lajss.org© Latin American Journal of Solids and Structures

www.lajss.org

Julia Language Implementation of the Finite Element Method for Linear Instability of Plane Frames: An Efficient Alternative for Structural Analysis

Leon Vale Lobo[a]*, Edilson Morais Lima e Silva[a]

[a] Faculdade de Engenharia Civil, Universidade Federal do Pará, Belém, Brasil. Emails: leon.lobo@itec.ufpa.br, edilson_morais@ufpa.br

* Corresponding author

https://doi.org/10.1590/1679-78257958

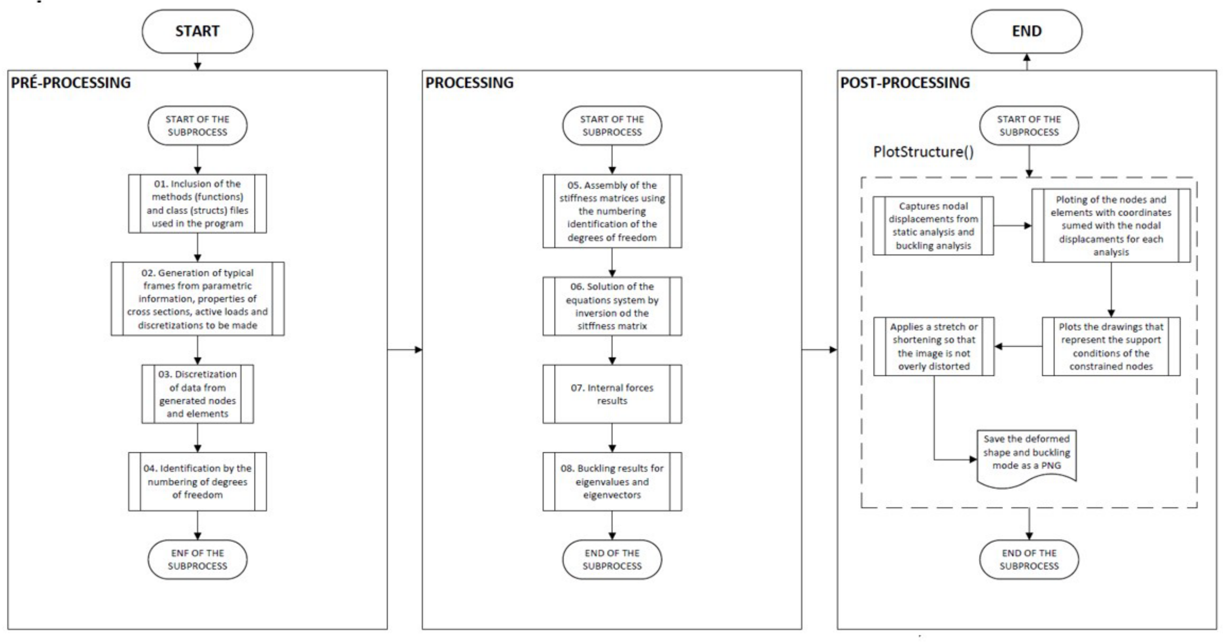
Abstract

This paper introduces the Julia programming language as a dynamic, cost-effective, and efficient framework for implementing structural analysis packages. To achieve this, the finite element method was implemented for plane frames addressing the elastic instability problem through the Finite Element Method (FEM). Julia is a language open source, multiplatform, high-level and high-performance for technical and scientific computing, its compiler allows you to achieve speeds comparable to languages such as C and FORTRAN, but with more productive development dynamics due to its programming flexibility. Benchmarks between Julia and MATLAB are employed to discuss the processing costs, the programming techniques and paradigms used for computational performance. The results demonstrate that Julia performed the same analysis as the language used for comparison in 88.40% of the time, in addition to the fact that in loops comparisons case it reached 41.7% of the time for iteration, confirming its significant potential as a development tool of computational packages for structural analysis and scientific computing in general.

Keywords

Computational Analysis, Buckling, Finite Elements, Frame Structures, Julia Language.

Graphical Abstract


START
PRÉ-PROCESSING
START OF THE SUBPROCESS
01. Inclusion of the methods (functions) and class (structs) files used in the program
02. Generation of typical frames from parametric information, properties of cross sections, active loads and discretizations to be made
03. Discretization of data from generated nodes and elements
04. Identification by the numbering of degrees of freedom
ENF OF THE SUBPROCESS
PROCESSING
START OF THE SUBPROCESS
05. Assembly of the stiffness matrices using the numbering identification of the degrees of freedom
06. Solution of the equations system by inversion od the sitffness matrix
07. Internal forces results
08. Buckling results for eigenvalues and eigenvectors
END OF THE SUBPROCESS
POST-PROCESSING
START OF THE SUBPROCESS
PlotStructure()
Captures nodal displacements from static analysis and buckling analysis
Ploting of the nodes and elements with coordinates sumed with the nodal displacements for each analysis
Plots the drawings that represent the support conditions of the constrained nodes
Applies a stretch or shortening so that the image is not overly distorted
Save the deformed shape and buckling mode as a PNG
END OF THE SUBPROCESS
END

Received: December 01, 2023. In revised form: January 24, 2024. Accepted: March 15, 2024. Available online: March 19, 2024.
https://doi.org/10.1590/1679-78257958

1 INTRODUCTION

In structural engineering, technical-scientific methods prioritize computational skills that enhance speed, performance, and accuracy in calculations, given that most problems are based on partial differential equations (Verdugo and Badia, 2021).

In scientific computing, the use of high-performance vectorized languages is common, such as MATLAB, Python, C++, R, and Octave (from de Oliveira Garcia and Paraguaia Silveira, 2017). The interest in these languages stems from the development of a clean, compact, easily implementable, and understandable code. This is crucial for teaching and fast prototyping in both research and industry. However, a point of concern is to ensure that this compact code is also fast and efficient for practical applications (Cuvelier, Japhet and Scarella, 2016).

Several studies, such as Dabrowski, Krotkiewski and Schmid (2008), present techniques to enhance the performance of codes written in MATLAB or Python by efficiently allocating properties of sparse matrices. However, even when optimized, these codes fall short of the performance achieved by compiled languages like Fortran and C (Bird, Coombs and Giani, 2017). On the other hand, well-established open-source programs known for their efficiency and robustness in structural analyses, such as OpenSees, face challenges in terms of interaction with the TCL language. As an alternative, a Python library is proposed to provide users with an additional language option (Zhu, McKenna and Scott, 2018).

In this context, recent studies by Xiao et al. (2021) indicate that Julia is challenging the 'natural' notions that high-level language programs must be slow or that machine performance requires sacrificing human convenience. Julia achieves this by incorporating computer science techniques such as multiparadigm programming, multiple dispatch, dynamic typing, and the use of package libraries in C and Python. It introduces a way to structure programs that is both easy and optimized, promoting recursion. This positions Julia as a promising language that addresses the balance between productivity and efficiency, a goal that older languages like FORTRAN attempted to achieve (Bezanson et al., 2012).

Julia language emerges as a valid alternative due to its characteristics tailored for technical-scientific practices: high performance, multiple dispatch, dynamic typing, and high-level syntax (Xiao et al., 2021). These features set it apart from current languages available for academic and engineering functions, promoting a shift in programming paradigms and renewing the architecture of code composition for problem-solving through computational analyses.

In relation to numerical methods applied to computational mechanics, there are already available packages for mesh methods (primarily based on FEM) that solve large systems of linear equations using various matrix operation methods, with efficient memory allocations and processing time. These packages intelligently employ sparse matrices for loops instead of vectorization, achieving speeds up to twice as fast in the processing of large matrices and vectors (Xiao et al., 2021).

Analyses conducted with recently developed FEM modules (Aho, Vuotikka and Frondelius, 2019) such as FEniCS and FreeFEM, performed in both static and dynamic analyses (Rapo, Aho and Frondelius, 2017), demonstrate optimistic results regarding the language's future for addressing current challenges, such as element refinement, plotting strain and stress fields, and algebraic operations with complete, sparse, or structured matrices.

Studies employing more advanced finite element techniques, such as smoothed finite elements, demonstrate tangible results in the computational efficiency of Julia compared to commercial software. For instance, the JuSFEM framework (Huo, Mei and Xu, 2021), when compared to ABAQUS, exhibited reduced processing time in a structural analysis of a three-dimensional clamped and free beam modeled with 2,000,000 tetrahedral elements. In this simulation, ABAQUS incurred a processing time of 930 seconds, whereas JuSFEM required only 543 seconds. More recently, Julia's characteristics were leveraged in the development of a battery modeling framework. The study highlights the ease of code implementation and compares the computational cost with a framework written in Python. The Julia framework costs approximately 1/5 of the execution time of the equivalent Python code (Ai and Liu, 2023).

However, there are still few studies in the literature that develop a framework and benchmark in Julia for structural modeling, presenting its characteristics and programming paradigms. The potential of the language for the development of packages for structural analysis through scientific computing is evident, as seen in the works of Antoshkin, Cherednichenko and Savelyeva (2021). Nevertheless, there are still limited studies addressing structural analysis in the Julia language.

The construction of the program is fully discussed in section 2.3, with the routine writing paradigm being emphasized. For examples of structuring the functionality of finite element codes for flat trusses, works such as (Zuo, Bai and Cheng, 2014) and (Zuo, Huang and Cheng, 2017) can demonstrate forms of traditional computational implementation in FEM. The present work chose to show the algorithms in tables so that the line-by-line explanation could be maintained.

In this context, this study utilizes the Julia language to implement linear buckling in plane systems, employing the eigenvalue and eigenvector solution through the Finite Element Methods. The programming types and Julia paradigms are systematically presented to enable the compiler to work with maximum computational efficiency. Benchmarks conducted on the developed framework for plane frames demonstrated accuracy in operations and computational performance superior to one of the world's leading scientific programming languages, MATLAB.

2 METHODOLOGY

2.1 Formulation of Linear Buckling using FEM

In this study, the implemented finite element is a plane frame bar based on the Euler-Bernoulli beam theory. The formulation of the plane frame element uses the principle of virtual work, interpolating displacements to obtain the element's stiffness matrix. Equation (1) the stiffness matrix of a finite element by the integral in the volume V of this same element (Bathe, 2014):

$$[k] = \int_V [B]^T [C][B] dV \tag{1}$$

where, $[k]$ is the stiffness matrix of a finite element; $[B]$ is the matrix of derivatives of displacement interpolation functions; $[C]$ is the elasticity matrix of the element.

Buckling is a stability loss phenomenon in slender elements due to the compression regime of structural elements (Hibbeler, 2010). It occurs in structural members that convert the energy from axial strain into flexural strain energy without any change in the applied external loading.

The critical condition for buckling occurs when the loss of axial strain energy is numerically equal to the gain in flexural strain energy, caused by a slight change in the strain state of a member. However, when studied under the linear elastic regime, the member remains perfectly straight, without eccentricity or geometric imperfection, until it reaches a critical load that induces lateral deflection.

Linear buckling is, by definition, the appearance of a bifurcation point in the equilibrium path, from which two configurations are possible: an undeformed (unstable) and a deformed (stable) one. Figure 1 illustrates a straight element experiencing stability loss and the possible equilibrium paths, respectively.

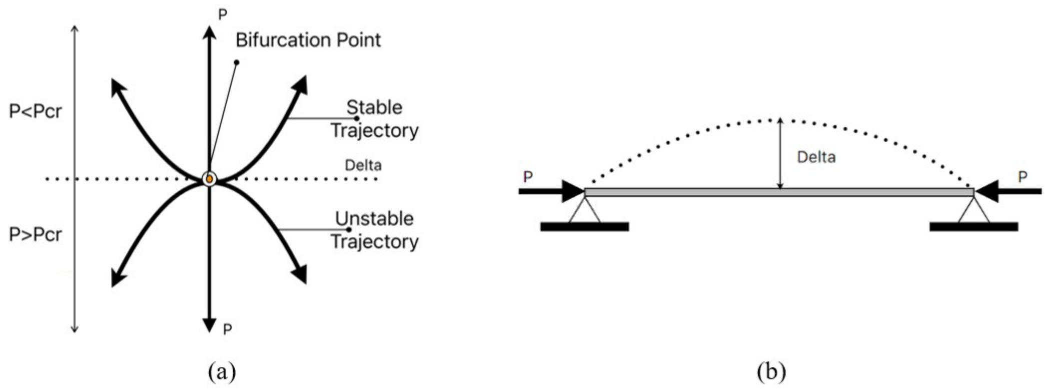


Figure 1 The critical load will cause a sudden change in the geometry of the member. (a) Displacement trajectory due to the load. (b) Bar under compressive load.

According to Cook et al., (2001), as shown in Equation (2) for a frame element in the loading configuration (Figure 1), the internal strain energy U of the system can be described as the sum of contributions from axial strain energy, the $P\Delta$ product effect, and flexural strain, this results in Equation (3).

$$U = \int_V \frac{1}{2} E\varepsilon_x^2 dV = \int_0^L \int_A \frac{1}{2} E\varepsilon_x^2 dA dx \tag{2}$$

$$U = \int_0^L \frac{EA}{2} u_{,x}^2 dx + \int_0^L \frac{P}{2} w_{,x}^2 dx + \int_0^L \frac{EI}{2} w_{,xx}^2 dx \tag{3}$$

Where L is the length of the frame element, E is the modulus of elasticity of the material, ε_x is the specific deformation in the axial direction, A is the cross-sectional area of the bar, I the moment of inertia of the cross-section of the bar, u is the displacement in the axial direction of the bar, w the curvature of the element with their respective derivatives and P the load applied in the axial direction of the element.

Using the FEM formulation, the derivatives of the matrices of functions describing the displacement fields at internal positions of the element (u, $w_{,xx}$ and $w_{,x}$) are replaced by the derivatives of the products between the interpolation functions and the nodal displacements of the elements, allowing to rewrite the product term $P\Delta$ in Equation (3) how is showed in the Equation (4).

$$\int_0^L \frac{P}{2} w_{,x}^2 dx = \frac{1}{2}\int_0^L w_{,x}^T P w_{,x} dx = \frac{1}{2}\int_0^L \{d\}[B]^T P[B]\{d\}^T dx \tag{4}$$

Therefore, is defined as the geometric stiffness the integral shown in the Equation (5).

$$[k_G] = \int_0^L [B]^T P[B] dx \tag{5}$$

It is observed, therefore, that axial forces change the flexural stiffness of a structural component. Buckling is the point at which compressive axial forces reduce the flexural stiffness to zero, causing a loss of stability due to the potential for any further strain. On the other hand, tensile axial forces increase flexural stiffness.

The effects caused by axial forces are accounted for in the matrix $[k_G]$, which is a function of the axial loads P. The matrix $[k_G]$ is defined through the geometry of the element, associated with its slenderness and internal forces, making it independent of elastic properties. Therefore, from Equation (3), it is evident that geometric stiffness complements the overall stiffness of the structure along with axial and flexural stiffness, together constituting the structure's stiffness matrix $[K]$. As $[K_G]$ is the geometric stiffness matrix of the structure, it is constructed by summing the components of the $[k_G]$ element matrix in a manner analogous to the structure's stiffness matrix $[K]$.

Modern structural analysis formulations, as can be seen in Equation (6), consider the contribution of all members of a structure to its stability and define buckling as the point reached when, using a buckling factor λ multiplied by the geometric stiffness $[k_G]$, a total stiffness of zero is obtained when summing the geometric component with the elastic stiffness $[k_E]$ (Alonso et al., 2015). This results in lateral deflection.

$$[k] = [k_E] + \lambda[k_G] = \{0\} \tag{6}$$

From this definition, a characteristic problem can be defined as in the Equation (7).

$$([K_E] + \lambda.[K_G]).\{d\} = \{0\} \tag{7}$$

Within engineering, buckling serves as a penalty to calculated strengths, as slender components will have limitations on application not necessarily associated with their failure (whether brittle or ductile) but rather with their stability condition. This condition is constrained by displacements, vibrations, and even collapse, limitations more commonly observed in high buildings.

In determining the load that will cause buckling using the FEM, there is an eigenvalue problem for which the solution involves an eigenvector and eigenvalue problem. In Equation (7), λ is a list of possible solutions as eigenvalues, where only the smallest value will be regarded as λ_{cr} and $\{d\}$ is the associated eigenvector defining the buckling mode. To calculate these, the value of $[k_G]$ must be known.

$$[k_G] = P.\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & \frac{36}{30.L} & \frac{1}{10} & 0 & \frac{-36}{30.L} & \frac{1}{10} \\ 0 & \frac{1}{10} & \frac{4.L}{30} & 0 & \frac{-1}{10} & \frac{-L}{30} \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & \frac{-36}{30.L} & \frac{1}{10} & 0 & \frac{36}{30.L} & \frac{-1}{10} \\ 0 & \frac{1}{10} & \frac{-L}{30} & 0 & \frac{-1}{10} & \frac{4.L}{30} \end{bmatrix} \tag{8}$$

For pinned bars, all stiffness associated with rotational degree of freedom should be zero. Applying the procedures with the transformation matrix $[R]$ (Equation (10)) to the global coordinate system and assembly through the structure's stiffness matrix with the kinematic incidence matrix $[H]$.

$$[k_{G_{global}}]=[R]^T.[k_{G_{local}}].[R] \tag{9}$$

$$[K_G]=[H]^T.[k_{G_{global}}].[H] \tag{10}$$

Solving the characteristic equation reveals how many times the structure's load needs to be multiplied for it to lose stability. This is the critical buckling factor (λ_{cr}).

$$([K_E]+\lambda_{cr}.[K_G]).\{d\}=0 \tag{11}$$

2.2 Julia Programming Language: A Compiled Dynamic Typing Language

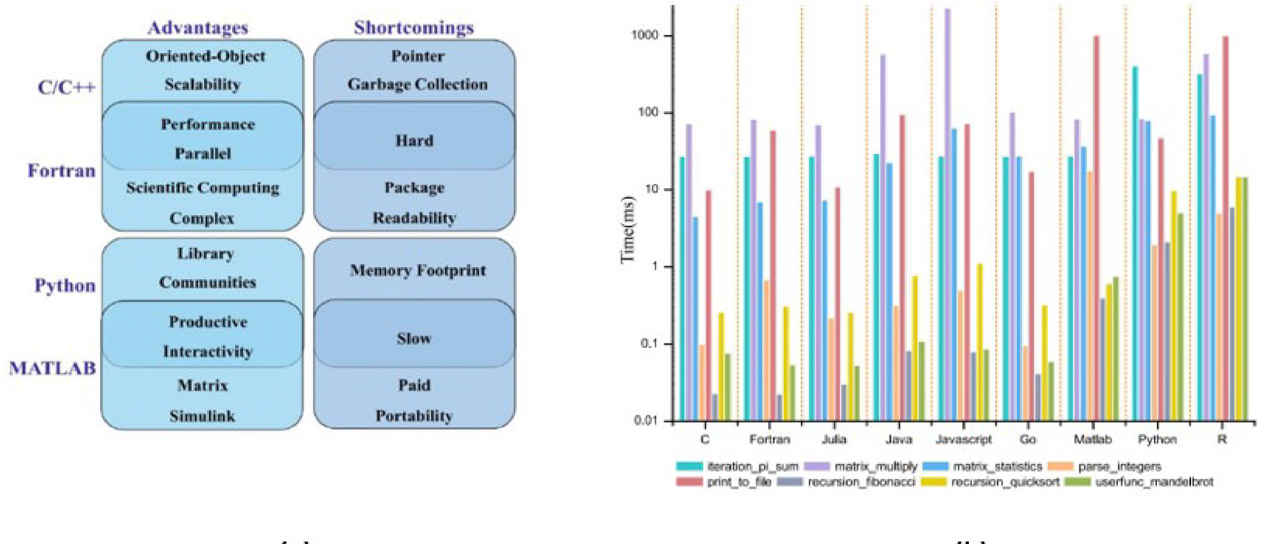
Julia is a high-level dynamic language developed to efficiently meet the requirements of numerical and scientific computing. A distinctive advantage of Julia over other high-level languages is the use of a Just-in-Time (JIT) compiler to generate binary code using the Low-Level Virtual Machine (Bezanson et al., 2012).

The Just-In-Time (JIT) compiler facilitates compilation simultaneously with code execution. This feature provides an advantage over languages that use an interpreter to execute their instructions (Bezanson et al., 2012). The JIT combines the speed of pre-compiled languages with the flexibility of interpreted languages thanks to the LLVM-based architecture.

The type inference used in LLVM allows, whenever possible, a specific type of scalar variable to be assigned within the scope of compilation for use in multiple dispatch (Bezanson et al., 2012). This enables Julia to gain efficiency by creating different methods for the same function depending on the type of data or structure that enters as an argument in the function/method. This characteristic explains Julia's equivalence when compared to languages such as C++, FORTRAN, MATLAB, and Python.

Among the languages compared in technical-scientific applications, the previous four fall into two broad groups: high-level languages and low-level languages. Languages with good syntax, easy to learn with steep learning curves, are called high-level because they have pleasant environments with functions and semantics closer to human language. Examples include MATLAB and Python, which are very different from machine languages, where fast processing is emphasized as the main advantage.

Medium-level languages like C and FORTRAN have static typing, defining variable types at compile time, which is beneficial for processing speed. However, when compared to MATLAB and Python, these languages lose in productivity and learning speed, posing a barrier for non-expert users in the field of computing (Coleman et al., 2021). Examples of advantages and disadvantages of these languages are shown in Figure 2.



(a) (b)

Figure 2 Comparisons of the Julia language. (a) Comparison of pros and cons of the 4 languages (Source: Xiao et al., 2021). (b) Julia micro-benchmarks using common code patterns (Source: https://julialang.org/benchmarks/).

2.3 Computational Implementation in Julia

Starting from a matrix formulation of the Finite Element Method, an algorithm was implemented for the problem of buckling in planes frame systems. Codes were created in different files that are called and executed by a main file called "00. Main.jl". The overall functioning of the code is presented in the flowchart shown in Figure 3. The functions are divided into three subprocesses:

- Pre-processing (formation of the data to be evaluated);
- Processing (subprocess where the most computationally costly calculations are performed);
- Post-processing (exporting the visual representation of the calculated results).

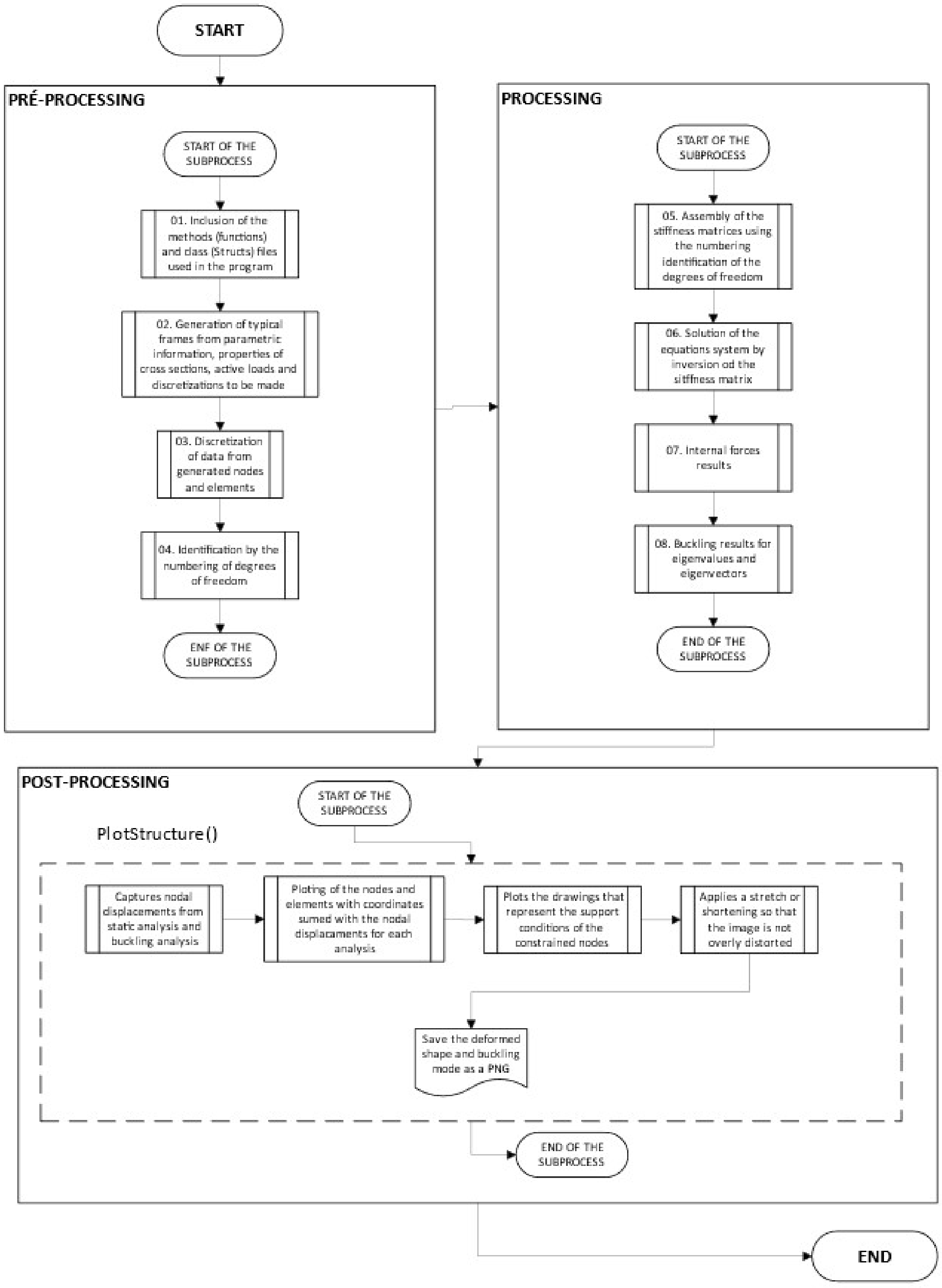

START
PRÉ-PROCESSING
START OF THE SUBPROCESS
01. Inclusion of the methods (functions) and class (Structs) files used in the program
02. Generation of typical frames from parametric information, properties of cross sections, active loads and discretizations to be made
03. Discretization of data from generated nodes and elements
04. Identification by the numbering of degrees of freedom
ENF OF THE SUBPROCESS
PROCESSING
START OF THE SUBPROCESS
05. Assembly of the stiffness matrices using the numbering identification of the degrees of freedom
06. Solution of the equations system by inversion od the sitffness matrix
07. Internal forces results
08. Buckling results for eigenvalues and eigenvectors
END OF THE SUBPROCESS
POST-PROCESSING
START OF THE SUBPROCESS
PlotStructure()
Captures nodal displacements from static analysis and buckling analysis
Ploting of the nodes and elements with coordinates sumed with the nodal displacements for each analysis
Plots the drawings that represent the support conditions of the constrained nodes
Applies a stretch or shortening so that the image is not overly distorted
Save the deformed shape and buckling mode as a PNG
END OF THE SUBPROCESS
END

Figure 3 Workflow of the buckling analysis of the program.

For the scope of what is discussed in this article, it focuses in detail on the way the code is written, as this is where the important nuances that enhance the computing efficiency of the alternative language presented are found. Algorithm 01 shows the main structure classes (structs) used as input for each method that composes the main algorithm, which performs linear buckling analysis using the Finite Element Method.

Algorithm 01 Class of objects used in Julia code

```
# Str Node
struct Node
    Coords:: Coordinates
    Constrs:: Constraints
    NLoads:: Nodal_Loads
    Displs:: Displacements
end
# Str Element
struct Element
    Connect:: Connectvt
    Section:: Section
    DLoads:: Distributed_Loads
    Discret:: Int64
end
# Str Structure Model
struct Structure_Model
    NodesData:: Matrix{Node}
    ElementsData:: Matrix{Element}
    NumbNodes:: Int64
    NumbElem:: Int64
    NumbDOF:: Int64
end
# Str Results
struct Results
    Id:: Identifiers
    MV:: Matrix_and_Vectors
    Ans:: Answers
    IF:: Solicitations
    BuckId:: Buckling_Id
    DynamId:: Dynamic_Id
end
# Str Analysis
struct Analysis
    SM:: Structure_Model
    Res:: Results
end
```

The function BuckMEF(), shown in Algorithm 02, generates parametric computational models of plane frames based on the number of horizontal divisions, vertical divisions, physical properties, geometric properties, boundary conditions, and loading modeling acting on the structure. With the computational model defined, the other functions within Algorithm 02 process the data for buckling analysis through the assembly of matrices, solving systems of equations, and presenting the results.

Algorithm 02 Analysis of stability in frames by Finite Element Method in Julia

```
01 : function BuckMEF(bw_b, h_b, bw_p, h_p, Nv)
02 :     #02. Automatic data generated for the template to analysis
03 :     FrameType = [5 3 5 Nv 2 3e7] #[beam_length pillar_length Number_of_Horizontal_Frames Number_of_Vertical_Frames constraints E]
04 :     GeoSec = [bw_b h_b bw_p h_p] #[bw_beam h_beam bw_pillar h_pillar]
05 :     Load_Disc = [10.0 -5.0 10 6] #[Px q discret_beam discret_pillar]
06 :     SM = GenerateFrames(FrameType,GeoSec,Load_Disc)
07 :     #03. Discretization of the tabulated structure with new nodes and elements
08 :     SM = Discretizer(SM)
09 :     #04. Numbering of degrees of freedom of the problem
10 :     Id = DOF(SM)
11 :     #05. Calculus of the stiffness matrices and equivalent nodal loads
12 :     (K,M,Peq) = (zeros(SM.NumbDOF,SM.NumbDOF),zeros(SM.NumbDOF,SM.NumbDOF),zeros(SM.NumbDOF))
13 :     MV = MatrixAssemling(SM,Id,K,M,Peq)
14 :     #06. Matricial solution of the linear system of equations
15 :     Ans = EqSolution(MV,Id)
16 :     #07. Results of nodal loads in local system of coordinates to assembly of diagrams
17 :     IF = InternalForces(MV,Ans)
18 :     #08. Solutions for the characteristic problem: λcr and the buckling mode of the structure
19 :     BuckId = CaracProblSol(Id,Ans,IF)
20 :     #09. Exportation of the deformed shape and buckling mode of the structure in PNG
21 :     PlotStructure(Analysis(SM,Res),100,4,0.20)
22 :     Res = Results(Id,MV,Ans,IF,BuckId) #concatenation of the results
23 :     return Analysis(SM,Res)
24 : end
```

Algorithms 03, 04 and 05 detail the Julia paradigms used in the implementation of the program for the elastic buckling analysis of plane frames. In Julia, for the developed program to be more performant, codes should be written within functions (*Performance Tips*, 2023). The way the Julia compiler works has the consequence that codes written within functions tend to run faster than codes outside. This characteristic results in data encapsulation since, to operate on variables, they must be passed to the function as arguments, avoiding direct operations on global variables. The efficient way Julia handles arguments or attributes is by using structures that are like components of object-oriented programming (OOP) classes. Therefore, analogously, functions that operate with structures in Julia are OOP methods.

These paradigms are demonstrated by the matrix assembly function and equivalent load vector in Algorithm 03. This function presents another important performance paradigm: the use of non-vectorized loops in a column-row order sequence (Canis et al., 2013). This characteristic differs from languages like MATLAB and Python, which use vectorized loops in a row-column sequence for greater efficiency.

The idea behind non-vectorized loops is to provide a simple and easy way to develop codes that are fast and efficient. The column-row sequential order is due to the way the compiler was developed to achieve computational efficiency in line with Julia's programming paradigms and concepts.

Loops are costly operations, and in Julia, loops work internally with local variables. Therefore, to avoid global scope variables, loops must operate within functions using their arguments.

Effective memory management leads to faster codes. Like any other programming language, better performance is achieved by pre-allocating variables, and with the Julia compiler, this is no different (Kemmer, Rjasanow and Hildebrandt, 2018). However, special attention should be given to declaring stable types in Julia, avoiding a variable changing types throughout the program, as for the Just-in-Time (JIT) compiler, this would incur additional costs for computation (Byrne, Wilcox and Churavy, 2019).

From line 05 to 09, the syntax for type declaration and memory allocation for vectors and matrices can be noted. This syntax uses the *undef* parameter, which generates a space in memory with low computational cost compared to initializing vectors and matrices with zero or one functions. However, special attention should be given to allocations with *undef*, as it should be used only for value replacement, avoiding algebraic operations. During initialization, random residues are generated, leading to loss of precision and instability in matrix operations.

From line 01, it is evident that the entire analysis is performed within a function. This choice is made (and replicated for each analysis process package) for optimal multiple dispatch utilization. In the multiple dispatch paradigm, the data types passed to the function define the sequence of methods that will be executed. This avoids the need to check the type of the variable being used and whether this type is processable for the operations programmed in this function, ensuring computational performance.

Another consideration for functions in Julia, such as "MatrixAssembling" in Algorithm 03, is data encapsulation. This occurs because the arguments passed to the function operate in a local scope, ensuring data privacy. This characteristic is also adopted for loops in Julia (Kemmer, Rjasanow, and Hildebrandt, 2018).

Another detail is the use of macros that optimize the code. For example, in Algorithm 03, the @SMatrix macro from the @StaticArrays library is applied. This macro informs the compiler that the allocated matrix size will not change throughout the loops, allowing the compiler to specialize the code for the matrix size to be used, storing the elements in the CPU register and avoiding unnecessary allocations (Huo et al., 2020).

Algorithm 03 Local and global stiffness calculation with the assembling using degrees of freedom

```
01 : function MatrixAssembling(SM:: Structure_Model, Id:: Identifiers, K:: Matrix{Float64}, M:: Matrix{Float64}, Peq:: Vector{Float64})
02 :     #Capture of the model information
03 :     NumbElem = SM.NumbElem
04 :     #Previous creation of matrices for stiffness matrices calculations
05 :     R = Array{Float64}(undef,(6,6,NumbElem))
06 :     kl = Array{Float64}(undef,(6,6,NumbElem))
07 :     pleq = Array{Float64}(undef,(6,1,NumbElem))
08 :     dof_elements = Matrix{Int16}(undef,NumbElem,6)
09 :     kgl = Array{Float64}(undef,(6,6,NumbElem))
10 :     #Sweep all the frames of structure for the [K] and [Peq] calculations
11 :     for Element in 1:NumbElem
12 :     Frame = SM.ElementsData[1,Element]
13 :     (L,cos,sen) = Orientation(Frame) #function 03.a.
14 :     (EA,EI) = SecProp(Frame) #function 03.b.
15 :         #Local stiffness matrix
16 :         kl[:,:,Element] = @SMatrix [
17 :             EA/L 0 0 -EA/L 0 0
18 :         0 12*EI/(L^3) 6*EI/(L^2) 0 -12*EI/(L^3) 6*EI/(L^2)
19 :         0 6*EI/(L^2) 4*EI/L 0 -6*EI/(L^2) 2*EI/L
20 :             -EA/L 0 0 EA/L 0 0
21 :         0 -12*EI/(L^3) -6*EI/(L^2) 0 12*EI/(L^3) -6*EI/(L^2)
22 :         0 6*EI/(L^2) 2*EI/L 0 -6*EI/(L^2) 4*EI/L
23 :             ]
24 :         #Rotation matrix
25 :     R[:,:,Element] = @SMatrix [
26 :             cos sen 0 0 0 0
27 :             -sen cos 0 0 0 0
28 :             0 0 1 0 0 0
29 :             0 0 0 cos sen 0
30 :             0 0 0 -sen cos 0
31 :             0 0 0 0 0 1
32 :             ]
33 :     #Global stiffness and mass matrix
34 :     kg = R[:,:,Element]'*kl[:,:,Element]*R[:,:,Element]
35 :     dof_elements[Element,:] = Int64.(DOFElement(Frame,SM,Id)) #function 03.c.
36 :     #Structure stiffness and mass matrix
37 :     K[dof_elements[Element,:],dof_elements[Element,:]] = K[dof_elements[Element,:],dof_elements[Element,:]] + kg
```

```
#Distributed loads on element
ql0 = [Frame.DLoads.Localqx;Frame.DLoads.Localqy]
qg = [Frame.DLoads.Globalqx;Frame.DLoads.Globalqy]
qp = [Frame.DLoads.Projectedqx;Frame.DLoads.Projectedqy]
#Rotation of global-local systems matrix
r = @SMatrix [cos sen;
 -sen cos]
#Scale projected-local systems matrix
e = @SMatrix [sen 0;
 0 cos]
#Transformations of load systems
ql1 = r*qg
ql2 = r*e*qp
#Total local distributed load
qlx = ql0[1]+ql1[1]+ql2[1]
qly = ql0[2]+ql1[2]+ql2[2]
#Equivalent nodal loads
pleq[:,1,Element] = @SMatrix [
 qlx*L/2;
 qly*L/2;
 qly*(L^2)/12;
 qlx*L/2;
 qly*L/2;
 -qly*(L^2)/12
 ]
pgeq = R[:,:,Element]'*pleq[:,:,Element]
Peq[dof_elements[Element,:]] = Peq[dof_elements[Element,:]] + pgeq
#Local geometric stiffness matrix
kgl[:,:,Element] = -@SMatrix [0 0 0 0 0 0;
 0 36/(30*L) 1/10 0 -36/(30*L) 1/10;
 0 1/10 4*L/30 0 -1/10 -L/30;
 0 0 0 0 0 0;
 0 -36/(30*L) -1/10 0 36/(30*L) -1/10;
 0 1/10 -L/30 0 -1/10 4*L/30]
end
return Matrix_and_Vectors(R, kl, K, pleq, Peq, dof_elements, kgl)
end
```

Algorithm 04 solves the system of equations to determine reactions and displacements. Julia has in its community, through GITHUB, various packages and libraries focused on scientific computing, including native implementations of linear algebra operations based on LAPACK libraries (Demmel, 1989). To use these libraries, they must be included in the developed program using the “include” command.

Algorithm 05 solves the eigenvalue and eigenvector problem for linear buckling of plane frames, determining the critical buckling coefficients and their respective strain shapes, which can be visualized in lines 10 to 13. In more complex systems, with more degrees of freedom, the stiffness matrix obtained by summing the elastic and geometric components is of large dimension, requiring significant computational effort. The “eig” technique from the Linear Algebra package is a solution for this. One way to overcome this inconvenience is to use the Lanczos algorithm for symmetric matrices. This algorithm allows truncation of the number of corresponding eigenvalues and eigenvectors, drastically reducing the cost of the operation. To do this, it is necessary to use the ARPACK package, initially developed in Fortran 77 and widely used in scientific computing languages such as MATLAB, Python, Octave, and Mathematica. Line 10 of Algorithm 05 presents the syntax for the eigs command from the ARPACK.jl library in the Julia language.

Algorithm 04 Matrix solution of the linear equation system

```
function EqSolution(MV:: Matrix_and_Vectors, Id:: Identifiers)
    #Capture of the model information
    K = MV.K
    Peq = MV.Peq
    P = Id.P
    D = Id.D
    PF = Id.PF
    DC = Id.DC
    doff = Id.doff
    NumbDOF = size(P,1)
    #Division in submatrices for the solution Free-Constraint
    KFF = K[1:doff,1:doff]
    KFC = K[1:doff,doff+1:NumbDOF]
    KCF = K[doff+1:NumbDOF,1:doff]
    KCC = K[doff+1:NumbDOF,doff+1:NumbDOF]
    PeqF = Peq[1:doff]
    PeqC = Peq[doff+1:NumbDOF]
    PTF = PF+PeqF
    #Answer of the analysis
    DF = KFF\(PTF-KFC*DC)
    PTC = KCF*DF+KCC*DC
    PC = PTC-PeqC
    #Final vectors of loads and displacements
    P[doff+1:NumbDOF,1] = PC
    D[1:doff,1] = DF
    PT = P+Peq
    return Answers(KFF, DF, PC, P, D, PT)
end
```

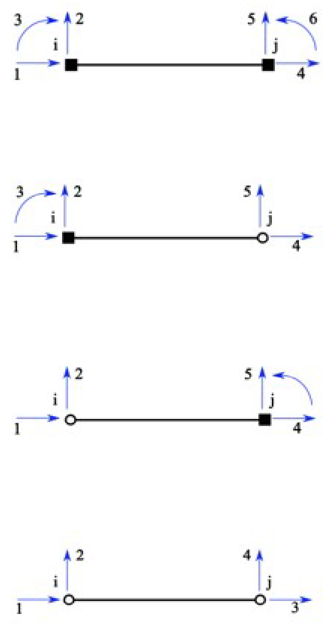
Algorithm 05 Buckling factor and buckling modes by the eigenvalues and eigenvectors solution

```
function CharacProblSol(Id:: Identifiers, Ans:: Answers, IF:: Solicitations)
    #Capture of the model
    doff = Id.doff
    KFF = Ans.KFF
    Kg = IF.Kg
    DC = Id.DC
    #Solution using the nomeation of free and constraint dof
    KgFF = Kg[1:doff,1:doff]
    #Eigenvalues and eigenvectors for 2 symmetric matrices with 1 eigen value (nev) which type of eigenvalue is smallest magnitude (SM)
    λcr,d = eigs(Symmetric(KFF),Symmetric(KgFF), nev=1, which=:SM)
    λcr = real(λcr); d = real(d)
    DBuck = [d;DC]
    return Buckling_Id(λcr, DBuck)
end
```

3 RESULTS AND DISCUSSION

3.1 Generated frames and validation

Within the code, the finite element of a plane frame bar has two nodes and six degrees of freedom. In the program, this element can be degenerated into three others that will be characterized according to the number of fixed or pinned nodes present in the bar. Figure 4 shows the characteristics of the available elements. In the Figure 4, the indices i and j represent the initial and final nodes of the bar. The degeneration of the element is implemented through a conditional structure for nodes that can have indices 0 or 1. Index 1 represents that the node is pinned, and 0 represents that the node is fixed. The respective stiffness matrices for each element are presented in Figure 4. The matrices shown for each element follow the notation found in (Martha, 2019).



$if\ (i=0\ \&\&\ j=0) \rightarrow$

$$\begin{bmatrix} EA/L & 0 & 0 & -EA/L & 0 & 0 \\ 0 & 12EI/(L^3) & 6EI/(L^2) & 0 & -12EI/(L^3) & 6EI/(L^2) \\ 0 & 6EI/(L^2) & 4EI/(L) & 0 & -6EI/(L^2) & 2EI/(L) \\ -EA/L & 0 & 0 & EA/L & 0 & 0 \\ 0 & -12EI/(L^3) & -6EI/(L^2) & 0 & 12EI/(L^3) & -6EI/(L^2) \\ 0 & 6EI/(L^2) & 2EI/(L) & 0 & -6EI/(L^2) & 4EI/(L) \end{bmatrix}$$

(a)

$if\ (i=1\ \&\&\ j=0) \rightarrow$

$$\begin{bmatrix} EA/L & 0 & 0 & -EA/L & 0 & 0 \\ 0 & 3EI/(L^3) & 3EI/(L^2) & 0 & -3EI/(L^3) & 0 \\ 0 & 3EI/(L^2) & 3EI/(L) & 0 & -3EI/(L^2) & 0 \\ -EA/L & 0 & 0 & EA/L & 0 & 0 \\ 0 & -3EI/(L^3) & -3EI/(L^2) & 0 & 3EI/(L^3) & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

(b)

$if\ (i=1\ \&\&\ j=0) \rightarrow$

$$\begin{bmatrix} EA/L & 0 & 0 & -EA/L & 0 & 0 \\ 0 & 3EI/(L^3) & 0 & 0 & -3EI/(L^3) & 3EI/(L^2) \\ 0 & 0 & 0 & 0 & 0 & 0 \\ -EA/L & 0 & 0 & EA/L & 0 & 0 \\ 0 & -3EI/(L^3) & 0 & 0 & 3EI/(L^3) & -3EI/(L^2) \\ 0 & 3EI/(L^2) & 0 & 0 & -3EI/(L^2) & 3EI/(L) \end{bmatrix}$$

(c)

$if\ (i=1\ \&\&\ j=1) \rightarrow$

$$\begin{bmatrix} EA/L & 0 & 0 & -EA/L & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ -EA/L & 0 & 0 & EA/L & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

(d)

Figure 4 Characteristics of the available elements in the program developed in Julia.

The structure presented in Figure 5 demonstrates all the modeling options available in the proposed program. Table 1 presents the properties and characteristics of the proposed model for validation, the vertical and horizontal bars formed by a concrete section with base bw and height h, with a modulus of elasticity E_c; the diagonal bar has a circular steel section with a diameter d and a modulus of elasticity E_s. To validate the code developed in the Julia language, a comparison is made between the results of linear buckling analysis of a typical frame using the proposed implementation and the SAP2000 software, as shown in Figure 6.

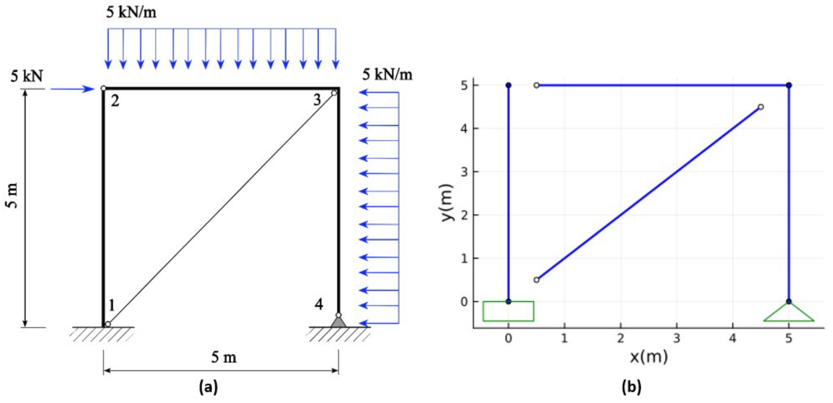


Figure 5 Frame used for validation of the buckling analysis program developed in Julia. (a) Adapted Analytical Model (François et al., 2021). (b) Computational modeling in the developed program.

Another feature developed in the proposed program is the automatic division of the implemented bar elements. This allows for increased discretization of the computational model, enabling the study of convergence and consequently gaining precision for the results. Figure 6 shows the deformed structure for the discretized validation example.

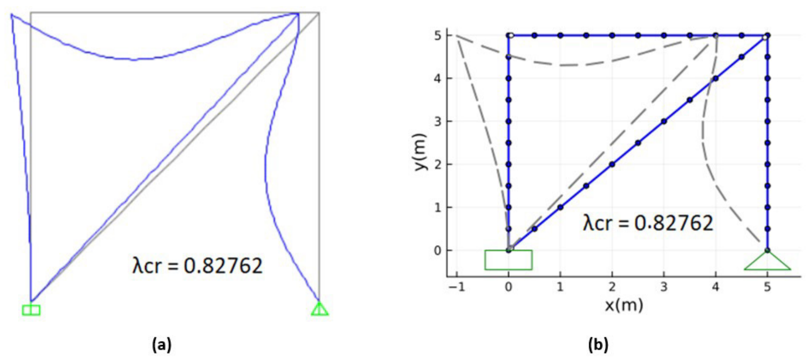


Figure 6 Deformed structure for the first mode of buckling. (a) Deformed Structure in SAP2000. (b) Strain of the structure in the developed code.

Table 1 Input values used in the frame.

Geometric and Material Properties		
Property	**Value**	**Unit**
bw	**0.20**	**m**
h	**0.40**	**m**
E_c	**2,10E+07**	**kN/m^2**
d	**0.05**	**m**
E_s	**2,10E+08**	**kN/m^2**

Table 2 Comparison of joint reactions between the Julia program and SAP2000.

Joint Reactions [kN or kN.m]		
DOF	**SAP2000**	**Julia**
116	10.532	10.532
117	19.816	19.816
118	0.9188	0.9188
119	9.468	9.468
120	5.184	5.184

From the results presented in Figure 6 and Table 2, the accuracy of the FEM code implemented in Julia for calculating the critical load of plane frames is demonstrated, when compared to the SAP2000 software, which is a program written in the object-oriented paradigm (Lallotra and Singhal, 2017), and the analytical solution.

3.2 Benchmarks

A benchmark is, by definition, an impartial evaluation that can be based on different characteristics of a language, allowing, under controlled conditions, the comparison of different strategies to define an optimal condition for the service of the tested code or software device (Rosen et al., 2006). This comparison is usually made on parameters of speed between two languages, and for the comparison conditions to be fair and equal, the analysis is typically performed on a code that is semantically written the same way in both languages, using native syntax that allows the same tasks to be compared fairly, simply measuring the script's processing efficiency. This performance efficiency is usually measured in terms of task execution time.

According to Lessmann et al. (2015), this technique is fundamental for identifying performance challenges in projects. Its importance grows as the need for high performance in routine execution increases, allowing the detection of computationally expensive processes and the application of writing or machine strategies to smooth out this process, consequently optimizing the code.

Being a compiled language, in Julia language the code processing speed is always subject to fluctuations that can depend on factors ranging from the computer hardware's capability performing the analysis to peaks and drops in RAM usage (Pereira and Benjamim Baptista, 2017). Because of this, it is a good practice to perform more than one simulation and use the average processing time as a parameter for a comparative analysis of performance between languages. Therefore, in the benchmark of this work, 22 repetitions per simulated frame were used before plotting the graphs.

The results are based on models of plane frames that vary by increasing the number of degrees of freedom. For the benchmark, the developed models consider five horizontal frames, increasing the degrees of freedom by replicating the vertical frames, as shown in the example in the Figure 7. The Figure 8, Figure 9 and Figure 10 show the performances of the characteristic problem-solving stage defining the critical buckling load and its associated strain modes. In the study, eleven simulations are performed, with the first model having 1221 and the eleventh 13371 degrees of freedom. The x-axis represents the number of free degrees (order of the elastic and geometric stiffness matrix used in the eigenvalue and eigenvector problem iteration). The y-axis shows the time taken in each language to perform this processing.

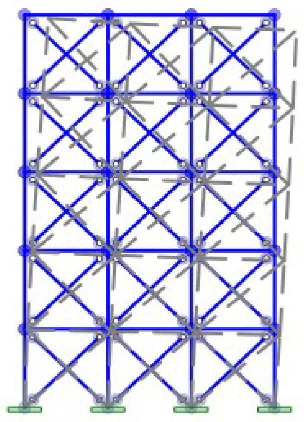


Figure 7 Example of a frame generated for the benchmarks, with 3 horizontal and 5 vertical spans

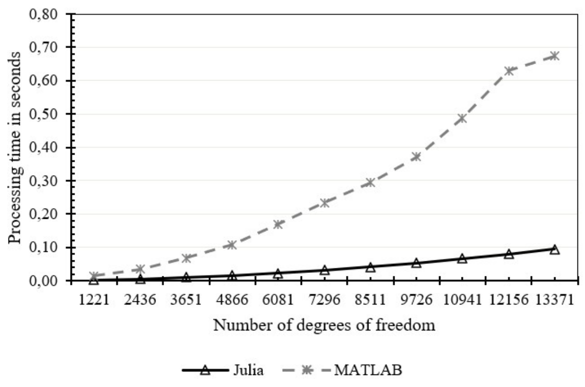


Figure *8* Benchmark of the code assembly in Julia and in MATLAB.

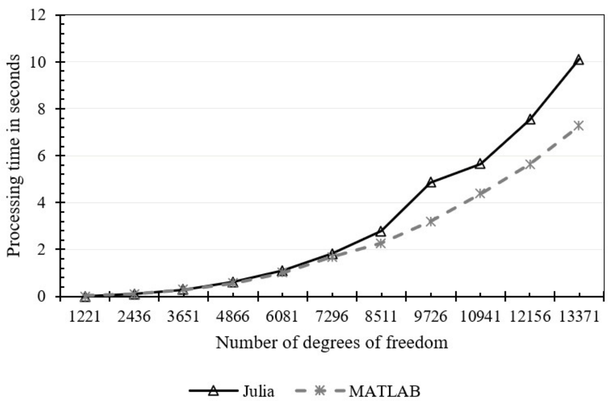


Figure 9 Benchmark of the code inversion in Julia and in MATLAB.

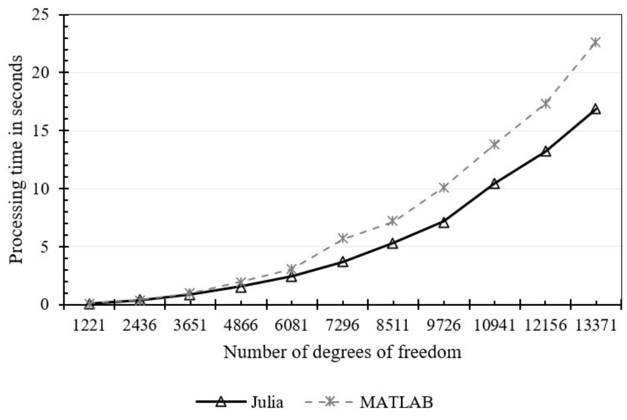


Figure 10 Benchmark of the code's characteristic problem in Julia and in MATLAB.

Therefore, as evident in Figure 8 and Figure 10, Julia demonstrated better processing time performance compared to MATLAB, with the advantage increasing as the number of free degrees of freedom increases. This is associated with the fact that Julia is a language designed for high-performance tasks, as mentioned throughout this work. However, in Figure 9, a positive difference for MATLAB in matrix inversion is noticeable, which is a consequence of using the Linear Algebra Package (LAPACK) that calls the Basic Linear Algebra Subprogramme (BLAS) package (Bird, Coombs and Giani, 2017).

Every time a BLAS function is called, there is an overhead that significantly affects the computing time, something common in finite element codes that work with matrix formulations throughout their scope. At this point, MATLAB activates multithreaded environments to execute such subroutines (Grim, Bueno Barajas and Gradvohl, 2019), which allows for time gain in multiprocessor architecture. This is the reason why, approximately from the simulation with 7296 degrees of freedom, MATLAB positively outperforms Julia's processing, which uses single thread processing as the default.

Moreover, MATLAB has integration with the Math Kernel Library (MKL) from Intel, which, recognizing this language as a giant already consolidated in the scientific programming field, provides standard implementations for efficient use of processor resources on computers of such brand, making the results found in the matrix inversion time comparisons not surprising (Unpingco, 2008). These conditions contrast with Julia, which in this work was not configured to have the most optimized use of hardware.

However, if the processing time for the simulation of the largest frame is added, which has 13371 degrees of freedom, overall, the structural analysis code written in Julia completes the entire process of the plane frame system in less time due to its relatively higher speed compared to MATLAB in most of the tested processes. As shown in Table 3, Julia performed the same computations in only 88.40% of the time MATLAB demanded.

Table 3 Processing time in seconds for the 3 analyzed processes.

	Processing time in seconds	
Process	**Julia**	**MATLAB**
Assembling [s]	0.09	0.67
Inversion [s]	10.12	7.27
Eigen [s]	16.88	22.70
Total [s]	27.09	30.64

Thus, it is interesting to move beyond a merely global analysis of benchmarks and average speed and observe how the code behaved in each of the simulations so that nuances that differentiate the two languages can be discussed.

In Figure 11, the graph shows the processing time used in the matrix assembly step in each of the simulations done for the calculation of the average time for the frame with 13371 degrees of freedom. As in any observation of events with repetitions, it is possible to notice that there are non-representative values that deviate significantly from the central tendency presented by most observations. These are outliers that are visibly contrasting for the studied case.

The outliers for each language curiously represent processing times smaller than the average of those considered "non-outliers", showing that within the developed research, computational stability was ensured from a minimum routine execution speed by the machine. In cases where there is some kind of mishap causing erratic behavior in reaching the speed, this time is reduced, meaning the system runs the code more efficiently. This was a pattern that remained consistent in all simulations for each of the benchmarks.

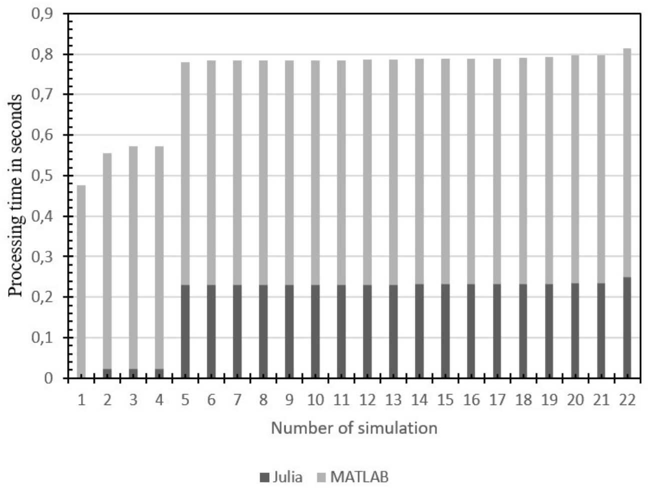


Figure 11 Time of each simulation in the benchmark of the matrix assembly.

Figure 12 shows the boxplot for the data shown in Figure 11, identifying outliers as those dispersed beyond the limits represented by the tips of the boxes. It is noticeable that the processing times in Julia have 5 non-representative values that disturb the descriptive measures of statistics aiming to express the behavior of all 22 repetitions. Therefore, their removal is necessary.

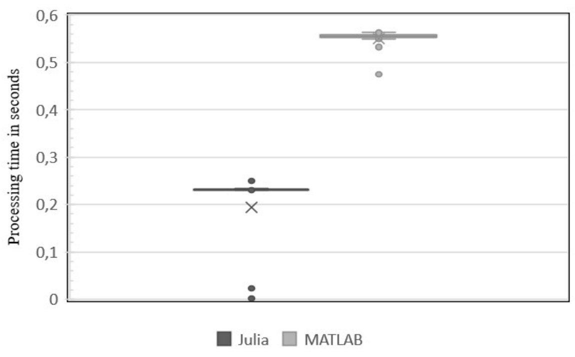


Figure 12 Outliers identification in processing time for: Julia and MATLAB.

In addition to the outliers removed from the data related to the Julia code, their respective counterparts in the MATLAB code were also excluded. After their removal in the former, there was a substantial decrease in the measures of sample dispersion, such as standard deviation and coefficient of variation. To ensure fair comparisons, it was necessary for both to reach relatively the same degree of variation in measurements, as can be seen in Table 4.

Table 4 Variation in measures of sample dispersion after the removal of outliers.

Statistical Measures	With Outliers		Without Outliers	
	Julia	MATLAB	Julia	MATLAB
Minimum [s]	0.00	0.47	0.23	0.55
Maximum [s]	0.25	0.56	0.23	0.56
Average [s]	0.19	0.55	0.23	0.56
Standard Deviation [s]	0.09	0.02	0.00	0.00
Coefficient of Variation	44%	3%	1%	1%

With the removal of the identified outliers, both datasets showed merely residual dispersion measured by a coefficient of variation of 1%. This value indicates that both languages achieved the same level of statistical interference absence in the observations. Now, the presented average comparisons become fair and consistent, as evident from Figure 13, showing that Julia's times were within the margin of 41.7% of MATLAB's times.

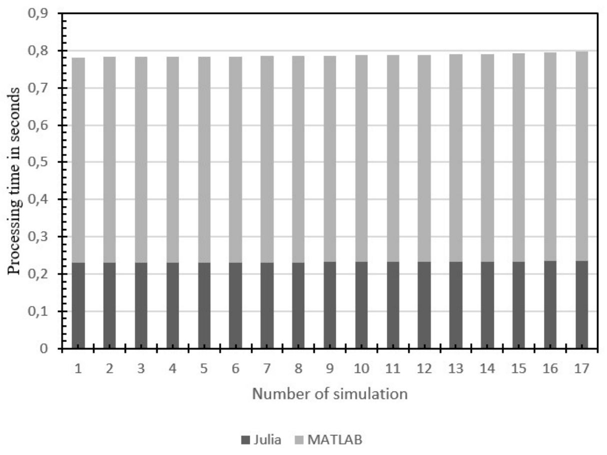


Figure 13 Time of each simulation in the benchmark of the matrix assembly after outliers removal.

In addition, when presenting the data in a sorted manner, it is noticeable that, before removing the outliers, there were sections of relative differences between processing times. These differences disappeared with the applied statistical treatment, and the fluctuations presented for both languages were very similar, with extremely low ranges of variation, as shown in Figure 14 and Figure 15, comparing such fluctuations before and after outlier removal.

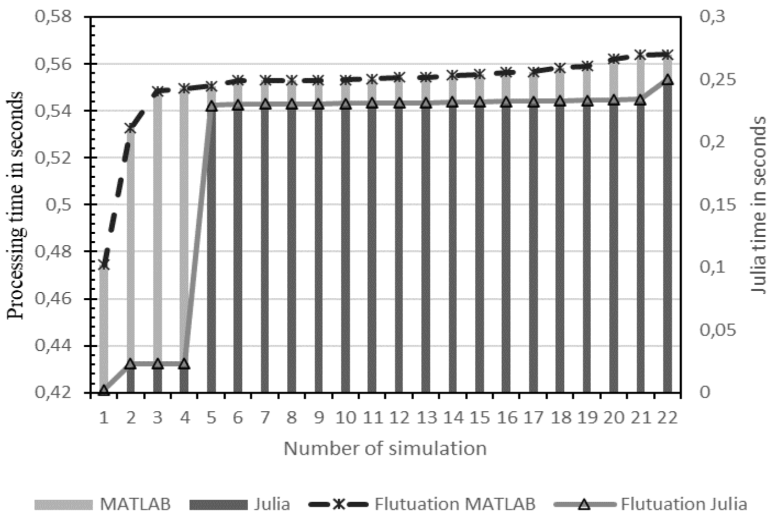


Figure 14 Processing times and fluctuations in benchmark with some dispersion degree.

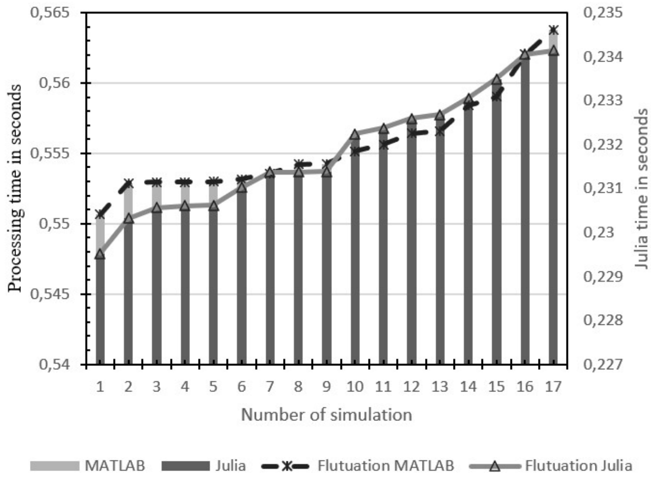


Figure 15 Processing times and fluctuations in benchmark with homogenized dispersion.

4 CONCLUSION

This paper presented the formulation of linear buckling using the Finite Element Method developed in the Julia programming language. Computational implementation was carried out to determine the critical buckling load of plane frame structures and their deformed shapes. The specific objective included the analysis of plane frames with different types of linear elements programmed in the code.

The Julia programming language offers several advantages that make it a strong alternative to languages commonly used in technical and scientific computing, such as MATLAB and Python. Julia features JIT compiler working with a paradigm of structures and functions combined with multiple dispatch. By knowing the type of each element in advance, Julia can search for the correct method calls during compilation time, avoiding runtime type checking, which contribute to and explain all the positive results achieved in processing speed benchmarks compared to interpreted languages. Some of the main conclusions of the study include:

1. With a syntax and semantics extremely like MATLAB, Julia stands out as an open-source alternative for works conducted within universities and the industry, eliminating the need for financial investments in the development of codes and projects that are traditionally written using globally established paradigms.
2. With straightforward syntax and a mastery of the concepts discussed here, it is already possible to write a code in Julia that outperforms MATLAB in processing speed for common and easily generalizable problems, such as loops with matrix operations and numerical solutions of eigenvalues and eigenvectors.
3. Even considering the matrix inversion process, which is a part of any computational implementation routine of the Finite Element Method, Julia still excels in processing time for a structure, as was the case with the studied plane frames, completing the analysis in 88.40% of the time compared to the interpreted language used in the comparisons.
4. For problems with more degrees of freedom, after a fair comparison with the removal of outliers and equalization of the coefficient of variation of the time samples that made up the benchmarks, Julia demonstrated a solid and stable victory in processing for loops, which is mandatory of any incremental or interactive process, present in nearly any computational algorithm, achieving the same results as MATLAB in only 41.7% of the time.

Future research could enhance the development of libraries and packages for structural analysis in this relatively new language and extend the results demonstrated here in the stability analysis problem to other branches of structural analysis, such as linear and nonlinear dynamics. This could involve a deeper exploration of different programming paradigms to understand how these changes make Julia's programming style even more efficient.

Acknowledgements

The authors gratefully acknowledge the support from PROPESP/UFPA (PAPQ).

Author's Contributions: Software, Writing – original draft, Data curation, Formal Analysis, Validation, LV Lobo; Conceptualization, Supervision, Visualization, Methodology, Writing – review & editing, EML Silva.

Editor: Marco L. Bittencourt

References

Aho, J., Vuotikka, A.-J. and Frondelius, T. (2019) ‘Introduction to JuliaFEM, an open source FEM solver’, *Rakenteiden Mekaniikka*, 52(3), pp. 148–159. Available at: https://doi.org/10.23998/rm.75103.

Ai, W. and Liu, Y. (2023) ‘Improving the convergence rate of Newman’s battery model using 2nd order finite element method’, *Journal of Energy Storage*, 67, p. 107512. Available at: https://doi.org/10.1016/j.est.2023.107512.

Alonso, R.C. de A. et al. (2015) ‘Modelagem computacional aplicada à flambagem elástica de colunas’, *Scientia Plena*, 11(8). Available at: https://doi.org/10.14808/sci.plena.2015.081310.

Antoshkin, A.D., Cherednichenko, A. V. and Savelyeva, I.Y. (2021) ‘Implementation of finite element method for solid mechanics problems in Julia programming language’, in *Journal of Physics: Conference Series*. IOP Publishing Ltd. Available at: https://doi.org/10.1088/1742-6596/1902/1/012096.

Bathe, K.-J. (2014) *Finite Element Procedures.* 2nd edn. New Jersey: Prentice Hall.

Bezanson, J. et al. (2012) 'Julia: A Fast Dynamic Language for Technical Computing'. Available at: http://arxiv.org/abs/1209.5145.

Bird, R.E., Coombs, W.M. and Giani, S. (2017) 'Fast native-MATLAB stiffness assembly for SIPG linear elasticity', *Computers and Mathematics with Applications*, 74(12), pp. 3209–3230. Available at: https://doi.org/10.1016/j.camwa.2017.08.022.

Byrne, S., Wilcox, L.C. and Churavy, V. (2019) *MPI.jl: Julia bindings for the Message Passing Interface*.

Canis, A. et al. (2013) 'LegUp: An open-source high-level synthesis tool for FPGA-based processor/accelerator systems', *Transactions on Embedded Computing Systems*, 13(2). Available at: https://doi.org/10.1145/2514740.

Coleman, C. et al. (2021) 'Matlab, Python, Julia: What to Choose in Economics?', *Computational Economics*, 58(4), pp. 1263–1288. Available at: https://doi.org/10.1007/s10614-020-09983-3.

Cook, R. et al. (2001) *Concepts and Applications of Finite Element Analysis.* 4th edn. Wiley.

Cuvelier, F., Japhet, C. and Scarella, G. (2016) 'An efficient way to assemble finite element matrices in vector languages', *BIT Numerical Mathematics*, 56(3), pp. 833–864. Available at: https://doi.org/10.1007/s10543-015-0587-4.

Dabrowski, M., Krotkiewski, M. and Schmid, D.W. (2008) 'MILAMIN: MATLAB-based finite element method solver for large problems', *Geochemistry, Geophysics, Geosystems*, 9(4). Available at: https://doi.org/10.1029/2007GC001719.

Demmel, J. (1989) *LAPACK: A Portable Linear Algebra Libmry for Supercomputers.* Available at: https://doi.org/10.1109/SUPERC.1990.129995.

François, S. et al. (2021) 'Stabil: An educational Matlab toolbox for static and dynamic structural analysis', *Computer Applications in Engineering Education*, 29(5), pp. 1372–1389. Available at: https://doi.org/10.1002/cae.22391.

Grim, L.F.L., Bueno Barajas, J.A. and Gradvohl, A.L.S. (2019) 'Implementações paralelas para o algoritmo Online Sequential Extreme Learning Machine aplicado à previsão de material particulado', *Revista Brasileira de Computação Aplicada*, 11(2), pp. 13–21. Available at: https://doi.org/10.5335/rbca.v11i2.9089.

Hibbeler, R. (2010) *Resistencia Materiais.* 7th edn. São Paulo: PEARSON.

Huo, Z. et al. (2020) 'Designing an efficient parallel spectral clustering algorithm on multi-core processors in Julia', *Journal of Parallel and Distributed Computing*, 138, pp. 211–221. Available at: https://doi.org/10.1016/j.jpdc.2020.01.003.

Huo, Z., Mei, G. and Xu, N. (2021) 'juSFEM: A Julia-based open-source package of parallel Smoothed Finite Element Method (S-FEM) for elastic problems', *Computers and Mathematics with Applications*, 81, pp. 459–477. Available at: https://doi.org/10.1016/j.camwa.2020.01.027.

Kemmer, T., Rjasanow, S. and Hildebrandt, A. (2018) 'NESSie.jl – Efficient and intuitive finite element and boundary element methods for nonlocal protein electrostatics in the Julia language', *Journal of Computational Science*, 28, pp. 193–203. Available at: https://doi.org/10.1016/j.jocs.2018.08.008.

Lallotra, B. and Singhal, D. (2017) 'State of the Art Report - A Comparative Study of Structural Analysis and Design Software - STAAD Pro, SAP-2000 & ETABS Software', *International Journal of Engineering and Technology*, 9(2), pp. 1030–1043. Available at: https://doi.org/10.21817/ijet/2017/v9i2/170902211.

Lessmann, S. et al. (2015) 'Benchmarking state-of-the-art classification algorithms for credit scoring: An update of research', *European Journal of Operational Research*, 247(1), pp. 124–136. Available at: https://doi.org/10.1016/j.ejor.2015.05.030.

Martha, L. (2019) *Análise matricial de estruturas com orientação a objetos.* 1st edn. Rio de Janeiro: Elsevier.

de Oliveira Garcia, R. and Paraguaia Silveira, G. (2017) 'Métodos numéricos aplicados às equações de Euler: comparação entre MatLab, Octave e Fortran', *C.Q.D. – Revista Eletrônica Paulista de Matemática*, 11, pp. 65–88. Available at: https://doi.org/10.21167/cqdvol11201723169664roggps6588.

Pereira, J.M. and Benjamim Baptista, M. (2017) *Linguagem de programação JULIA: uma alternativa open source e de alto desempenho ao MATLAB.* João Pessoa.

Performance Tips *(2023)* https://docs.julialang.org/en/v1/manual/performance-tips/#Performance-critical-code-should-be-inside-a-function.

Rapo, M., Aho, J. and Frondelius, T. (2017) 'Natural Frequency Calculations with JuliaFEM', *Rakenteiden Mekaniikka*, 50(3), pp. 300–303. Available at: https://doi.org/10.23998/rm.65040.

Rosen, C. et al. (2006) 'Implementing ADM1 for plant-wide benchmark simulations in Matlab/Simulink', *Water Science and Technology*, 54(4), pp. 11–19. Available at: https://doi.org/10.2166/wst.2006.521.

Unpingco, J. (2008) 'Some comparative benchmarks for linear algebra computations in MATLAB and scientific Python', in *2008 Proceedings of the Department of Defense High Performance Computing Modernization Program: Users Group Conference - Solving the Hard Problems*, pp. 503–505. Available at: https://doi.org/10.1109/DoD.HPCMP.UGC.2008.49.

Verdugo, F. and Badia, S. (2021) 'The software design of Gridap: a Finite Element package based on the Julia JIT compiler'. Available at: https://doi.org/10.1016/j.cpc.2022.108341.

Xiao, L. et al. (2021) 'Julia Language in Computational Mechanics: A New Competitor', *Archives of Computational Methods in Engineering* [Preprint]. Available at: https://doi.org/10.1007/s11831-021-09636-0.

Zhu, M., McKenna, F. and Scott, M.H. (2018) 'OpenSeesPy: Python library for the OpenSees finite element framework', *SoftwareX*, 7, pp. 6–11. Available at: https://doi.org/10.1016/j.softx.2017.10.009.

Zuo, W., Bai, J. and Cheng, F. (2014) 'EFESTS: Educational finite element software for truss structure. Part I: Preprocess', *International Journal of Mechanical Engineering Education*. SAGE Publications Inc., pp. 298–306. Available at: https://doi.org/10.1177/0306419015574637.

Zuo, W., Huang, K. and Cheng, F. (2017) 'EFESTS: Educational finite element software for truss structure - Part 3: Geometrically nonlinear static analysis', *International Journal of Mechanical Engineering Education*, 45(2), pp. 154–169. Available at: https://doi.org/10.1177/0306419016689503.