);J=[];
    if FE_model.AnalysisType=="modal"
      Idm=pagemtimes(ones(1,1,nel),[(1/12)*h^3 0 0;0 (1/12)*h^3 0;0 0 h])
          ;
      Me=Me+rho*pagemtimes(N,'transpose',pagemtimes(Idm,N).*detJ.*weight(
          gq),'none');
    end
    Ke=Ke+pagemtimes((1/12)*Bf,'transpose',pagemtimes(C,Bf).*detJ*weight(
        gq)*weight(gq)*h^3,'none');Bf=[];detJ=[];
end
Bf=[];detJ=[];
% shear component
[NI,dN_dai]=Shapefunction(FE_model,[0;0],1);Ng=NI(:,1);
J=pagemtimes(dN_dai(:,:,1),'none',Pts(1:2,:,:),'transpose');
dN_dX=pagemldivide(J,dN_dai(:,:,1));
comp="shear";[~,Bc,~]=getB_plate(dN_dX,Ng,nel,m,n,comp,AnalysisType);
dN_dX=[];dN_dai=[];
detJ=DetJacobean(J,nel,d); J=[]; G=0.5*FE_model.E/(1+FE_model.nu);
Id=pagemtimes(ones(1,1,nel),eye(2,2));weight=2;
Ke=Ke+pagemtimes((FE_model.k)*Bc,'transpose',pagemtimes(G*Id,Bc).*detJ*
    weight*weight*h,'none');
B=[];detJ=[];N=[];Pts=[];
%% CALCULATION OF ELEMENT MATRICE INDICES(INDEXI,INDEXJ)
Matrix_dof=int32(zeros(nel,n*m)');% initialization of the degrees of
    freedom connectivity matrix
k=(1:n);
for a=1:m
 Matrix_dof(m*(k-1)+a,:)=(m*(FE_model.connect(1:nel,k)-1)+a)';
end
IndexJ=int32(repmat(Matrix_dof(:)',m*n,1));IndexI=int32(repmat(
    Matrix_dof,m*n,1));Matrix_dof=[];
%% ASSEMBLY OF THE GLOBAL MATRICES
if FE_model.AnalysisType=="modal"
 Mg=sparse(IndexI(:),IndexJ(:),reshape(Me,[],1));Me=deal([]);
end
Kg=sparse(IndexI(:),IndexJ(:),reshape(Ke,[],1));[IndexI,IndexJ,Ke]=deal
    ([]);
```

References

[1] Hrennikoff, A.: Solution of problems of elasticity by the framework method. J. Appl. Mech. **4**, A169–A175 (1941). https://doi.org/10.1115/1.4009129

[2] Courant, R.: Variational methods for the solution of problems of equilibrium and vibrations. Bull. Am. Math. Soc. **49**(1), 1–23 (1943). https://doi.org/10.1090/S0002-9904-1943-07818-4
[3] Kuhn, R., Parallel, P.D.: Processing to 2020. Synth. Lect. Comput. Archit. **15**(4), 1–166 (1980). https://doi.org/10.1007/978-3-031-01768-1
[4] Liu, W.K., Li, S., Park, H.S.: Eighty years of the finite element method: birth, evolution, and future. Arch. Comput. Methods Eng. **29**(6), 4431–53 (2022). https://doi.org/10.1007/s11831-022-09740-9
[5] Babuška, I., Narasimhan, R.: The Babuška–Brezzi condition and the patch test: an example. Comput. Methods Appl. Mech. Eng. **140**(1–2), 183–199 (1997). https://doi.org/10.1016/S0045-7825(96)01058-4
[6] Williamson, F.: A historical note on the finite element method. Int. J. Num. Meth. Eng. **15**, 930–934 (1980)
[7] Mulakkal, M.C., Castillo, A.C., Taylor, A.C., Blackman, B.R., Balint, D.S., Pimenta, S., et al.: Advancing mechanical recycling of multilayer plastics through finite element modelling and environmental policy. Resour. Conserv. Recycl. **166**, 105371 (2021). https://doi.org/10.1016/j.resconrec.2020.105371
[8] Panchal, J.H., Kalidindi, S.R., McDowell, D.L.: Key computational modeling issues in integrated computational materials engineering. Comput. Des. **45**(1), 4–25 (2013). https://doi.org/10.1016/j.cad.2012.06.006
[9] Keyes, D.E., McInnes, L.C., Woodward, C., Gropp, W., Myra, E., Pernice, M., et al.: Multiphysics simulations: challenges and opportunities. Int. J. High Perform. Comput. Appl. **27**(1), 4–83 (2013). https://doi.org/10.1177/1094342012468181
[10] Dede, E.M., Nomura, T., Lee, J.: Multiphysics Simulation. Springer-Verlag, London (2014). https://doi.org/10.1007/978-1-4471-5640-6
[11] The MathWorks, Inc. MATLAB version: 9.13.0 (R2022b). https://www.mathworks.com [Accessed 10 Mar 2023]
[12] Koko, J.: Vectorized MATLAB codes for linear two-dimensional elasticity. Sci. Program. **15**, 157–172 (2007). https://doi.org/10.1155/2007/838942
[13] Dabrowski, M., Krotkiewski, M., Schmid, D.W.: MILAMIN: MATLAB-based finite element method solver for large problems. Geochem. Geophys. Geosyst. (2008). https://doi.org/10.1029/2007GC001719
[14] Calo, V.M., Collier, N.O., Pardo, D., Paszyński, M.R.: Computational complexity and memory usage for multi-frontal direct solvers used in p finite element analysis. Proc. Comput. Sci. **4**, 1854–61 (2016). https://doi.org/10.1016/j.procs.2011.04.201
[15] Cuvelier, F., Japhet, C., Scarella, G.: An efficient way to assemble finite element matrices in vector languages. BIT Numer. Math. **56**, 833–864 (2016)
[16] Khoroshev, A., Vasyukov, I., Pavlenko, A., Batishchev, D.: Reduction in the computational complexity of calculating losses on Eddy currents in a hydrogen fuel cell using the finite element analysis. Inventions **8**(1), 1–38 (2023). https://doi.org/10.3390/inventions8010038
[17] Sumets, P.: Computational framework for the finite element method in MATLAB®and python. CRC Press (2022). https://doi.org/10.1201/9781003265979
[18] Bracikowski, N., Hecquet, M., Brochet, P., Shirinskii, S.V.: Multiphysics modeling of a permanent magnet synchronous machine by using lumped models. IEEE Trans. Industr. Electron. **59**(6), 2426–37 (2011). https://doi.org/10.1109/TIE.2011.2169640
[19] Sun, T., Mitchell, L., Kulkarni, K., Klöckner, A., Ham, D.A., Kelly, P.H.: A study of vectorization for matrix-free finite element methods. Int. J. High Perform. Comput. Appl. **34**(6), 629–44 (2020). https://doi.org/10.1177/1094342020945005
[20] Cecka, C., Lew, A., Darve, E.: Introduction to assembly of finite element methods on graphics processors. IOP Conf. Ser. Mater. Sci. Eng. (2010). https://doi.org/10.1088/1757-899X/10/1/012009
[21] Marcinkowski, L., Valdman, J.: MATLAB Implementation of Element-Based Solvers. In: Lirkov, I., Margenov, S. (eds) Large-Scale Scientific Computing. Springer, Cham. (2019)
[22] Rahman, T., Valdman, J.: Fast MATLAB assembly of FEM matrices in 2D and 3D: nodal elements. Appl. Math. Comput. **219**(13), 7151–7159 (2013). https://doi.org/10.1016/j.amc.2011.08.043
[23] Fritz, J.: Finite Element Methods and Vectorized Procedures in MATLAB, pp. 1-25 (2016)
[24] Chen, L.: Programming of Finite Element Methods in matlab. arXiv preprint arXiv:1804.05156. (2018) [Online]. Available: https://doi.org/10.48550/arXiv.1804.05156
[25] Lin, Q., Yan, N., Zhou, A.: A sparse finite element method with high accuracy Part I: Part I. Numer. Math. **88**, 731–742 (2001). https://doi.org/10.1007/s002110000244
[26] Bunch, JR., Rose, DJ.: Sparse Matrix Computations. Academic Press, 1-452. (1976) https://doi.org/10.1016/C2013-0-10439-4
[27] Zlotnik, S., Diez, P.: Assembling sparse matrices in MATLAB. Int. J. Numer. Methods Biomed. Eng. **26**(6), 760–9 (2010). https://doi.org/10.1002/cnm.1174
[28] Ljungkvist, K.: Techniques for Finite Element Methods on Modern Processors. Licentiate theses, Uppsala University, Dep. Inf. Technol. p.1-86. (2015) https://www.it.uu.se/research/publications/lic/2015-001/ [Accessed 04 Apr 2023]
[29] Anjam, I., Valdman, J.: Fast MATLAB assembly of FEM matrices in 2D and 3D: edge elements. Appl. Math. Comput. **267**, 252–63 (2015). https://doi.org/10.1016/j.amc.2015.03.105

[30] Zienkiewicz, OC., Taylor, RL., Zhu, JZ.: The finite element method: its basis and fundamentals. Elsevier, 7th ed., p1-714. (2013) https://doi.org/10.1016/C2009-0-24909-9
[31] Zienkiewicz, OC., Taylor, RL.: The Finite Element Method for Solid and Structural Mechanics. Elsevier, 6th ed, pp. 1-735 (2005)
[32] Wunderlich, W., Pilkey, W.D.: Mechanics of structures: variational and computational methods. CRC Press (2002). https://doi.org/10.1201/9781420041835
[33] Qin, Q.H., Wang, H.: Matlab and C programming for Trefftz finite element methods. CRC Press (2008). https://doi.org/10.1201/9781420072761
[34] Alberty, J., Carstensen, C., Funken, S.A., Klose, R.: MATLAB implementation of the finite element method in elasticity. Computing **69**, 239–63 (2002). https://doi.org/10.1007/s00607-002-1459-8
[35] Ferreira, A.J.: MATLAB Codes for Finite Element Analysis. Springer, Netherlands (2009)
[36] Karypis, G., Kumar, V.: A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices. University of Minnesota, Department of Computer Science and Engineering, Army HPC Research Center, Minneapolis (1998), https://hdl.handle.net/11299/215346 [Accessed 25 Mar 2023]
[37] Farhat, C., Pierson, K., Lesoinne, M.: The second generation FETI methods and their application to the parallel solution of large-scale linear and geometrically non-linear structural analysis problems. Comput. Methods Appl. Mech. Eng. **184**(2–4), 333–374 (2000). https://doi.org/10.1016/S0045-7825(99)00234-0
[38] Abaqus/CAE User's Guide-Abaqus: DS SIMULIA Corp. Johnston, USA (2020)
[39] Valdman, J.: Fast FEM assembly: edge elements, MATLAB Central File Exchange. Retrieved Oct 01, (2023). (https://www.mathworks.com/matlabcentral/fileexchange/46635-fast-fem-assembly-edge-elements)
[40] Cuvelier, F.: mVecFEMP1. https://www.math.univ-paris13.fr/cuvelier/software/
[41] Engblom, S., Lukarski, D.: Fast MATLAB compatible sparse assembly on multicore computers. Parallel Comput. **56**, 1–17 (2016). https://doi.org/10.1016/j.parco.2016.04.001
[42] Engblom, S.: stenglib. https://github.com/stefanengblom/stenglib
[43] Ferreira António, J.M.: MATLAB Codes For Finite Element Analysis. Springer, Netherlands (2009)
[44] Martínez-Pañeda, E.: On the finite element implementation of functionally graded materials. Materials **12**(2), 287 (2019). https://doi.org/10.3390/ma12020287

Baurice Sylvain Sadjiep Tchuigwa, Jan Krmela and Petr Jilek
Faculty of Transport Engineering
University of Pardubice
Studentská 95
532 10 Pardubice
Czech Republic
e-mail: sylvainsadjiep@gmail.com ;
bauricesylvain.sadjieptchuigwa@student.upce.cz

Jan Krmela
e-mail: jan.krmela@upce.cz ;
jan.krmela@tnuni.sk

Petr Jilek
e-mail: petr.jilek@upce.cz

Jan Pokorny and Vladimíra Krmelová
Faculty of Industrial Technologies in Púchov
Alexander Dubček University of Trenčín
Ivana Krasku 491/30
02001 Púchov
Slovak Republic
e-mail: vladimira.krmelova@tnuni.sk

Jan Pokorny
e-mail: jan.pokorny@upce.cz

(Received: January 17, 2024; revised: June 19, 2024; accepted: July 12, 2024)