FERNANDO GEREMIAS TONI

Parallelized element-by-element architecture for structural analysis of flexible pipes using finite macroelements

Sao Paulo

2018

FERNANDO GEREMIAS TONI

Parallelized element-by-element architecture for structural analysis of flexible pipes using finite macroelements

Master's thesis presented to Escola Politécnica da Universidade de São Paulo in fulfillment of the requirements for the Master of Science degree

Sao Paulo
2018

FERNANDO GEREMIAS TONI

Parallelized element-by-element architecture for structural analysis of flexible pipes using finite macroelements

Master's thesis presented to Escola Politécnica da Universidade de São Paulo in fulfillment of the requirements for the Master of Science degree

Area: Mechanical Engineering

Advisor: Professor Doutor Clóvis de Arruda Martins

Sao Paulo
2018

Este exemplar foi revisado e corrigido em relação à versão original, sob responsabilidade única do autor e com a anuência de seu orientador.

São Paulo, _____ de _____ de _____

Assinatura do autor: _____

Assinatura do orientador: _____

Catalogação-na-publicação

Toni, Fernando Geremias
Parallelized element-by-element architecture for structural analysis of flexible pipes using finite macroelements / F. G. Toni -- versão corr. -- São Paulo, 2018.
229 p.

Dissertação (Mestrado) - Escola Politécnica da Universidade de São Paulo. Departamento de Engenharia Mecânica.

1.Tubos flexíveis 2.Método dos Elementos Finitos 3.Métodos Numéricos 4.Arquiteturas Paralelas I.Universidade de São Paulo. Escola Politécnica. Departamento de Engenharia Mecânica II.t.

Acknowledgments

To my parents, for all the support provided during the accomplishment of this work and for always encouraging me to invest in my personal education. To my advisor, by whom I have great admiration for his vast experience and wisdom. To the "*Laboratório de Mecânica Offshore - LMO*" of the Polytechnic School of the University of Sao Paulo, for providing all the necessary infrastructure to this work, and to CNPq for the financial support (161991/2015-7).

Abstract

Flexible pipes are used in the offshore oil production to transport fluid and gas from the sea bead to the floating stations, and vice versa. These pipes have several concentric layers, of different materials, geometries and structural functions, since they are exposed to adverse operating environments, subjected to high internal and external pressures, high axial stresses and a series of dynamic loads. The local analysis is an important stage of a flexible pipe design and it consists on determining the stresses and strains distributions along the layers of the pipe. Multipurpose finite element packages are commonly used in the local analysis of flexible pipes, but they possess many limitations due to its generic nature, varying from the absence of specific tools for model creation to heavy restrictions of the number of degrees-of-freedom to make computational processing feasible.

At the Polytechnic School of the University of São Paulo, within a research line in progress, several finite macroelements were formulated specifically for structural analysis of flexible pipes, taking into account their particularities, such as geometric patterns and layers assemblage. However, the numerical tools that implement these elements present very high memory and processing consumptions, limiting its usage for large-scale models. Therefore, this work has been motivated by memory and processing limitations of finite element structural analysis of flexible pipes for offshore applications.

In this context, the Element-by-Element method, which does not require the global stiffness matrix, was chosen for its potential in memory reduction and processing capabilities, given its scalability and ease of parallelization. After an extensive literature review on numerical methods regarding the EBE method, it was chosen the Element-by-Element Diagonal Preconditioned Conjugate Gradient Method (EBE-PCG) algorithm.

Aiming higher computational performance, the finite macroelements formulated by (PROVASI, 2013) were converted to the C++ language, implemented and parallelized in a new analysis tool, named as PipeFEM.

The diagonal preconditioned EBE-PCG algorithm was implemented and parallelized with OpenMP. The scalability of the PCG algorithm is directly influenced by the efficiency of the matrix-vector product, an operation that, in the element-by-element method, is computed in a local basis with the blocks that comprise the model, and that requires synchronization techniques when performed in parallel. Four different

synchronization strategies were developed, being the one based on geometric- and mesh-based mappings the most efficient of them. Numerical experiments showed a reduction of almost 92% in the EBE-PCG solution time of the parallelized version in comparison to the sequential one.

In order to compare the efficiency of PipeFEM with the well-established finite element package ANSYS, a simplified flexible pipe was modeled in both software. PipeFEM was approximately 82 times faster than ANSYS to solve the problem, spending 24.27 seconds against 33 minutes and 18 seconds. In addition to this, PipeFEM required much less memory, 61.8MB against 6.8GB in ANSYS. In comparison to the dense version of MacroFEM, a reduction of more than three orders of magnitude was achieved in memory consumption.

Despite the low the rate of convergence presented by the diagonal preconditioner, the implementation is very efficient in computational terms. Therefore, the objectives of this work were fulfilled with the development and application of the EBE method, allowing a reduction of memory and simulation costs.

Keywords: Flexible pipes, finite element method, numerical methods, parallel architectures.

Resumo

Tubos flexíveis são utilizados na produção *offshore* de petróleo para o transporte de fluidos e gás natural das estruturas submersas até as estações flutuantes, e vice-versa. Estes tubos possuem diversas camadas concêntricas, de diferentes materiais, geometrias e funções estruturais, pois são expostos a ambientes adversos de operação, nos quais são submetidos à elevadas pressões internas e externas, elevados carregamentos e tensões axiais, além de uma série de carregamentos dinâmicos. A análise local é uma etapa importante do dimensionamento de um tubo flexível e consiste em determinar as distribuições de tensões e deformações ao longo das camadas do tubo. Pacotes multiuso de elementos finitos são comumente utilizados na análise local de tubos flexíveis, mas, devido as suas naturezas genéricas, possuem limitações que variam desde a ausência de ferramentas específicas para a criação de modelos até restrições pesadas no número total de graus de liberdade para tornar exequível o processo computacional.

Na Escola Politécnica da Universidade de São Paulo, dentro de uma linha de pesquisa em andamento, diversos macroelementos finitos foram formulados especificamente para a análise estrutural de tubos flexíveis, levando em consideração as suas particularidades, como por exemplo padrões de geometrias e de montagem de camadas. Entretanto, a ferramenta numérica que implementa esses elementos apresenta elevado consumo de memória e de processamento, o que limita o seu uso para modelos de grande escala. Portanto, este trabalho foi motivado por limitações de memória e processamento em análises estruturais com o método dos elementos finitos para tubos flexíveis de aplicações *offshore*.

Neste contexto, o método elemento-a-elemento, caracterizado pela eliminação da matriz global de rigidez, foi escolhido devido ao seu potencial de redução de consumo de memória e às suas capacidades de processamento, dada a sua escalabilidade e facilidade de paralelização. Após uma extensa revisão bibliográfica em métodos numéricos a respeito do método EBE, foi escolhido a versão diagonalmente precondicionada do método do gradiente conjugado (EBE-PCG).

Com o intuito de se obter maior performance computacional, os macroelementos finitos formulados por (PROVASI, 2013) foram convertidos para a linguagem C++,

paralelizados e implementado em uma nova ferramenta de análise chamada de PipeFEM, totalmente escrita em C++ e que explora paralelismo em todos as etapas.

O algoritmo EBE-PCG foi implementado e paralelizado com OpenMP. A escalabilidade do algoritmo PCG é diretamente influenciada pela eficiência do produto entre matriz e vetor, uma operação que no método elemento-a-elemento é calculada na base local com os blocos que compõem o modelo, o que requer técnicas de sincronização quando realizada de modo paralelo. Quatro diferentes estratégias de sincronização foram desenvolvidas, sendo a mais eficiente delas a que utilizada mapeamentos baseados em características da geometria e malha. Experimentos numéricos mostraram uma redução de quase 92% no tempo de simulação do algoritmo PCG da versão paralelizada em relação à sequencial.

De modo a comparar a eficiência do PipeFEM com o pacote bem estabelecido de elementos finitos, ANSYS, um tubo simplificado foi modelado em ambos os programas. PipeFEM foi aproximadamente 82 vezes mais rápido do que o ANSYS, gastando 24.27 segundos contra 33 minutos e 18 segundos. Além disso, PipeFEM consumiu muito menos memória, 61.8MB contra 6.8GB in ANSYS. Em comparação com a versão densa do MacroFEM, uma redução superior a três ordens de grandeza no consum e de memória foi obtida.

Assim, apesar da baixa taxa de convergência apresentada pelo precondicionador diagonal, a implementação está muito eficiente em termos computacionais. Portanto, os objetivos deste trabalho foram alcançados com o desenvolvimento e aplicação do método EBE, o que permitiu uma redução considerável dos custos de simulação e memória.

Palavras-chave: Tubos flexíveis, método dos elementos finitos, métodos numéricos, arquiteturas paralelas.

List of Acronyms

API	American Petroleum Institute
BiCG	Bi-Conjugate Gradient Method
CAD	Computer Aided Design
CISC	Complex Instruction Set Computing
EBE	Element-by-Element
FEM	Finite Element Method
GMRES	Generalized Minimal Residual Method
HDPE	High-density polyethylene
HW	Hughes-Winget Preconditioner
LMO	Laboratory of Offshore Mechanics of the University of Sao Paulo "*Laboratório de Mecânica Offshore da Universidade de São Paulo*"
OpenMP	Open Multi-Processing
MPI	Message Passing Interface
PA-11	Polyamide 11 or Nylon 11
PA-12	Polyamide 12 or Nylon 12
PCG	Preconditioned Conjugate Gradient Method
PSD	Preconditioned Steepest Descent Method
PVDF	Polyvinylidene Difluoride
RISC	Reduced Instruction Set Computer
VIV	Vortex-induced Vibration
XLPE	Cross-linked Polyethylene

List of Illustrations

List of Tables

Contents

1 Introduction

1.1 Flexible Pipe

Flexible pipes, Fig. 1.1, are essential components in the offshore oil production, since they are responsible for connecting the floating stations to the submerged equipment. Besides the transportation of oil and natural gas, they are also employed in the processes known as *gas lift* that consists of the fluid injection in the wells with the objective of increasing their productivity or service life.

Fig. 1.1 – Flexible pipe. Source: own authorship.

Flexible pipes are characterized by having high axial stiffness, but low bending stiffness, allowing deflections of large amplitude without being damaged. This flexibility is essential, since, during its operation, the pipe is subjected to a wide range of static and dynamic loads.

Several technological challenges are imposed by the hostile environment in which flexible pipes are immersed, starting with the high water depths in the extraction point

that can surpass three thousand meters in some cases of Brazilian pre-salt area. In these depths, the pipe is subjected to very high external pressures that could collapse it. Great depths are also a problem for *risers* (a flexible pipe configuration that connects a platform or ship to the seafloor installations), since longer hanging cable lengths means higher axial stresses. Besides that, flexible pipes are also subjected to very high internal pressures, from the internal fluid pumping process that could cause the explosion of the pipe, a phenomenon known as *burst*. Dynamic loads are also applied to flexible pipes during its installation and operation. Platform movements and sea currents generate traction, compressive, torsional and bending loads that could lead to its structural failure. These dynamic loads, in conjunction with vibration phenomenon, such as VIV (vortex-induced vibrations), could also lead the pipe to fatigue failure. Besides mechanical loads, flexible pipes are also exposed to sharp temperature variations and to corrosive agents. In order to support these loads and meet the functionality requirements, a flexible pipe must contain several layers of helically extruded metallic wires, helically extruded interlocked profiles and also extruded thermoplastics. Each of these layers is carefully designed to carry out important structural and non-structural functions that must work together for the full operation of the flexible pipe.

1.1.1 Flexible Pipes Layers

In this section, a general overview of the typical layers of a flexible pipe will be provided, emphasizing their main characteristics and structural functions. The typical nomenclature used for the layers of a flexible pipe is illustrated in Fig. 1.2, although it may vary slightly according to the manufacturer. In the following items, these layers will be individually described.

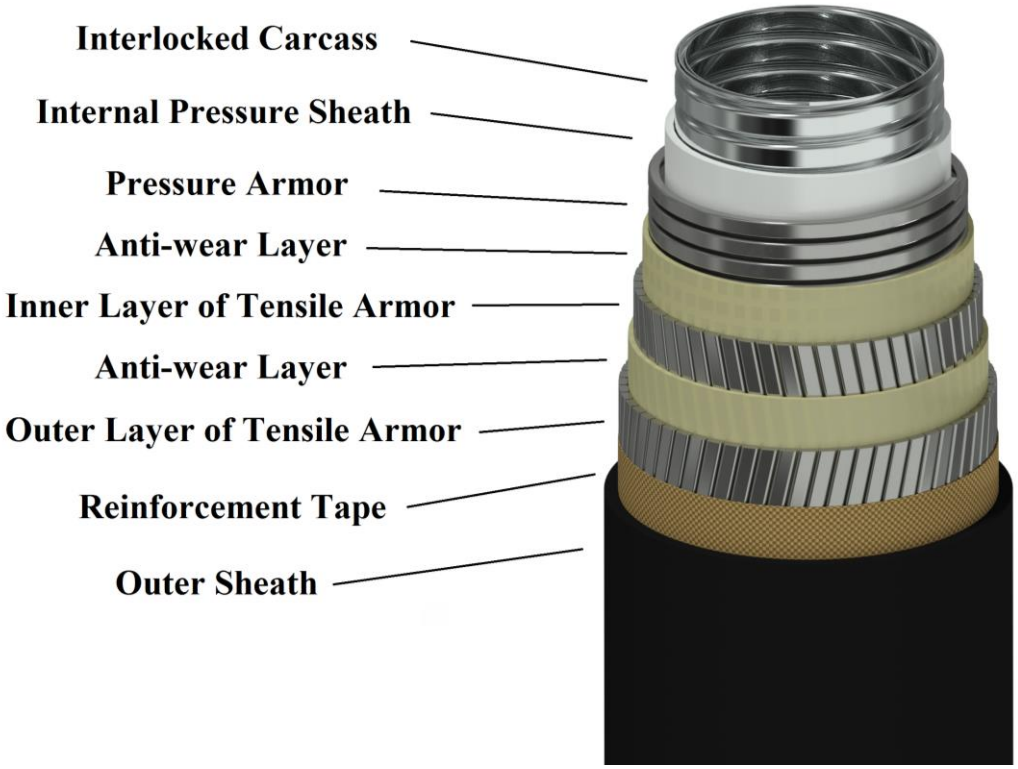

Fig. 1.2 – Layers of a unbonded flexible pipe. Source: own authorship.

1.1.1.1 Interlocked Carcass

The interlocked carcass (Fig. 1.3) is a metallic layer manufactured through the helically extrusion of a profile similar to the one illustrated in Fig. 1.4. That extrusion is performed by a series of forming rolls that progressively transform steel stripes into a fully interlocked section, as shown in Fig. 1.5. The carcass provides collapse resistance to the pipe, allowing it to support high external pressures.

Fig. 1.3 – Interlocked carcass. Source: own authorship.

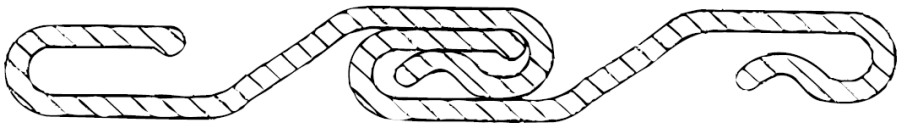

Fig. 1.4 – Example of interlocked carcass profile. Source: (API RP 17B , 2002).

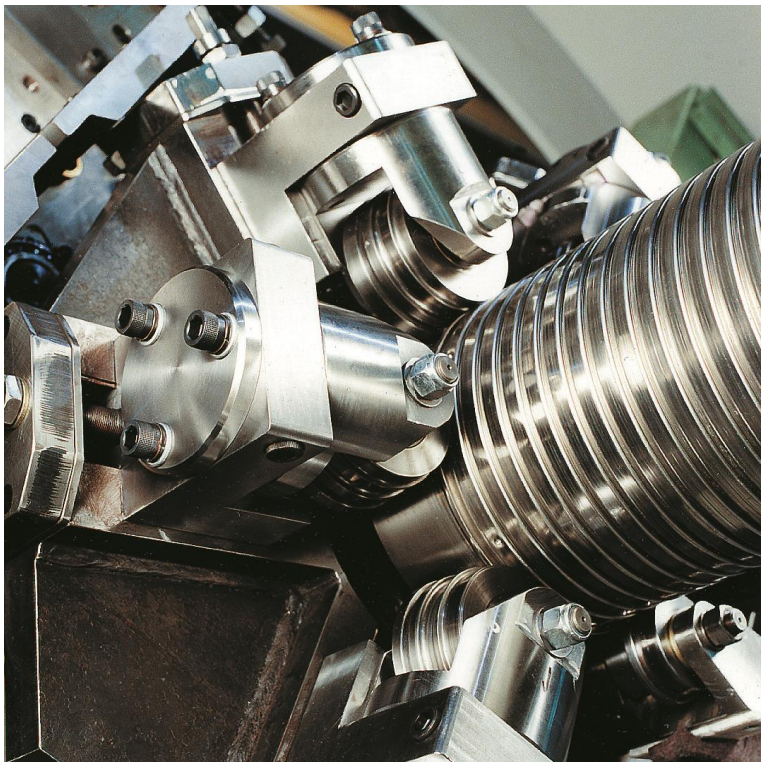

Fig. 1.5 – Carcass being manufactured. Source: (BARTELL, 2016).

For being in constant contact with the transported fluid, the carcass material must support the wear generated by the flow, which may contain aggressive agents, such as sand or corrosive products. According to (API RP 17B , 2002), the materials typically used for the carcass are:

- Carbon steel;
- Ferritic stainless steel (AISIs 409 and 430);
- Austenitic stainless steel (AISIs 304, 304L, 316, 316L);
- High-alloyed stainless steel (e.g., Duplex UNS S31803);
- Nickel based alloys (e.g., N08825).

Carbon steel is the cheapest option, but appropriate just for non-corrosive fluids. High-alloyed stainless steels are more expensive, but suitable for corrosive conditions. The material selection must also consider the fluid temperature and the presence of hydrogen sulfide (H_2S), carbon dioxide (CO_2), chlorides and oxygen in the transported fluid. Therefore, the material selection depends on the application particularities.

1.1.1.2 Internal Pressure Sheath

The main function of the internal pressure sheath is to seal the interlocked carcass, which is not waterproof, containing the fluid and maintaining the flow integrity. Compared with the others, this is a layer of simplified geometry, manufactured through the direct extrusion of a polymeric material over the carcass.

The typical materials used for the pressure sheath are thermoplastic polymers: high-density polyethylene (HDPE), cross-linked polyethylene (XLPE), Nylon or Polyamide 11 and 12 (PA 11 and PA12) and polyvinylidene difluoride (PVDF). The main characteristics of these materials are summarized in Table 1.1.

When selecting the material, it should be considered the aging of the polymer, once its mechanical properties degrade with time. If the transported fluid is a gas, it is important to consider the blistering resistance and the permeability of the selected material to the internal pressure sheath.

Table 1.1 – Typically thermoplastic polymer materials used for flexible pipes.

Polymer Material	General Compatibility Characteristics	Blistering Characteristics[1]
HDPE	Good ageing behaviour and resistance to acids, seawater and oil.	Good blistering resistance at low temperatures and pressures only.
	Weak resistance to amines and sensitive to oxidation.	
	Susceptible to environmental stress cracking (environments include alcohols and liquid hydrocarbons).	
XLPE	Good ageing behaviour and resistance to seawater, weak acids (dependent on concentrations and dosage frequency) and production fluid with high water cuts.	Better blistering resistance than HDPE, with positive results obtained in excess of 3000 psi.
	Weak resistance to amines and strong acids (dependent on concentrations and dosage frequency) and sensitive to oxidation. Less susceptible to environmental stress cracking than HDPE (environments include alcohols and liquid hydrocarbons).	
PA-11	Good ageing behaviour and resistance to crude oil.	Good blistering resistance up to 7500 psi and 100°C.
	Good resistance to environmental stress cracking.	
	Limited resistance to acids at high temperatures (recommend pH > 4.5 or TAN < 4.0). Limited resistance to bromides.	
	Weak resistance to high temperatures when any liquid water is present.	
PVDF	High resistance to ageing and environmental stress cracking.	Good blistering resistance up to 7500 psi and 130°C.
	Compatible with most produced or injected well fluids at high temperatures including alcohols, acids, chloride solvents, aliphatic and aromatic hydrocarbons and crude oil.	
	Weak resistance to strong amines, concentrated sulfuric and nitric acids and sodium hydroxide (recommend pH < 8.5)	

Notes:
1. Blistering characteristics are taken from [9]. Note that blistering characteristics will be a function of transported fluid, pressure, depressurization rate, and temperature.
2. The suitability of a material for a particular application should be verified by the manufacturer.

Source: (API RP 17B , 2002).

1.1.1.3 Pressure Armor

The pressure armor, Fig. 1.6, is a metallic layer manufactured through the helically extrusion of profiles similar to the ones from Fig. 1.7. By supporting internal loads in the radial direction generated by the fluid pumping process, the main function of the pressure armor is to prevent the occurrence of *burst*, a failure mode which causes the rupture of pressure armor due to excess of internal pressure.

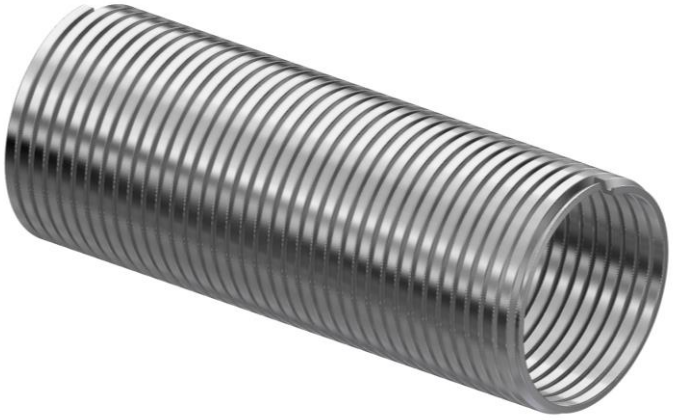

Fig. 1.6 – Interlocked pressure armor. Source: own authorship.

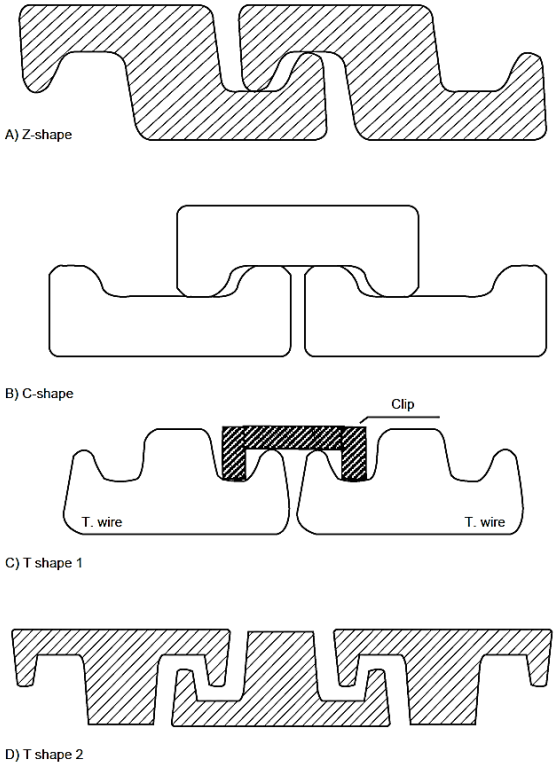

Fig. 1.7 – Pressure armor profiles. Source: (API RP 17B , 2002).

In conjunction with the interlocked carcass, the pressure armor also provides *collapse* resistance. During the pipe installation and launching, the caterpillar tensioners, Fig. 1.8, may cause the pipe *crush* and a part of these external radial loads is transmitted to the pressure armor. In addition, this layer also resists to constriction that arises from tensile armors traction, a phenomenon known as *squeeze*.

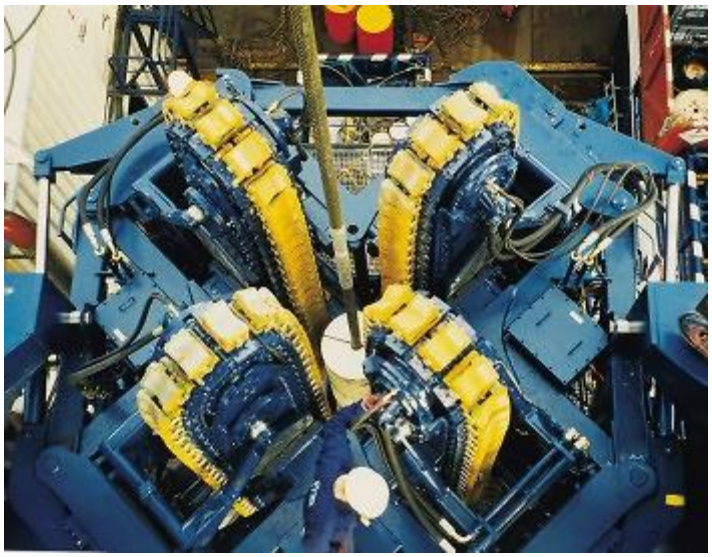

Fig. 1.8 – Caterpillar tensioner machine for flexible pipe launching. Source: (HUISMAN, 2008).

1.1.1.4 Antiwear layers

The anti-wear layers are tapes of polymeric material helically wrapped between the tensile armors and the pressure armor with the objective of reducing the friction wear between them, increasing the service life of the flexible pipe.

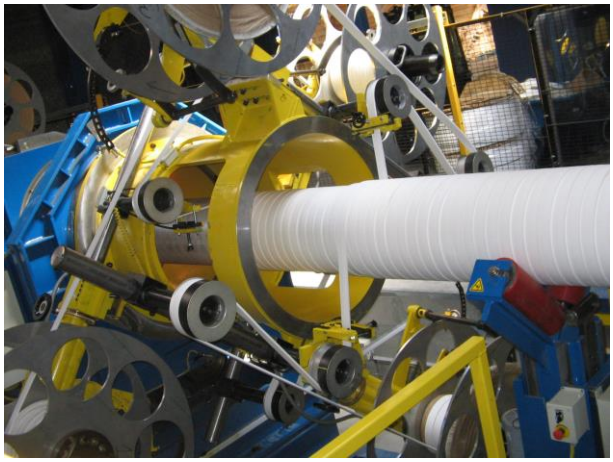

Fig. 1.9 – Manufacturing process of the anti-wear layers. Source: (BARTELL, 2016).

1.1.1.5 Tensile Armor

According to (API RP 17B , 2002), *the tensile armor layers typically use flat, round or shaped metallic wires, in two or four layers cross-wound at an angle between 20 degrees and 60 degrees*. These wires are helically wounded by large rotating machines, like the one from Fig. 1.10. The main function of a tensile layer is to resist the axial loads

and to the stresses caused by the action of environmental loads and by the platform movement. In general, a pipe must contain an even number of tensile layers, in a crossed configuration, i.e., intercalating layers whose helices grow in clockwise and anti-clockwise directions respectively, in order to obtain a torsionally balanced pipe. When the pipe has at least one pressure armor, lay angles near 35 degrees are used. In the absence of pressure armor, the tensile armors must have larger lay angles (close to 55 degrees), providing radial stiffness to the pipe.

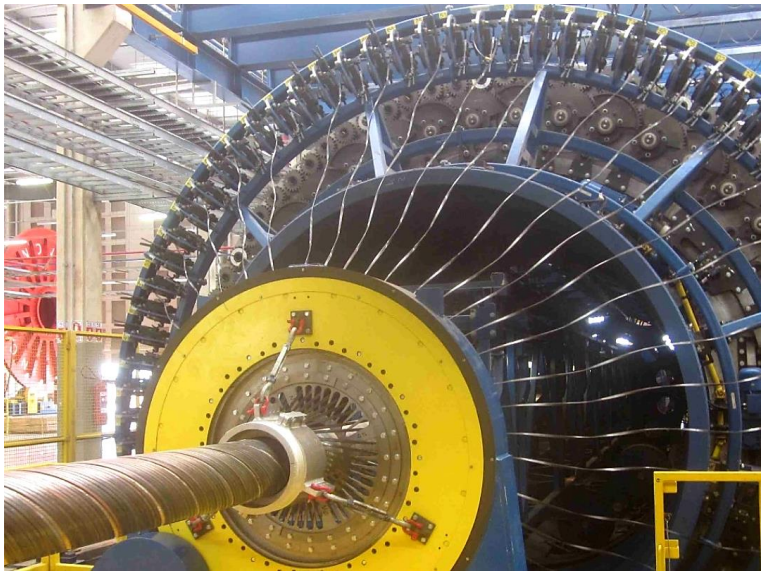

Fig. 1.10 – Tensile armor manufacturing process. Source: (MALI, 2016).

The typical material used for tensile armors is carbon steel. For great depths, which require very high axial strength, it is used high carbon steel instead of the conventional.

1.1.1.6 Antibuckling Tape

Reinforcement tapes are usually made of aramid, a heat-resistant and very strong synthetic fiber. They are applied over the outermost tensile armor layer, as shown in Fig. 1.11, in order to prevent the occurrence of *birdcaging*, Fig. 1.12, an instability phenomenon caused by the tensile armors buckling when they are excessively compressed. The reinforcement tape increases the critical compressive load that a pipe can endure before failing.

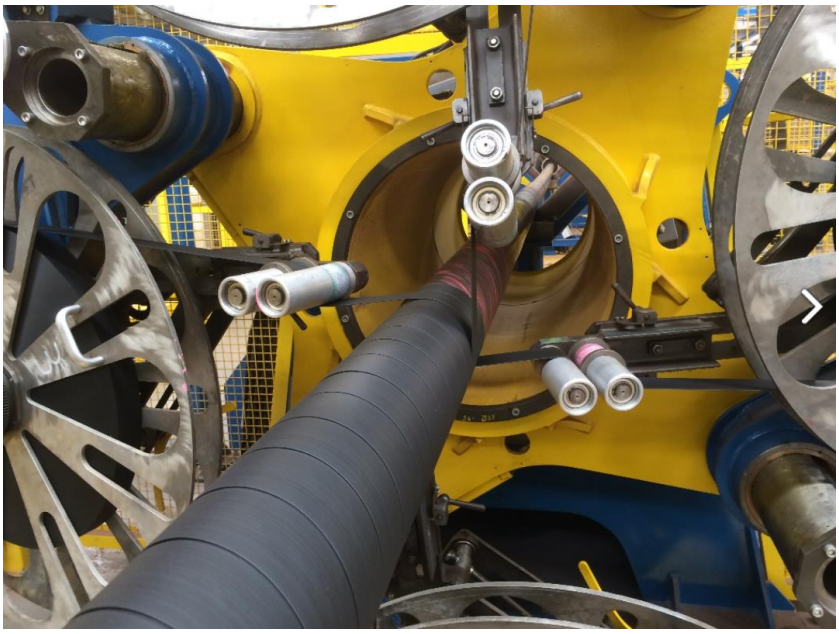

Fig. 1.11 – Reinforcement tape being applied over the tensile layer. Source: (MALI, 2016).

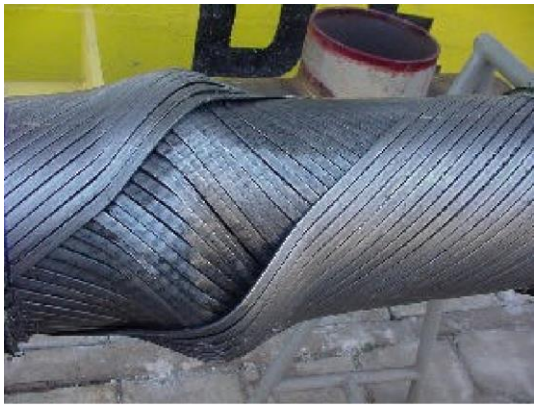

Fig. 1.12 – *Birdcaging* of the tensile armors. Source: (BRAGA & KALLEF, 2004).

1.1.1.7 Outer Sheath

The outer sheath has the function of sealing the pipe and protecting its internal layers against sea water corrosion and small impacts. It also provides additional radial stiffness to the tensile armors when they are compressed, acting in conjunction with the antibuckling tape to prevent the *bird caging* instability phenomenon. Like the internal pressure sheath, this layer is extruded directly over the former one.

1.1.2 Flexible Pipe Classification

In general, flexible pipes are specifically designed for every application, due to the particularities of each extraction region, such as water depth, sea conditions and specific operational requirements. Customized design enables optimized solutions, but generates a wide variety of existing pipe configurations.

For clarifying and organizational purposes, flexible pipes can be classified into the *bonded* and *unbonded* families. Illustrated in Fig. 1.13, "*a typical bonded flexible pipe consists of several layers of elastomer either wrapped or extruded individually and then bonded together through the use of adhesives or by applying heat and/or pressure to fuse the layers into a single construction*" (API RP 17B , 2002). Bonded flexible pipes can be further classified into "*smooth bore"* or "*rough bore"*, as shown in Table 1.2

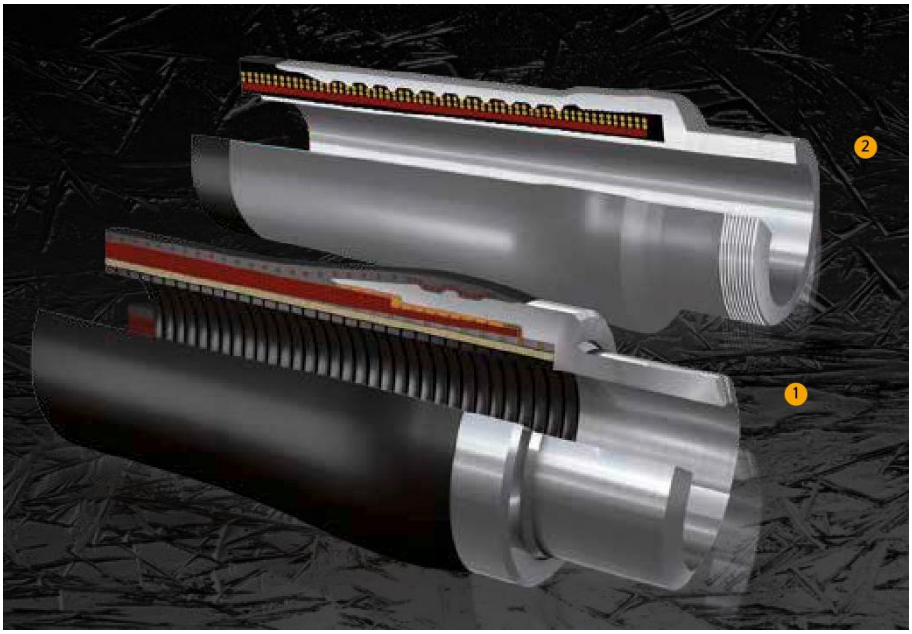

Fig. 1.13 – Bonded flexible pipe. Source: (CONTINENTAL, 2014)

Table 1.2 – Description of the bonded family of flexible pipes.

Layer No.	***Layer Primary Function***	***Bonded Flexible Pipe***	
		Smooth Bore Pipe	***Rough Bore Pipe***
1	Prevent collapse		Carcass
2	Internal fluid integrity	Liner	Liner
3	Hoop and tensile load resistance	Reinforcement layer(s)	Reinforcement layer(s)
4	External fluid integrity and protection	Cover	Cover

Notes:
1. All pipe constructions may include various non-structural layers, such as filler layers and breaker fabrics.
2. An external carcass may be added for protection purposes.
3. The number of crosswound reinforcement plies may vary, though generally is either two, four or six.

Adapted from: (API RP 17B , 2002).

Unbonded flexible pipes, Fig. 1.14, consist of concentric unbonded metallic helically extruded and unbonded polymeric extruded layers, with relative movement between them. According to (API RP 17B , 2002), unbonded flexible pipes can be further classified into "*smooth bore*", "*rough bore*" and "*rough bore reinforced*", as shown in Table 1.3.

Fig. 1.14 – Unbonded flexible Pipe. Source: own authorship.

Table 1.3 – Description of the unbonded family of flexible pipes.

Layer No.	*Layer Primary Function*	*Unbonded Flexible Pipe*		
		Smooth Bore Pipe	***Rough Bore Pipe***	***Rough Bore Reinforced Pipe***
1	Prevent collapse		Carcass	Carcass
2	Internal fluid integrity	Internal pressure sheath	Internal pressure sheath	Internal pressure sheath
3	Hoop stress resistance	Pressure armor layer(s)		Pressure armor layer(s)
4	External fluid integrity	Intermediate sheath		
5	Tensile stress resistance	Crosswound tensile armors	Crosswound tensile armors	Crosswound tensile armors
6	External fluid integrity	Outer sheath	Outer sheath	Outer sheath

Notes:
1. All pipe constructions may include various nonstructural layers, such as anti-wear layers, tapes, manufacturing aid layers, etc.
2. An external carcass may be added for protection purposes.
3. The pressure layer may be subdivided into an interlocked layer(s) and back-up layer(s).
4. The number of crosswound armor layers may vary, though generally is either two or four.
5. Thermal insulation may be added to the pipe.
6. The internal pressure and outer sheaths may consist of a number of sublayers.
7. Rough bore reinforced pipes are generally used for higher pressure applications.
8. The intermediate sheath for smooth bore pipes is optional when there is no external pressure or external pressure is less than the collapse pressure of the internal pressure sheath for the given application.

Adapted from: (API RP 17B , 2002).

Currently, unbonded flexible pipes are the most commonly used type, once they permit larger deflections than bonded flexible pipes (i.e. smaller radius of curvature). They also impose greater design challenges, due to the interactions between layers. The occurrence of relative movement with friction between layers makes the behavior of the pipe highly nonlinear. The understanding and prediction of this complex behavior is of great and practical interest for offshore pipe industry and therefore this work will focus on unbonded flexible pipes.

1.2 Flexible Pipe Design

For being employed in offshore applications to transport oil from great depths to the surface and also to pump fluids to the extraction wells, flexible pipes must be well-dimensioned; otherwise an accident would cause serious environmental and economic prejudices. The design of a flexible pipe consists, therefore, in the determination of an

economically feasible configuration which satisfies the requirements of functionality, performance and safety.

Due to the wide variety of loads and environmental conditions, a flexible pipe may present several failure modes. In Table 1.4 are summarized the most important structural failure modes of an unbonded flexible pipe and some alternatives to prevent them. The pipe design must therefore ensure that these failure modes will not occur during its installation and operation.

Table 1.4 – Check list of failure modes for primary structural design of unbonded flexible pipe.

Pipe Global Failure Mode to Design Against	Potential Failure Mechanisms	SA or DA[1]	Design Solution/Variables [Ref. API Spec 17J Design Criteria]
Collapse	1. Collapse of carcass and/or pressure armor due to excessive tension.	SA, DA	1. Increase thickness of carcass strip, pressure armor or internal pressure sheath (smooth bore collapse).
	2. Collapse of carcass and/or pressure armors due to excess external pressure.	SA, DA	2. Modify configuration or installation design to reduce loads.
	3. Collapse of carcass and/or pressure armor due to installation loads or ovalisation due to installation loads.	SA, DA	3. Add intermediate leak-proof sheath (smooth bore pipes).
	4. Collapse of internal pressure sheath in smooth bore pipe.	SA, DA	4. Increase the area moment of inertia of carcass or pressure armor.
Burst	1. Rupture of pressure armors because of excess internal pressure.	SA, DA	1. Modify design, e.g., change lay angle, wire shape, etc.
	2. Rupture of tensile armors due to excess internal pressure.	SA, DA	2. Increase wire thickness or select higher strength material if feasible.
			3. Add additional pressure or tensile armor layers.
Tensile failure	1. Rupture of tensile armors due to excess tension.	SA, DA	1. Increase wire thickness or select higher strength material if feasible.
	2. Collapse of carcass and/or pressure armors and/or internal pressure sheath due to excess tension.	SA, DA	2. Modify configuration designs to reduce loads.
	3. Snagging by fishing trawl board or anchor, causing overbending or tensile failure.	SA, DA	3. Add two more armor layers.
			4. Bury pipe.
Compressive failure	1. Birdcaging of tensile armor wires.	SA, DA	1. Avoid riser configurations that cause excessive pipe compression.
	2. Compression leading to upheaval buckling and excess bending (see also Upheaval Buckling failure mode).	SA, DA	2. Provide additional support/restraint for tensile armors, such as tape and/or additional or thicker outer sheath.
Overbending	1. Collapse of carcass and/or pressure armor or internal pressure sheath.	SA, DA	1. Modify configuration designs to reduce loads.
	2. Rupture of internal pressure sheath.	SA, DA	
	3. Unlocking of interlocked pressure or tensile armor layer.	SA, DA	
	4. Crack in outer sheath.	SA, DA	
Torsional failure	1. Failure of tensile armor wires.	SA, DA	1. Modify system design to reduce torsional loads.
	2. Collapse of carcass and/or internal pressure sheath.	SA, DA	2. Modify cross-section design (e.g. change lay angle of wires, add extra layer outside armor wires, etc.) to increase torsional capacity.
	3. Birdcaging of tensile armor wires.	SA, DA	
Fatigue failure	1. Tensile armor wire fatigue.	DA	1. Increase wire thickness or select alternative material, so that fatigue stresses are compatible with service life requirements.
	2. Pressure armor wire fatigue.	DA	2. Modify design to reduce fatigue loads.
Erosion	1. Of internal carcass.	SA, DA	1. Material selection.
			2. Increase thickness of carcass.
			3. Reduce sand content.
			4. Increase MBR.
Corrosion	1. Of internal carcass.	SA, DA	1. Material selection.
	2. Of pressure or tensile armor exposed to seawater, if applicable.	SA, DA	2. Cathodic protection system design.
	3. Of pressure or tensile armor exposed to diffused product.	SA, DA	3. Increase layer thickness.
			4. Add coatings or lubricants.

Notes:
1. SA = static application, DA = dynamic application.
2. Burst, tensile, overbending and torsional failure are not considered in isolation for final design of the flexible pipe.

Source: (API RP 17B , 2002).

Predicting the effect of the loads that are applied on a flexible pipe is not a trivial task and requires advanced engineering techniques, once the structural behavior of the pipe is highly nonlinear. This nonlinearity arises not only from the material and geometry, but also from the several frictional interactions between the layers. Thus, the flexible pipe design is a complex procedure and requires a multi-stage iterative process.

As can be seen in Fig. 1.15, the (API RP 17B , 2002) subdivides the design of flexible pipes for static applications into five stages:

- **Stage 1 – Material Selection**: in this stage the materials are selected in accordance to the environment characteristics (temperature, transported fluid corrosivity, etc.) and to the functional requirements.
- **Stage 2 – Cross-section configuration design:** the cross-section is defined based on the functional requirements of the pipe, such as a predetermined internal diameter to achieve the desired fluid flow rate. This stage requires the use of specific tools for structural calculations and checks.
- **Stage 3 – System configuration design:** this stage consists on determining the system configuration. For static applications, this stage is much simpler in comparison to the dynamic ones.
- **Stage 4 – Detail and service life design:** "*this stage includes the detailed design of ancillary components and corrosion protection. Service life analysis is also performed at this stage as it applies to the pipe and components*".
- **Stage 5 – Installation design:** "*this stage completes the design process and involves the selection/design of the installation system, including vessel, equipment, methodology, and environment conditions. Stage 5 requires detailed global and local analyses to confirm the feasibility of the selected installation system. For flowlines, this stage is-in many cases-critical for the pipe design, and it is therefore recommended that preliminary installation analyses be performed at an early stage in the design process*".

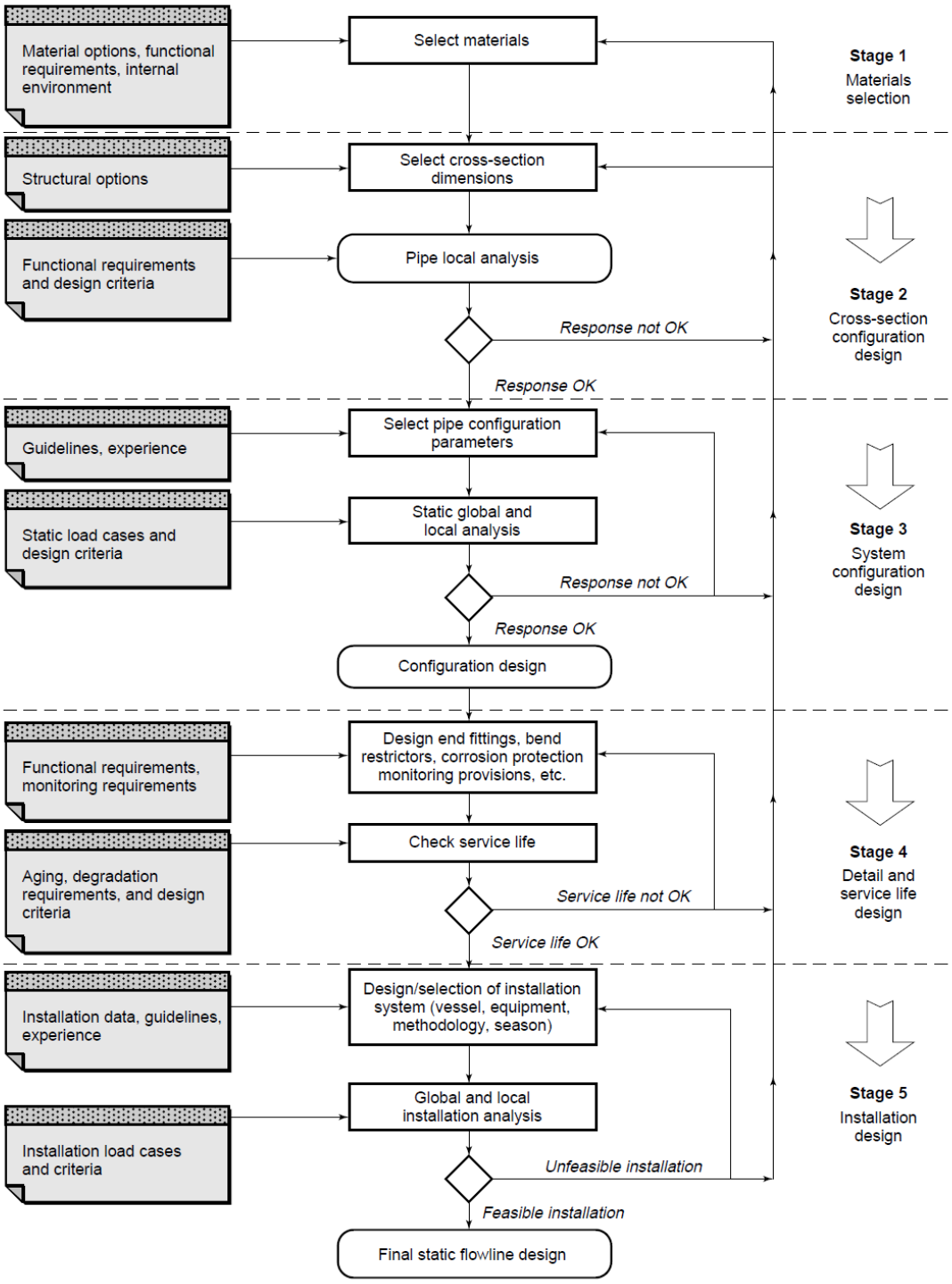

Fig. 1.15 – Static application design flowchart. Source: (API RP 17B , 2002).

The design for dynamic applications also follows a multi-stage iterative scheme, but with the particularities and complications of dynamic analysis, as shown in Fig. 1.16.

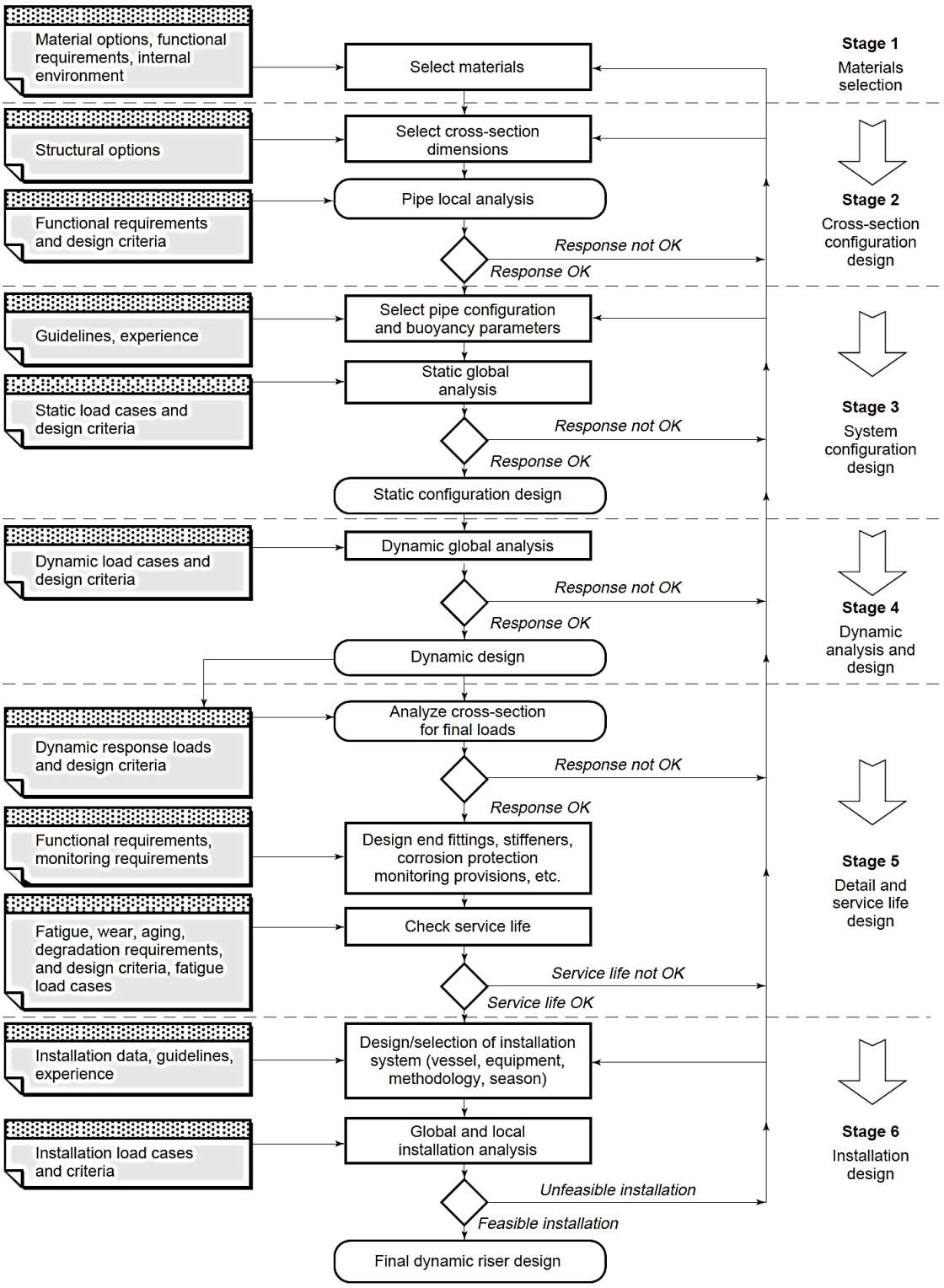

Fig. 1.16 – Dynamic application design flowchart. Source: (API RP 17B , 2002).

It is important to notice that, the global and local analyses are included in most of the design phases for both the static and dynamic applications. Due to the importance of these analyses, they will be discussed individually in the next items.

1.2.1 Global Analysis

"*Global analysis is performed to evaluate the global load effects on the pipe during all stages of installation, operation, and retrieval, as applicable. The static configuration and extreme response of displacement, curvature, force and moment from environmental effects should be evaluated in the global analysis*" (API RP 17B , 2002).

In the global analysis, the pipe is modeled as a curved line on the global scale. After the cross-section definition, the equivalent properties of the pipe can be evaluated, such as mass per unit length and axial, bending and torsional stiffnesses. These equivalent properties are employed in the global analysis to determine the efforts distribution along the pipe, without worrying about the stresses and strains values on the layers and the possible interaction effects between them.

This stage may require the use of specific numeric computational tools. *Flexcom*, *OrcaFlex*, *Deeplines* and *Riflex* are examples of dedicated commercial software for global analysis of flexible pipes. Fig. 1.17 illustrates an example of a global analysis performed with the software *OrcaFlex*.

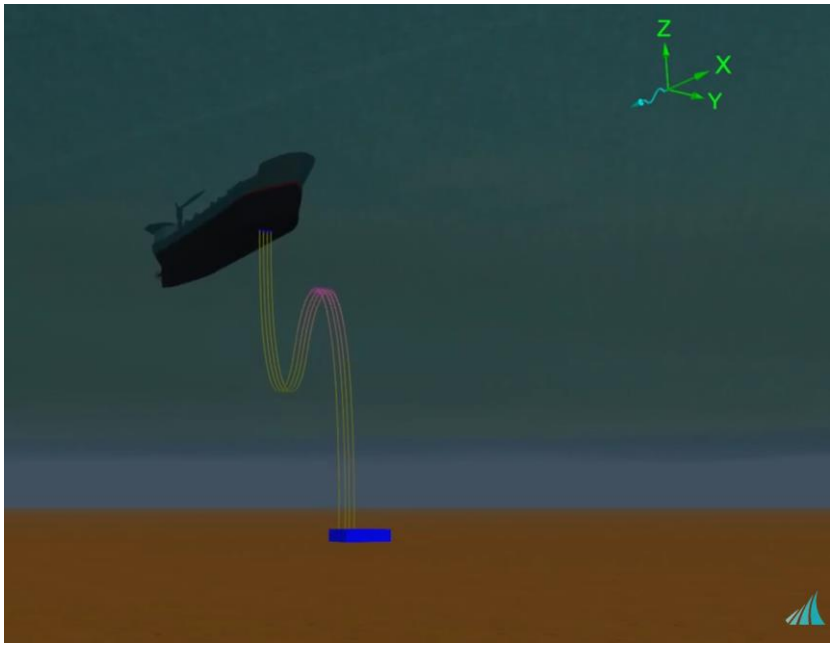

Fig. 1.17 – Global analysis performed on Orcaflex. Source: (PDL GROUP, 2015).

1.2.2 Local Analysis

"*Because of the composite layer structure of a flexible pipe, local cross-section analysis is a complex subject, particularly for combined loads. Local analysis is required to relate global loadings to stresses and strains in the pipe. The calculated stresses and strains are then compared to the specified design criteria for the load cases identified in the project design premise*" (API RP 17B , 2002).

Therefore, the local analysis consists of a more refined analysis, which objective is to determine the stresses and strains distributions along the flexible pipe layers, being of fundamental importance for the correct dimensioning of the same. Fig. 1.18 illustrates one example of local analysis, in which it can be seen the stress distribution along an interlocked carcass.

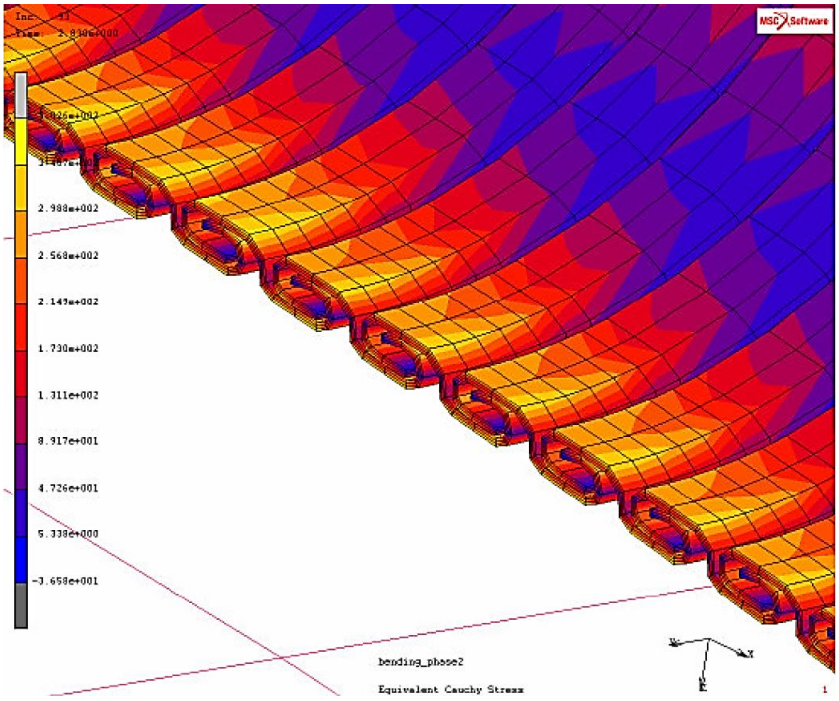

Fig. 1.18 – Detailed stresses analysis of an interlocked carcass. Source: (MUREN, 2007).

Over the last decades, several approaches were developed for the solution of the local analysis. They can be classified into analytical and numerical, each of them with advantages and disadvantages, that complement each other. The analytical methods consists of modeling the flexible pipe beyond a system of equations that can be analytically solved with a computer. In this case, the modeling of the pipe is the most complex activity in most of the time and often a number of hypothesis, assumptions or

simplifications are needed to simplify or make feasible the modeling. However, with modelling challenges overcome, the analytical methods are characterized by fast solutions. The numerical methods are mainly based on the finite element method (FEM) and, comparatively, require a much lower number of assumptions, which confers greater capability of solving more generic problems. Despite the advantages, the computational costs from numerical methods can become very high and even infeasible in some cases.

The analytical methods for local analysis present so far are not able to solve sliding problems of tensile armors with friction. Therefore, numerical methods are the only alternative in this case, in special the finite element method, which stands out for its ability of solving problems of complex and irregular geometries, besides the inclusion of nonlinearities in the model, such as material plasticity and nonlinear interactions between components involving friction. The finite element method also has disadvantages. Simulations of flexible pipes are problems of difficult convergence, given the high level of nonlinearity of the models. It is also possible to achieve incoherent or without physical sense solutions, requiring a critical evaluation of results by an experienced analyst. Simulation time and cost may not be viable for models with some millions of degrees of freedom, thus limiting its applicability. In some situations, the model can be simplified, assuming plan or axisymmetric hypothesis, or limiting the analysis to the essential layers for the understanding of the phenomenon of interest.

Multipurpose software, such ANSYS® and ABAQUS®, are generic packages for the finite element method, designed to meet the widest range possible of applications. Despite all available resources and solution methods, these computer programs show many limitations regarding the simulation of flexible pipes. In the preprocessing stage, the absence of specific CAD tools makes the pipe drawing a costly activity and difficult to be automatized. In some cases, the definition of contact pairs must be performed individually, making it a laborious task, given the high number of interactions between components. In the processing stage, limitations on the number of degrees of freedom make unfeasible the analysis of a model of flexible pipe with several layers. Moreover, these programs require extensive training periods until the user is able to use them and capable of circumventing their limitations.

The limitations and problems found in multipurpose finite element software have motivated the development of dedicated tools for the design of flexible pipes. *BFLEX*, *UFLEX*, *UmbiliCAD* are examples of commercial software specifically developed to the local analysis of flexible pipes and umbilicals. However, in many cases, the use of specific

software is restricted to large pipe manufacturers, which financed their developments. Besides, even if they were available, these programs would be of limited academic interest, once they would be black box solutions, i.e., the implemented mathematical and numerical models would not be known for commercial and confidential reasons, being possible only application-based developments with these tools.

In this context lies the finite macroelements field, which are finite elements formulated to solve a specific problem. Finite macroelements enable the reduction of computational costs and facilitate the implementation of the model by considering the particularities of the problem. Therefore, finite macroelements possess great potential for local structural analysis of flexible pipes, allowing simulation of pipes that were not possible or that were very costly with conventional elements.

1.2.3 Finite Macroelements Introduction

In his PhD work, (PROVASI, 2013) developed several finite macroelements for modeling layers of a flexible pipe and a full description of these elements is presented in Chapter 2. In order to validate the formulations, (PROVASI, 2013) also implemented these finite macroelements in an analysis tool called *MacroFEM*. For prioritizing the validation process, little attention was initially given to the performance of the implemented code, making this promising analysis tool impractical, due to the excessively high computational demands and simulation times. With the objective of obtaining an efficient tool for structural analysis of flexible pipes, (TONI, F.G., 2014) performed an extension of PROVASI's work. A series of modifications in the code structure were made, implementation bugs were fixed and a new library of solution of linear systems was adopted, which allowed a reduction of up to 95% of the global stiffness matrix assemblage and a reduction of two orders of magnitude in time resolution of the linear systems. This increase in performance allowed the simulation of a more complex, but still simplified, three-layered flexible pipe illustrated in Fig. 1.19:

- one internal tensile armor layer with 16 tendons;
- one external tensile armor layer with 18 tendons;
- one external polymeric sheath.

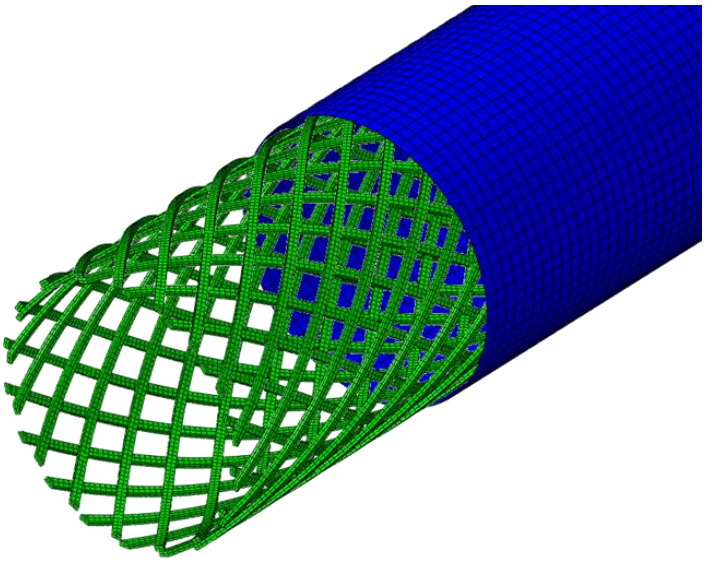

Fig. 1.19 – Simplified pipe simulated by (TONI, F.G., 2014).

It was assumed an isotropic linear elastic constitutive law for all materials. All geometry and material properties from this simplified model are summarized in Table 1.5 and Table 1.6, respectively.

Table 1.5 – Geometry properties.

Property	***Internal Armor***	***External Armor***	***Polymeric Sheath***
Mean Radius (mm)	101.25	105.25	110.75
Cross Section W x H (mm)	8 x 4	8 x 4	---
Number of Tendons	16	18	---
Lay Angle (deg.)	36	-38	---
Thickness (mm)	---	---	7

Source: own authorship.

Table 1.6 – Material properties.

Property	***Internal Armor***	***External Armor***	***Polymeric Sheath***
Young Modulus (MPa)	207,000	207,000	570.88
Poisson Ratio	0.3	0.3	0.45

Source: own authorship.

The armor layers were modeled with helical beam elements, the polymeric sheath with orthotropic cylinder and the rigid connections with bonded and bridge *node-to-node* contact elements. For comparison, the same pipe was modeled in ABAQUS, Fig. 1.20, using 3D linear 8-node solid elements without reduced integration for the tensile armors and 3D quadratic 8-node doubly curved thick shell with reduced integration for the polymeric sheath. Regarding the contact, it was used the *General Contact* method, which determines automatically the contact pairs. In the same Fig. 1.20 are also illustrated the boundaries conditions: an end fully constrained and an axial traction-displacement of 10 mm applied to the other.

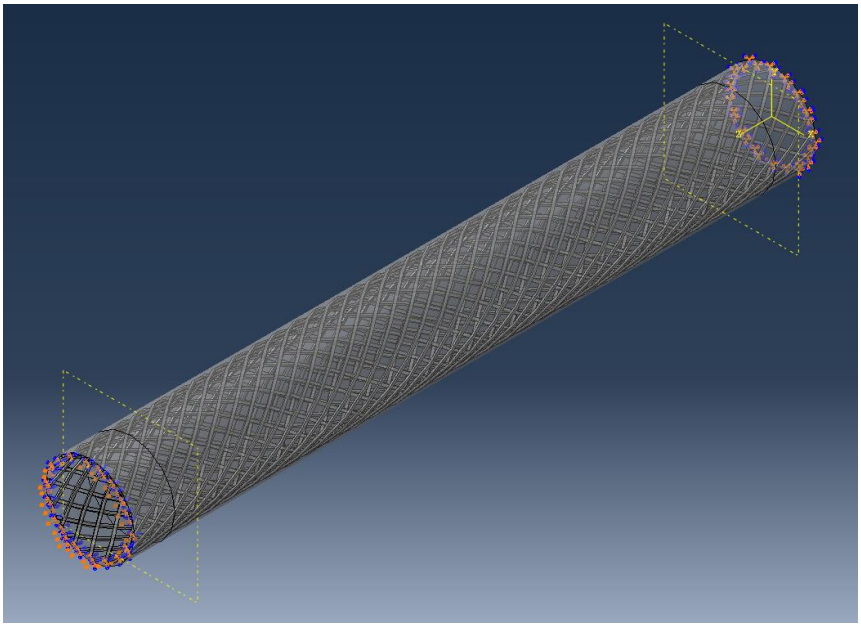

Fig. 1.20 – Boundary conditions applied to the simplified pipe. Source: (TONI, F.G., 2014).

The graph of Fig. 1.21 shows the radial displacements of a tendon of the internal and external tensile armor layers along the axial length of the pipe obtained with both programs.

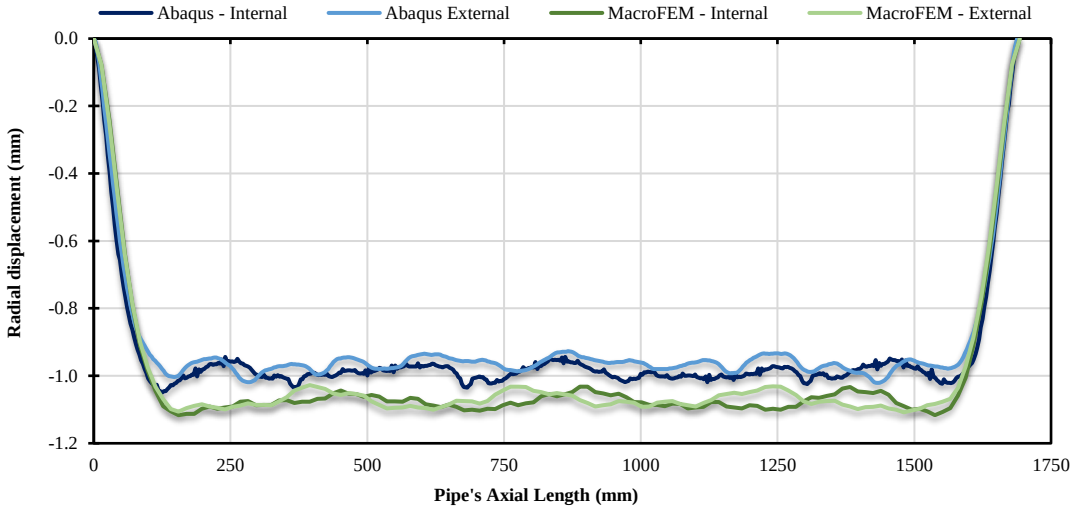

Fig. 1.21 – Radial displacement of a tendon from internal and external tensile armor layers along the axial length of the pipe. Source: (TONI, F.G., 2014).

For a fairer comparison, the same pipe was modeled in ANSYS® with second-order beam elements for the helices (*BEAM189*) and second-order isoparametric solid elements for the external sheath (*SOLID186*). Orientation *keypoints* were used to rotate correctly the cross section of the beam elements. The final mesh is illustrated in Fig. 1.22. The interface between the two armors is modeled with 3D line-to-line contact elements (*CONTA176*) in crossing condition, which enables great results for beam-to-beam contact. The interface between the external armor and the external sheath used 3D line-to-surface contact elements (*CONTA177*). The contact behavior was selected as bonded always for both interfaces.

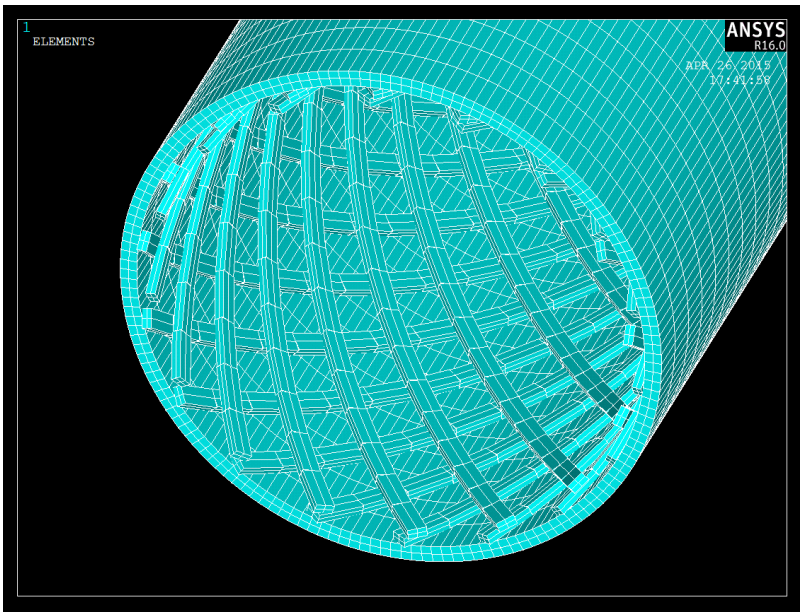

Fig. 1.22 – Element mesh, with active beam section rendering option. Source: own authorship.

The radial, circumferential and axial displacements along a tendon of the internal and external tensile armors obtained with ANSYS and MacroFEM can be seen in Fig. 1.23 to Fig. 1.25, respectively.

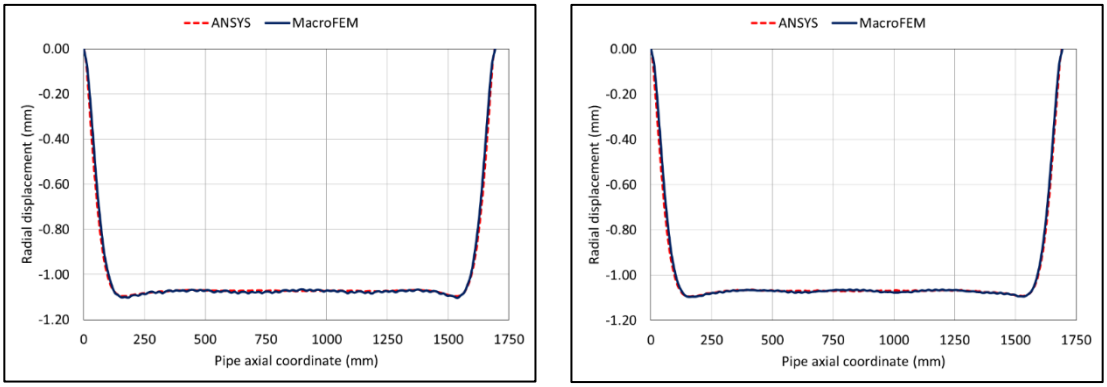

Fig. 1.23 – Radial displacements along the pipe axial coordinate. Left: internal armor; right: external armor. Source: own authorship.

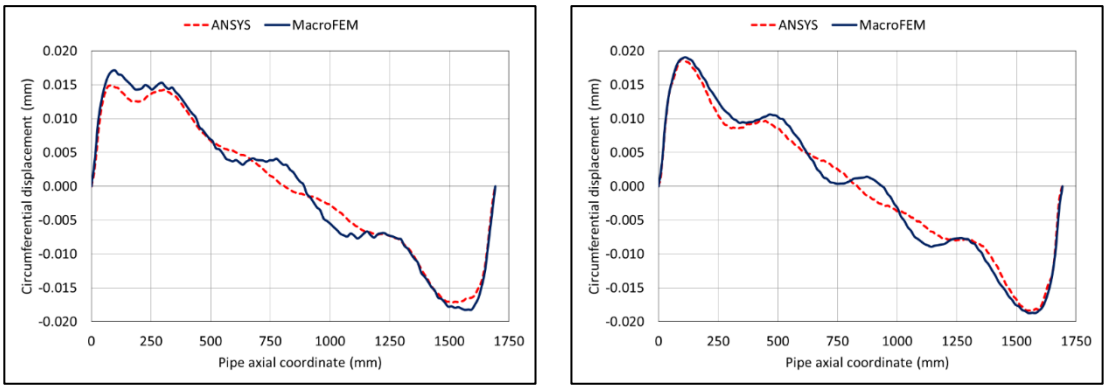

Fig. 1.24 – Circumferential displacements along the pipe axial coordinate. Left: internal armor; right: external armor. Source: own authorship.

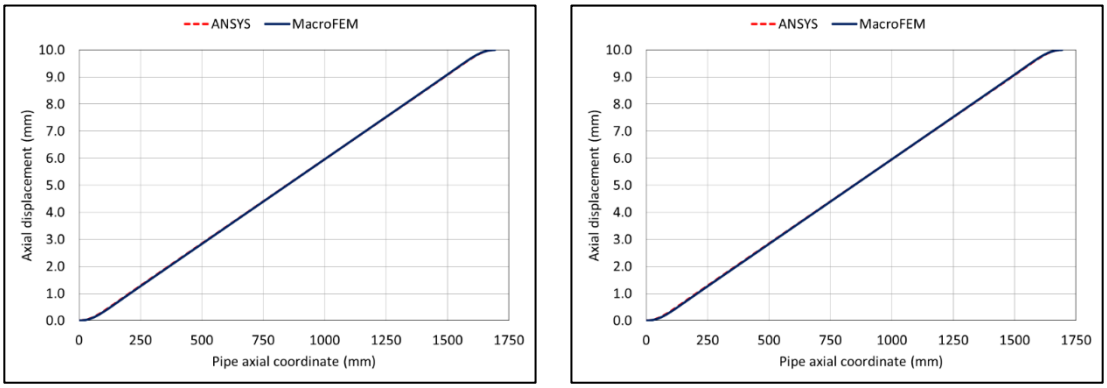

Fig. 1.25 – Axial displacements along the pipe axial coordinate. Left: internal armor; right: external armor. Source: own authorship.

Comparing both implementations, the differences of displacements in radial and axial directions are under 1%. The most noticeable difference appears in circumferential direction, which is an order lower than the radial one, but even so it is still less than 10%. Therefore, it can be concluded that the results obtained with MacroFEM are pretty good when compared with the well-established multipurpose software ANSYS®.

For comparison, a full solid model was also implemented and tested in ANSYS®. In this case, the tensile armors were modeled with second-order SOLID186 elements and all interactions were of the type surface-to-surface (CONTA174/TARGE170). The radial displacements along a tendon from the internal and external armors are in Fig. 1.26 and Fig. 1.27, respectively.

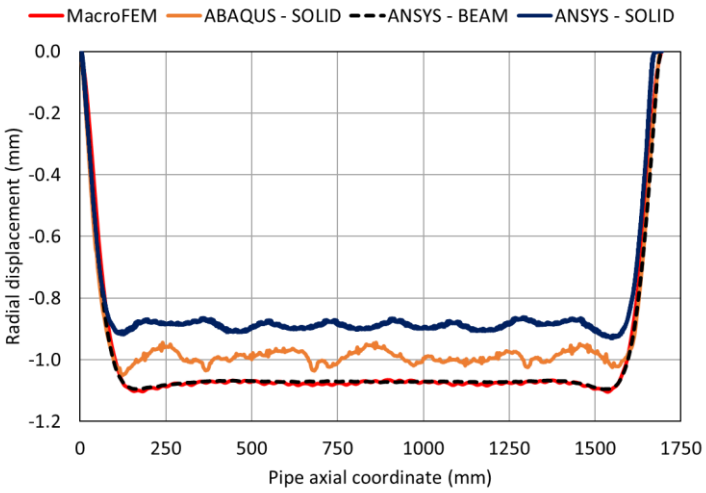

Fig. 1.26 – Radial displacement along the pipe coordinate axis for the internal armor. Source: own authorship.

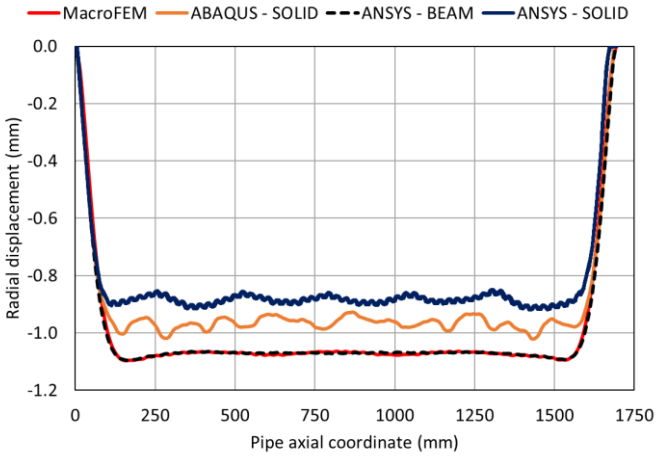

Fig. 1.27 – Radial displacement along the pipe coordinate axis for the external armor. Source: own authorship.

Analyzing these results, it is concluded that the magnitude of the radial displacements from the full solid models are approximately between 10% to 15% lower, what can be explained by modelling differences: the *surface-to-surface* contact type between armors used in ABAQUS® and ANSYS® contributed to rigidly connect hundreds of small contact areas, making the structure stiffer in comparison to the *node-to-node* model in MacroFEM.

Despite the progress in performance and simulation time, MacroFEM still possesses a serious limitation regarding the amount of consumed memory. This is because most of the data, including the global stiffness matrix, is stored in dense matrices and, therefore, memory consumption grows quadratically with the number of degrees of freedom of the

model. To get an idea of the magnitude of this consumption, the simulation of the simplified pipe of Fig. 1.19 required more than 60 GB of RAM, eliminating any possibility of simulating pipes with more layers or components.

The global stiffness matrix is the main responsible for the excessively high memory consumption, once its dimensions are determined by the total number of degrees of freedom of the model. Fig. 1.28 shows the global stiffness matrix from the simplified pipe simulated by (TONI, F.G., 2014), in which the non-zero elements are represented in black. The contact elements occupy the distant positions from the main diagonal, which increases the bandwidth of the matrix. An important conclusion can be made after analyzing this matrix: a drastic reduction in memory consumption can be obtained with the implementation of a convenient data structure.

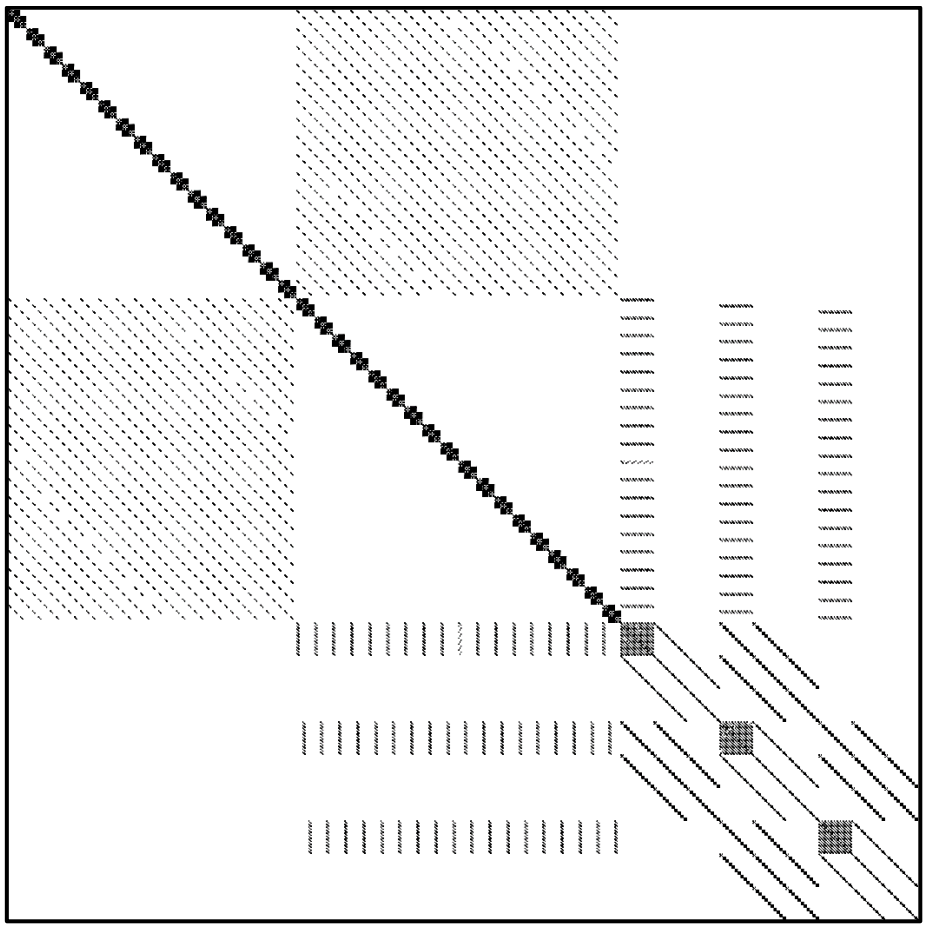

Fig. 1.28 – Sparsity pattern of global stiffness matrix of the simplified pipe simulated by (TONI, F.G., 2014).

1.3 Element-by-Element Method

Given the sparsity pattern of the matrix of Fig. 1.28, a data structure that utilizes sparse matrices could significantly reduce the amount of consumed memory. Sparse matrices are vastly used in the literature and commercial finite element packages. However, a generic implementation of sparse matrices would hardly be competitive with consolidated and well-established existing linear algebra libraries, such as *Pardiso*.

A possible solution to this problem is the development and implementation of customized sparse matrices specifically to the finite elements used to simulate a flexible pipe. By knowing beforehand which the global matrix sparsity pattern is, it is possible to optimize mathematical operators, eliminating the execution of unnecessary numerical operations and develop a specific algorithm of linear system resolution and, this way, achieve an efficient solution for problem. This, nevertheless, would be a very low flexible solution. The data structure of sparse matrices should be parametric enough to meet the combinations and variations in the type and arrangement of elements, what is a very complex task to be determined and predicted. In addition, this solution also would hold the risk of not being able to receive new types of elements in the future, due to probable differences in element stiffness matrices patterns, freezing future improvements of the analysis tool.

In this context, arises the element-by-element method (EBE), which is an alternative to the sparse formulation. In this method, the global stiffness matrix is eliminated and all calculations are performed in an element level. Therefore, the memory consumption increases linearly with the number of elements. The sparse form requires a smaller number of mathematical operations to execute the same algorithm than the EBE form, but with a potential increase in memory depending on the details of the implementation.

However, the main advantage of the EBE formulation consists on the scalability and ease of parallelization of the numerical solution. The larger number of numeric operations required by the EBE, in comparison to the sparse formulation, is rapidly compensated by techniques of parallel programming and element based domain decompositions, taking advantage from clusters and modern processors, which have several processing cores.

The easiness of adding new types of elements also must be highlighted in the EBE method. As the calculations are carried out in a local basis, it is necessary to implement only a matrix-vector multiplication for this element and a scattering method between the local and global degrees-of-freedom.

1.4 Objectives

The sparsity pattern of the global stiffness matrix of Fig. 1.28 illustrates the potential in memory reduction that can be obtained with the application of the element-by-element method. From the point of view of processing capacity, the EBE presents advantages for its scalability and easy of parallelization. A considerable reduction in both memory consumption and processing time when simulating large-scale structural problems of flexible pipes can be achieved with the application of the EBE method, being of great interest for the development of flexible pipes and in practical industry applications.

The implementation of the EBE method imposes some challenges, since it requires a proper data structure for storing and manipulating with efficiency the element stiffness matrices. A suitable indexing system is also needed, since it is responsible for relating the local and global degrees-of-freedom from the elements that comprise the model. Lastly, an EBE iterative algorithm for solving of linear system of equations also must be implemented.

When developing a program for large-scale applications, it is important to employ proper data structures, algorithms and programming languages. In this context, the EBE method already meets these first two requirements. Concerning the third one, it was decided to carry out this work in C++, aiming higher computational performance. Besides that, although third-party linear algebra libraries accelerate the implementation, their use was banned in this work, because they may pose obstructions on the development for being black-box solutions.

Therefore, his work consists in the development and implementation of a new analysis tool that utilizes the EBE method and the finite macroelements developed by (PROVASI, 2013) for large-scale structural analysis of flexible pipes. This involves the development of a proper data structure in C++, an indexing system that relates local and global degrees-of-freedom, and a parallelized EBE algorithm for solution of linear system of equations. By combining memory reduction with parallelized data processing, it is possible to obtain a balanced and efficient analysis tool of large-scale models of flexible pipes.

In Chapter 2, a complete bibliographic review of the finite macro element theory is performed. The deep knowledge of the characteristics of the elements is of fundamental importance for a successful implementation. In sequence, the element-by-element method

is reviewed in detail in Chapter 3, with the purpose of determining the most appropriate algorithms and methods to be implemented for the solution of large-scale problems of flexible pipes. In Chapter 4, the new analysis tool developed in this work for the local analysis of flexible pipes is presented. It is entirely written in C++ and explores parallelism at all stages. Chapter 5 presents the developed containers of the implemented linear algebra library, used for data storage and manipulation, besides mathematical operations, which are widely employed in the computation of the element stiffness matrices, for instance. Aiming high computational performance also in the model generation (what includes geometry and mesh), a fully indexed data structure was developed in conjunction with parallel meshing methods, which are presented in Chapters 6 to 7. The numerical solution of the problem is presented in Chapters 8 and 9, which describes the implementation of the solver and the EBE-PCG algorithm, respectively. A series of numerical results and comparisons are presented in Chapter 10 and, lastly, the final conclusions are made in the Chapter 11.

2 Finite Macroelement Theory

Finite macroelements are finite elements formulated for the solution of a specific problem, considering and taking advantage of its particularities, such as geometry patterns, and thereby, due to a better quality in the representation of the problem, reduce the total number of elements and degrees-of-freedom, besides advantages related to ease of use and implementation.

Over the last years, several finite macroelements were formulated specifically for modeling and solving problems involving flexible pipes, allowing improved computational performance and simpler layer descriptions, among other advantages. These elements are presented throughout this chapter. It is important to note that this work does not aim the development of new finite macroelements, but the application of already existing ones into a convenient data structure, suitable for large-scale models. This work focus on the finite macroelements formulated by PROVASI & MARTINS, given continuity to a research line of the Laboratory of Offshore Mechanics of the University of Sao Paulo (LMO-USP), and they are presented as follows:

- **Orthotropic cylindrical element:** this finite macroelement, fully described in item 2.1, can be used to model polymeric sheaths and equivalent cylindrical layers. An equivalent reinforcement tape layer may also be modeled with this element, due to its orthotropic characteristic. This three-dimensional element has 4 nodes, whose displacements are expanded in a Fourier series.
- **Three-dimensional curved helical beam:** this helically curved beam element, presented in item 2.2, can be used to model the tensile armors of a flexible pipe. Conventional beams would require cross-section rotations, besides a considerably larger number of elements to represent the tensile armor tendons with the same quality.
- **Bridge contact element with nodes of different displacements natures:** this element (item 2.3) can be employed to simulate rigid connections between the first two previously finite macroelements. Due the different nodal natures (cylinder nodes have Fourier expanded displacements), this specific element had to be formulated.

- **Standard contact element with nodes of different displacements natures:** for the same reasons of the bridge contact, this element (2.4) simulates *standard* interactions between the cylindrical and helix elements, which may involve gap formation, tangential sliding and friction.

2.1 Finite Macroelement for Orthotropic Cylindrical Layer Modeling

This finite macroelement, illustrated in Fig. 2.1, was formulated by (PROVASI & MARTINS, 2013-c) and consists in an extension of the isotropic cylinder shown in (COOK, MALKUS, PLESHA, & WITT, 2002). It belongs to a special class of elements known as *Solids of Revolution*, once its formulation differs from the conventional ones by having the nodal displacements expanded into a Fourier series using a cylindrical coordinate system:

$$\begin{cases} u = \sum_{n=0}^{\infty} \bar{u}_n(r,z)\cos n\theta + \sum_{n=0}^{\infty} \bar{\bar{u}}_n(r,z)\sin n\theta \\ v = \sum_{n=0}^{\infty} \bar{v}_n(r,z)\sin n\theta - \sum_{n=0}^{\infty} \bar{\bar{v}}_n(r,z)\cos n\theta \\ w = \sum_{n=0}^{\infty} \bar{w}_n(r,z)\cos n\theta + \sum_{n=0}^{\infty} \bar{\bar{w}}_n(r,z)\sin n\theta \end{cases} \quad \text{Eq. 2.1}$$

where:

- u – is the displacement in the radial direction;
- v – is the displacement in the circumferential direction;
- w – is the displacement in the axial direction;
- θ – denotes the circumferential direction;
- n – is the order of the Fourier series expansion;
- ∞ – represents an infinite sum of the expansion terms. Computationally, it is replaced by a user-defined maximum expansion order, n_{MAX}.
- $\bar{u}_n$, $\bar{\bar{u}}_n$, $\bar{v}_n$, $\bar{\bar{v}}_n$, $\bar{w}_n$ and $\bar{\bar{w}}_n$ – are amplitudes of displacements that may depend on r, z and/or n but are independent of θ. "*Single-barred series describe displacements states that are symmetric with respect to* $\theta = 0$*; double-barred*

series describe displacements states that are antisymmetric with respect to $\theta = 0$" (Cook et al., 2002).

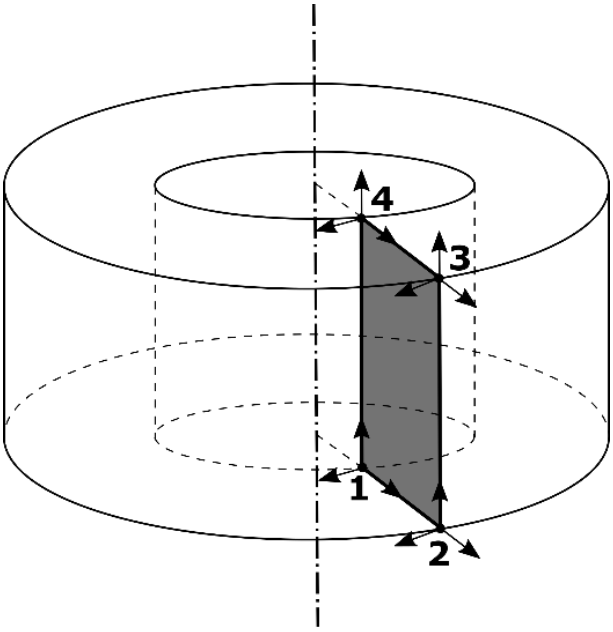

Fig. 2.1 – Four nodes that compose the finite macroelement for orthotropic cylindrical layer modeling. Source: own authorship.

"The advantage of this element is that no division in the θ direction is required, so that instead of solving one large 3D problem, we instead solve a few 2D problems and combine results. Thus, data preparation is simplified and the analysis is much less demanding of computer resources" (COOK, MALKUS, PLESHA, & WITT, 2002).

It was adopted a linear elastic material model. Thus, the infinitesimal deformations are given by:

$$\varepsilon_r = \frac{\partial u}{\partial r} \qquad \text{Eq. 2.2}$$

$$\varepsilon_\theta = \frac{u}{r} + \frac{1}{r}\frac{\partial v}{\partial \theta} \qquad \text{Eq. 2.3}$$

$$\varepsilon_z = \frac{\partial w}{\partial z} \qquad \text{Eq. 2.4}$$

$$\gamma_{r\theta} = \frac{1}{r}\frac{\partial u}{\partial \theta} + \frac{\partial v}{\partial r} - \frac{v}{r} \qquad \text{Eq. 2.5}$$

$$\gamma_{rz} = \frac{\partial u}{\partial z} + \frac{\partial w}{\partial r} \qquad \text{Eq. 2.6}$$

$$\gamma_{\theta z} = \frac{\partial v}{\partial z} + \frac{1}{r}\frac{\partial w}{\partial \theta} \qquad \text{Eq. 2.7}$$

where:

- ε_i – are the strains in the direction given by the subscript i;
- γ_{ij} – are the shear strains in the plane given by the subscript ij;

The strain-displacements relations in cylindrical coordinates can be rewritten in matrix form:

$$\boldsymbol{\varepsilon} = [\boldsymbol{\partial}]\,\boldsymbol{u} \qquad \text{Eq. 2.8}$$

$$\begin{Bmatrix} \varepsilon_r \\ \varepsilon_\theta \\ \varepsilon_z \\ \gamma_{r\theta} \\ \gamma_{rz} \\ \gamma_{\theta z} \end{Bmatrix} = \begin{bmatrix} \frac{\partial}{\partial r} & 0 & 0 \\ \frac{1}{r} & \frac{\partial}{r\partial\theta} & 0 \\ 0 & 0 & \frac{\partial}{\partial z} \\ \frac{\partial}{\partial z} & 0 & \frac{\partial}{\partial r} \\ \frac{\partial}{r\partial\theta} & \left(\frac{\partial}{\partial r} - \frac{1}{r}\right) & 0 \\ 0 & \frac{\partial}{\partial z} & \frac{\partial}{r\partial\theta} \end{bmatrix} \begin{Bmatrix} u \\ v \\ w \end{Bmatrix} \qquad \text{Eq. 2.9}$$

The stress-strains relations are given by:

$$\boldsymbol{\sigma} = \mathbf{E}\,\boldsymbol{\varepsilon} \qquad \text{Eq. 2.10}$$

$$\boldsymbol{\sigma} = [\sigma_r \quad \sigma_\theta \quad \sigma_z \quad \tau_{r\theta} \quad \tau_{rz} \quad \tau_{\theta z}]^T \qquad \text{Eq. 2.11}$$

The most generic expression for the material elasticity matrix, **E**, when θ is a principal material direction is given by:

$$\boldsymbol{E} = \begin{bmatrix} E_{11} & E_{12} & E_{13} & E_{14} & 0 & 0 \\ E_{21} & E_{22} & E_{23} & E_{24} & 0 & 0 \\ E_{31} & E_{32} & E_{33} & E_{34} & 0 & 0 \\ E_{41} & E_{42} & E_{43} & E_{44} & 0 & 0 \\ 0 & 0 & 0 & 0 & E_{55} & E_{56} \\ 0 & 0 & 0 & 0 & E_{65} & E_{66} \end{bmatrix} \quad \text{Eq. 2.12}$$

When r and z are also principal material directions, or when the material is isotropic (i.e. uniform property in all orientations), are the terms $E_{14} = E_{24} = E_{34} = E_{41} = E_{42} = E_{43} = E_{56} = E_{65} = 0$. For an orthotropic material with principal material directions aligned with the principal element directions, the material stiffness matrix can be obtained from the compliance material matrix, $\boldsymbol{C}$:

$$\boldsymbol{E} = \boldsymbol{C}^{-1} \quad \text{Eq. 2.13}$$

$$\boldsymbol{C} = \begin{bmatrix} \frac{1}{E_r} & -\frac{\nu_{\theta r}}{E_\theta} & -\frac{\nu_{zr}}{E_z} & 0 & 0 & 0 \\ -\frac{\nu_{r\theta}}{E_r} & \frac{1}{E_\theta} & -\frac{\nu_{z\theta}}{E_z} & 0 & 0 & 0 \\ -\frac{\nu_{rz}}{E_r} & -\frac{\nu_{\theta z}}{E_\theta} & \frac{1}{E_z} & 0 & 0 & 0 \\ 0 & 0 & 0 & \frac{1}{2G_{r\theta}} & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{1}{2G_{rz}} & 0 \\ 0 & 0 & 0 & 0 & 0 & \frac{1}{2G_{\theta z}} \end{bmatrix} \quad \text{Eq. 2.14}$$

$$\frac{\nu_{r\theta}}{E_r} = \frac{\nu_{\theta r}}{E_\theta} \;;\; \frac{\nu_{rz}}{E_r} = \frac{\nu_{zr}}{E_z} \;;\; \frac{\nu_{\theta z}}{E_\theta} = \frac{\nu_{z\theta}}{E_z} \quad \text{Eq. 2.15}$$

where:

- E_i – is the Young modulus in the i direction;
- ν_{ij} – is the Poisson ration in the ij plane;
- G_{ij} – is the shear modulus in the ij plane.

As can be seen in Fig. 2.1, this element has four nodes, whose displacements are used to interpolate the displacements within this element:

$$\begin{Bmatrix} u \\ v \\ w \end{Bmatrix} = [\boldsymbol{N}_1 \quad \boldsymbol{N}_2 \quad \boldsymbol{N}_3 \quad \boldsymbol{N}_4]\, \boldsymbol{A}\, \{\boldsymbol{d}\} \qquad \text{Eq. 2.16}$$

with:

$$\boldsymbol{N}_i = \begin{bmatrix} N_i & 0 & 0 \\ 0 & N_i & 0 \\ 0 & 0 & N_i \end{bmatrix}, \quad i = 1, \dots, 4 \qquad \text{Eq. 2.17}$$

$$\boldsymbol{A} = \begin{cases} \begin{bmatrix} \cos n\theta & 0 & 0 \\ 0 & \sin n\theta & 0 \\ 0 & 0 & \cos n\theta \end{bmatrix} & for\ single\text{-}barred \\ \begin{bmatrix} \sin n\theta & 0 & 0 \\ 0 & -\cos n\theta & 0 \\ 0 & 0 & \sin n\theta \end{bmatrix} & for\ double\text{-}barred \end{cases} \qquad \text{Eq. 2.18}$$

$$\{\boldsymbol{d}\} = \{\overline{\boldsymbol{d}}\} + \{\overline{\overline{\boldsymbol{d}}}\} \qquad \text{Eq. 2.19}$$

$$\{\overline{\boldsymbol{d}}\} = \sum_{n=0}^{n_{MAX}} \{\overline{\boldsymbol{d}}^n\} \qquad \text{Eq. 2.20}$$

$$\{\overline{\boldsymbol{d}}^n\} = \{\bar{u}_1^n \quad \bar{v}_1^n \quad \bar{w}_1^n \quad \cdots \quad \bar{u}_4^n \quad \bar{v}_4^n \quad \bar{w}_4^n\}^T \qquad \text{Eq. 2.21}$$

$$\{\overline{\overline{\boldsymbol{d}}}\} = \sum_{n=0}^{n_{MAX}} \{\overline{\overline{\boldsymbol{d}}}^n\} \qquad \text{Eq. 2.22}$$

$$\{\overline{\overline{\boldsymbol{d}}}^n\} = \{\bar{\bar{u}}_1^n \quad \bar{\bar{v}}_1^n \quad \bar{\bar{w}}_1^n \quad \cdots \quad \bar{\bar{u}}_4^n \quad \bar{\bar{v}}_4^n \quad \bar{\bar{w}}_4^n\}^T \qquad \text{Eq. 2.23}$$

where:

- i – refers to the nodal index, varying from 1 to 4;
- n – refers to the expansion order;
- n_{MAX} – is the maximum adopted expansion order;
- N_i – are the element shape functions;
- $\boldsymbol{d}$ **–** is the vector of nodal displacements;
- $\overline{\boldsymbol{d}}$ – is the vector of nodal single-barred displacement terms;

- $\overline{\overline{\boldsymbol{d}}}$ – is the vector of nodal double-barred displacement terms;
- $\overline{\boldsymbol{d}}^n$ – is the vector of nodal single-barred displacement terms for the order n;
- $\overline{\overline{\boldsymbol{d}}}^n$ – is the vector of nodal double-barred displacement terms for the order n.

The shape functions N_i are given by:

$$N_1 = \frac{\left(\frac{t}{2} - (r - R_m)\right)\left(\frac{L}{2} - z\right)}{t\,L} \quad \text{Eq. 2.24}$$

$$N_2 = \frac{\left(\frac{t}{2} + (r - R_m)\right)\left(\frac{L}{2} - z\right)}{t\,L} \quad \text{Eq. 2.25}$$

$$N_3 = \frac{\left(\frac{t}{2} + (r - R_m)\right)\left(\frac{L}{2} + z\right)}{t\,L} \quad \text{Eq. 2.26}$$

$$N_4 = \frac{\left(\frac{t}{2} - (r - R_m)\right)\left(\frac{L}{2} + z\right)}{t\,L} \quad \text{Eq. 2.27}$$

where:

- t – is the cylinder thickness (measured in radial direction);
- L – is the element length (measured in axial direction).
- R_m – is the mean radius of the element;
- z – is the mean axial coordinate of the element.

The displacements can be broken into the sum of the single-barred and double-barred terms of the Fourier expansion series:

$$\begin{Bmatrix} u \\ v \\ w \end{Bmatrix} = \sum_{n=0}^{n_{MAX}} \begin{Bmatrix} u^n \\ v^n \\ w^n \end{Bmatrix} = \sum_{n=0}^{n_{MAX}} \begin{Bmatrix} \bar{u}^n \\ \bar{v}^n \\ \bar{w}^n \end{Bmatrix} + \sum_{n=0}^{n_{MAX}} \begin{Bmatrix} \bar{\bar{u}}^n \\ \bar{\bar{v}}^n \\ \bar{\bar{w}}^n \end{Bmatrix} \quad \text{Eq. 2.28}$$

$$\begin{aligned} u^n &= \bar{u}^n + \bar{\bar{u}}^n \\ v^n &= \bar{v}^n + \bar{\bar{v}}^n \\ w^n &= \bar{w}^n + \bar{\bar{w}}^n \end{aligned} \quad \text{Eq. 2.29}$$

For each expansion order n (or harmonic number):

$$\begin{Bmatrix} \bar{u}^n \\ \bar{v}^n \\ \bar{w}^n \end{Bmatrix} = [\boldsymbol{N}_1 \quad \boldsymbol{N}_2 \quad \boldsymbol{N}_3 \quad \boldsymbol{N}_4] \begin{bmatrix} \cos n\theta & 0 & 0 \\ 0 & \sin n\theta & 0 \\ 0 & 0 & \cos n\theta \end{bmatrix} \{\bar{\boldsymbol{d}}^n\} \qquad \text{Eq. 2.30}$$

$$\begin{Bmatrix} \bar{\bar{u}}^n \\ \bar{\bar{v}}^n \\ \bar{\bar{w}}^n \end{Bmatrix} = [\boldsymbol{N}_1 \quad \boldsymbol{N}_2 \quad \boldsymbol{N}_3 \quad \boldsymbol{N}_4] \begin{bmatrix} \sin n\theta & 0 & 0 \\ 0 & -\cos n\theta & 0 \\ 0 & 0 & \sin n\theta \end{bmatrix} \{\bar{\bar{\boldsymbol{d}}}^n\} \qquad \text{Eq. 2.31}$$

For ease in notation, the shape functions are combined with the harmonic matrix for each case:

$$\bar{\boldsymbol{N}}_i = \begin{bmatrix} N_i \cos n\theta & 0 & 0 \\ 0 & N_i \sin n\theta & 0 \\ 0 & 0 & N_i \cos n\theta \end{bmatrix} \qquad \text{Eq. 2.32}$$

$$\bar{\bar{\boldsymbol{N}}}_i = \begin{bmatrix} N_i \sin n\theta & 0 & 0 \\ 0 & -N_i \cos n\theta & 0 \\ 0 & 0 & N_i \sin n\theta \end{bmatrix} \qquad \text{Eq. 2.33}$$

Then,

$$\begin{Bmatrix} \bar{u}^n \\ \bar{v}^n \\ \bar{w}^n \end{Bmatrix} = [\bar{\boldsymbol{N}}_1 \quad \bar{\boldsymbol{N}}_2 \quad \bar{\boldsymbol{N}}_3 \quad \bar{\boldsymbol{N}}_4] \{\bar{\boldsymbol{d}}^n\} \qquad \text{Eq. 2.34}$$

$$\begin{Bmatrix} \bar{\bar{u}}^n \\ \bar{\bar{v}}^n \\ \bar{\bar{w}}^n \end{Bmatrix} = [\bar{\bar{\boldsymbol{N}}}_1 \quad \bar{\bar{\boldsymbol{N}}}_2 \quad \bar{\bar{\boldsymbol{N}}}_3 \quad \bar{\bar{\boldsymbol{N}}}_4] \{\bar{\bar{\boldsymbol{d}}}^n\} \qquad \text{Eq. 2.35}$$

Applying Eq. 2.34 and Eq. 2.35 into Eq. 2.8, the strain can be expressed in function of the nodal displacements:

$$\boldsymbol{\varepsilon} = \boldsymbol{\partial}[\bar{\boldsymbol{N}}_1 \quad \bar{\boldsymbol{N}}_2 \quad \bar{\boldsymbol{N}}_3 \quad \bar{\boldsymbol{N}}_4] \{\bar{\boldsymbol{d}}\} + \boldsymbol{\partial}[\bar{\bar{\boldsymbol{N}}}_1 \quad \bar{\bar{\boldsymbol{N}}}_2 \quad \bar{\bar{\boldsymbol{N}}}_3 \quad \bar{\bar{\boldsymbol{N}}}_4] \{\bar{\bar{\boldsymbol{d}}}\} \qquad \text{Eq. 2.36}$$

The strain-displacement matrix can be defined as $\boldsymbol{B}_i = \boldsymbol{\partial}\, \boldsymbol{N}_i$. It will be shown the calculations of the single-barred terms, but the procedure is analogous to the double-

barred terms. For the order n, the strain-displacement matrix for the single-barred terms is given by:

$$\begin{Bmatrix} \varepsilon_r^n \\ \varepsilon_\theta^n \\ \varepsilon_z^n \\ \gamma_{r\theta}^n \\ \gamma_{rz}^n \\ \gamma_{\theta z}^n \end{Bmatrix} = \begin{bmatrix} N_{1,r}\cos n\theta & 0 & 0 & \dots \\ \frac{N_1}{r}\cos n\theta & \frac{nN_1}{r}\cos n\theta & 0 & \dots \\ 0 & 0 & N_{1,z}\cos n\theta & \dots \\ -\frac{nN_1}{r}\sin n\theta & \left(N_{1,r}-\frac{N_1}{r}\right)\sin n\theta & 0 & \dots \\ N_{1,z}\cos n\theta & 0 & N_{1,r}\cos n\theta & \dots \\ 0 & N_{1,z}\sin n\theta & -\frac{nN_1}{r}\sin n\theta & \dots \end{bmatrix} \begin{Bmatrix} \bar{u}_1^n \\ \bar{v}_1^n \\ \bar{w}_1^n \\ \vdots \\ \bar{u}_4^n \\ \bar{v}_4^n \\ \bar{w}_4^n \end{Bmatrix} \quad \text{Eq. 2.37}$$

$$[\boldsymbol{B}^n] = \begin{bmatrix} N_{1,r}\cos n\theta & 0 & 0 & \dots \\ \frac{N_1}{r}\cos n\theta & \frac{nN_1}{r}\cos n\theta & 0 & \dots \\ 0 & 0 & N_{1,z}\cos n\theta & \dots \\ -\frac{nN_1}{r}\sin n\theta & \left(N_{1,r}-\frac{N_1}{r}\right)\sin n\theta & 0 & \dots \\ N_{1,z}\cos n\theta & 0 & N_{1,r}\cos n\theta & \dots \\ 0 & N_{1,z}\sin n\theta & -\frac{nN_1}{r}\sin n\theta & \dots \end{bmatrix} \quad \text{Eq. 2.38}$$

Considering all expansion orders,

$$[\boldsymbol{B}] = [\boldsymbol{B}^0 \quad \boldsymbol{B}^1 \quad \dots \quad \boldsymbol{B}^{n_{MAX}}] \quad \text{Eq. 2.39}$$

For the single-barred terms, the element stiffness matrix is given by:

$$\boldsymbol{K} = \int_V \boldsymbol{B}^T \boldsymbol{E} \boldsymbol{B}\ dV = \int_{-\pi}^{\pi}\int \boldsymbol{B}^T \boldsymbol{E} \boldsymbol{B}\ d\theta dA \quad \text{Eq. 2.40}$$

$$\boldsymbol{K} = \begin{bmatrix} \boldsymbol{K}^{00} & \boldsymbol{K}^{01} & \cdots & \boldsymbol{K}^{0N} \\ \boldsymbol{K}^{10} & \boldsymbol{K}^{11} & \cdots & \boldsymbol{K}^{1N} \\ \vdots & \vdots & \ddots & \vdots \\ \boldsymbol{K}^{N0} & \boldsymbol{K}^{N1} & \cdots & \boldsymbol{K}^{NN} \end{bmatrix} \quad \text{Eq. 2.41}$$

"*Let there be J nodes per element and M harmonics included. Then the integrand matrix $\boldsymbol{B}^T\boldsymbol{E}\boldsymbol{B}$ is full and of size $3JM$ by $3JM$. It is composed of an M by M array of $3J$ by $3J$ submatrices. Off-diagonal submatrices contain $(\sin m\theta \sin n\theta)$ or $(\cos m\theta \cos n\theta)$*

in every term, where m and n are different integers that represent different harmonics. With limits -π to +π, integrals of these terms are zero. We are left with only M submatrices on the diagonal, which means that different Fourier harmonics are uncoupled. Each on-diagonal submatrix is $3J$ by $3J$ and contains $(sin^2 n\theta)$ or $(cos^2 n\theta)$ in every term. With limits -π to +π, $(sin^2 n\theta)$ and $(cos^2 n\theta)$ each integrate to π (or to 2π for $cos^2 n\theta$ when $n = 0$)" (Cook et al., 2002).

$$\begin{bmatrix} \boldsymbol{K}^0 & 0 & 0 & 0 \\ 0 & \boldsymbol{K}^1 & 0 & 0 \\ 0 & 0 & \boldsymbol{K}^2 & 0 \\ 0 & 0 & 0 & \ddots \end{bmatrix} \begin{Bmatrix} \bar{\boldsymbol{d}}_0 \\ \bar{\boldsymbol{d}}_1 \\ \bar{\boldsymbol{d}}_2 \\ \vdots \end{Bmatrix} = \begin{Bmatrix} \bar{\boldsymbol{R}}_0 \\ \bar{\boldsymbol{R}}_1 \\ \bar{\boldsymbol{R}}_2 \\ \vdots \end{Bmatrix} \qquad \text{Eq. 2.42}$$

"*If the double-barred series is used rather than the single barred series, one finds that $\cos n\theta$ and $\sin n\theta$ are interchanged in Eq. 2.32, Eq. 2.37 and Eq. 2.38. Also, algebraic signs are reversed in the fourth and sixth rows of the matrix from Eq. 2.38. However, for $n > 0$, submatrices $[\boldsymbol{K}_n]$ turn out to be the same as those produced by the single-barred series. This convenience is the motivation for the arbitrarily chosen negative sign in Eq. 2.1*" (Cook et al., 2002).

(PROVASI & MARTINS, 2013-c) have further expanded Eq. 2.42 and obtained analytical solutions for the stiffness matrix terms. These expressions are omitted here, but can be found in the original work.

2.2 Three-Dimensional Curved Helical Beam Element

(PROVASI & MARTINS, 2014) formulated a three-dimensional curved beam element that considers the effects of curvature and tortuosity, making it ideal for modeling tensile armor tendons of a flexible pipe.

The displacements are described in a cylindrical coordinate system, making easier its integration with other types of finite elements. The beam cross-section system is chosen to be coincident with the Fernet system, this requires a coordinate system rotation to the cylindrical one. The strain-displacement relations are given by:

$$\varepsilon_z = \frac{\partial w(s)}{\partial s} - k\, u(s) \qquad \text{Eq. 2.43}$$

$$\omega_x = \frac{\partial \varphi_x(s)}{\partial s} - \tau\, \varphi_y(s) + k\varphi_z(s) \qquad \text{Eq. 2.44}$$

$$\omega_y = \frac{\partial \varphi_y(s)}{\partial s} + \tau\, \varphi_x(s) \qquad \text{Eq. 2.45}$$

$$\omega_z = \frac{\partial \varphi_z(s)}{\partial s} - k\varphi_x(s) \qquad \text{Eq. 2.46}$$

where:

- s – is the curvilinear coordinate;
- ε_z – is the axial strain or the strain in z direction;
- ω_x, ω_y and ω_z – are the angular strain around the x (normal), y (binormal) and z (tangent) axis, respectively;
- φ_x, φ_y and φ_z – are the angles around the x, y and z axis, respectively;
- w – is the displacement in z direction;
- τ – is the initial tortuosity;
- k – is the initial curvature.

Differently from conventional curved beam elements, the curvature and tortuosity are not variables, but input parameters, which need to be calculated previously. Their final values are given by variation of angles, calculated from the nodal displacements.

The formulation of this element considers the following hypothesis:

- Small displacements and deformations;
- No cross-section warp;
- Linear elastic isotropic material.
- Linear variation from the variables ε_z and ω_z within the element, in order to avoid the shear locking phenomenon.

The displacements for normal (u) and bi-normal (v) directions are given by fifth order polynomials:

$$u(s) = a_0 + a_1 s + a_2 s^2 + a_3 s^3 + a_4 s^4 + a_5 s^5 \qquad \text{Eq. 2.47}$$

$$v(s) = b_0 + b_1 s + b_2 s^2 + b_3 s^3 + b_4 s^4 + b_5 s^5 \quad \text{Eq. 2.48}$$

The following expressions are also valid:

$$\varphi_x = -\frac{\partial v}{\partial s} - \tau\, u \quad \text{Eq. 2.49}$$

$$\varphi_y = \frac{\partial u}{\partial s} - \tau\, v + kw \quad \text{Eq. 2.50}$$

Eq. 2.43 can be manipulated to obtain the axial displacement, remembering that it was assumed a linear behavior to ε_z:

$$w(s) = \int (\varepsilon_z + ku)\, ds \quad \text{Eq. 2.51}$$

$$w(s) = a_6 + (a_7 + ka_0)s + \left(\frac{a_8}{2} + k\frac{a_1}{2}\right)s^2 + k\frac{a_2}{3}s^3 + k\frac{a_3}{4}s^4 + k\frac{a_4}{5}s^5 + k\frac{a_5}{6}s^6 \quad \text{Eq. 2.52}$$

For being a locking-free element, the displacement interpolation functions must recover the inextensible bending of the curved beam in Eq. 2.43:

$$\varepsilon_z = \frac{\partial w(s)}{\partial s} - k\, u(s) \quad \text{Eq. 2.53}$$

$$\varepsilon_z = (a_7 + ka_0) + 2\left(\frac{a_8}{2} + k\frac{a_1}{2}\right)s + ka_2 s^2 + ka_3 s^3 + ka_4 s^4 + ka_5 s^5 \quad \text{Eq. 2.54}$$

φ_z can be calculated from Eq. 2.44:

$$\varphi_z = \int \left(\omega_z - k\frac{\partial v}{\partial s} - k\tau u\right) ds \quad \text{Eq. 2.55}$$

$$\varphi_z(s) = b_6 + (b_7 - k\,b_1 - k\tau a_0)s + \left(\frac{b_8}{2} - kb_2 - \frac{k\tau}{2}a_1\right)s^2 - k\left(b_3 + \frac{\tau}{3}a_2\right)s^3 - k\left(b_4 + \frac{\tau}{4}a_3\right)s^4 - k\left(b_5 + \frac{\tau}{5}a_4\right)s^5 - \frac{k\tau}{6}a_5 s^6 \quad \text{Eq. 2.56}$$

φ_x and φ_y are calculated by:

$$\varphi_x = -\frac{\partial v}{\partial s} - \tau u \quad \text{Eq. 2.57}$$

$$\varphi_x(s) = -(\tau a_0 + b_1) - (\tau a_1 + 2b_2)s - (\tau a_2 + 3b_3)s^2 - (\tau a_3 + 4b_4)s^3 - (\tau a_4 + 5b_5)s^4 - \tau a_5 s^5 \quad \text{Eq. 2.58}$$

$$\varphi_y = \frac{\partial u}{\partial s} - \tau v + kw \quad \text{Eq. 2.59}$$

$$\varphi_y(s) = (a_1 - \tau b_0 + k a_6) + \left(2a_2 - \tau b_1 + k(a_7 + k a_0)\right)s + \left(3a_3 - \tau b_2 + k\left(a_8 + k\frac{a_1}{2}\right)\right)s^2 + \left(4a_4 - \tau b_3 + k^2\frac{a_2}{3}\right)s^3 + \left(5a_5 - \tau b_4 k^2\frac{a_3}{4}\right)s^4 + \left(-\tau b_5 + k^2\frac{a_4}{5}\right)s^5 + k^2\frac{a_5}{6}s^6 \quad \text{Eq. 2.60}$$

Three nodes, each one with 6 degrees-of-freedom, are enough to obtain all constants ($\boldsymbol{q}^T = [a_0 \quad \cdots \quad a_8 \quad b_0 \quad \cdots \quad b_8]$). The vector of nodal displacements is written by:

$$\boldsymbol{u}_{nodal}{}^T = [\boldsymbol{u}_1^T \quad \boldsymbol{u}_2^T \quad \boldsymbol{u}_2^T] \quad \text{Eq. 2.61}$$

$$\boldsymbol{u}_i^T = \begin{bmatrix} u^i & v^i & w^i & \varphi_x^i & \varphi_y^i & \varphi_z^i \end{bmatrix} \quad \text{for } i = 1, \dots, 3 \quad \text{Eq. 2.62}$$

A relation between the nodal displacements and the constants $\boldsymbol{q}$ can be written as:

$$\boldsymbol{u}_{nodal} = \boldsymbol{C}\boldsymbol{q} \quad \text{Eq. 2.63}$$

$$\boldsymbol{q} = \begin{bmatrix} \boldsymbol{C_1} & \boldsymbol{C_2} & \boldsymbol{C_3} \\ \boldsymbol{C_4} & \boldsymbol{C_5} & \boldsymbol{C_6} \\ \boldsymbol{C_7} & \boldsymbol{C_8} & \boldsymbol{C_9} \end{bmatrix} \qquad \text{Eq. 2.64}$$

Each of the terms of $\boldsymbol{q}$ ($\boldsymbol{C_1}$,…, $\boldsymbol{C_9}$) is a 6 by 6 matrix:

$$\boldsymbol{C_1} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & \tau & -\kappa & 0 & 1 & 0 \\ -\frac{23}{L^2} & -6\frac{\tau}{L} & 6\frac{\kappa}{L} & 0 & -\frac{6}{L} & 0 \\ \frac{66}{L^3} & 13\frac{\tau}{L^2} & -13\frac{\kappa}{L^2} & 0 & \frac{13}{L^2} & 0 \\ -\frac{68}{L^4} & -12\frac{\tau}{L^3} & 12\frac{\kappa}{L^3} & 0 & -\frac{12}{L^3} & 0 \\ \frac{24}{L^5} & 4\frac{\tau}{L^4} & -4\frac{\kappa}{L^4} & 0 & \frac{4}{L^4} & 0 \end{bmatrix} \qquad \text{Eq. 2.65}$$

$$\boldsymbol{C_2} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ \frac{16}{L^2} & -8\frac{\tau}{L} & 8\frac{\kappa}{L} & 0 & -\frac{8}{L} & 0 \\ -\frac{32}{L^3} & 32\frac{\tau}{L^2} & -32\frac{\kappa}{L^2} & 0 & \frac{32}{L^2} & 0 \\ \frac{16}{L^4} & -40\frac{\tau}{L^3} & 40\frac{\kappa}{L^3} & 0 & -\frac{40}{L^3} & 0 \\ 0 & 16\frac{\tau}{L^4} & -16\frac{\kappa}{L^4} & 0 & \frac{16}{L^4} & 0 \end{bmatrix} \qquad \text{Eq. 2.66}$$

$$\boldsymbol{C_3} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ \frac{7}{L^2} & -\frac{\tau}{L} & \frac{\kappa}{L} & 0 & -\frac{1}{L} & 0 \\ -\frac{34}{L^3} & 5\frac{\tau}{L^2} & -5\frac{\kappa}{L^2} & 0 & \frac{5}{L^2} & 0 \\ \frac{52}{L^4} & -8\frac{\tau}{L^3} & 8\frac{\kappa}{L^3} & 0 & -\frac{8}{L^3} & 0 \\ -\frac{24}{L^5} & 4\frac{\tau}{L^4} & -4\frac{\kappa}{L^4} & 0 & \frac{4}{L^4} & 0 \end{bmatrix} \qquad \text{Eq. 2.67}$$

$$\boldsymbol{C_4}=\begin{bmatrix}0 & 0 & 1 & 0 & 0 & 0\\ \frac{47}{120}\kappa & -\frac{3}{80}\kappa L\tau & \frac{3(-80+\kappa^2L^2)}{80L} & 0 & -\frac{3\kappa L}{80} & 0\\ \frac{3}{8}\frac{\kappa}{L} & \frac{25}{48}\kappa\tau & -\frac{-96+25\,\kappa^2L^2}{48L^2} & 0 & \frac{25\kappa}{48} & 0\\ 0 & 1 & 0 & 0 & 0 & 0\\ -\tau & 0 & 0 & -1 & 0 & 0\\ 6\frac{\tau}{L} & -\frac{23}{\mathrm{L}^2} & 0 & \frac{6}{\mathrm{L}} & 0 & 0\end{bmatrix} \quad \text{Eq. 2.68}$$

$$\boldsymbol{C_5}=\begin{bmatrix}0 & 0 & 0 & 0 & 0 & 0\\ -\frac{8}{15}\kappa & \frac{\kappa L\tau}{6} & \frac{24-\kappa^2L^2}{6L} & 0 & \frac{\kappa L}{6} & 0\\ 0 & -\frac{\kappa\,\tau}{6} & \frac{-24+\kappa^2L^2}{6L^2} & 0 & -\frac{\kappa}{6} & 0\\ 0 & 0 & 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 0 & 0 & 0\\ 8\,\frac{\tau}{L} & \frac{16}{L^2} & 0 & \frac{8}{\mathrm{L}} & 0 & 0\end{bmatrix} \quad \text{Eq. 2.69}$$

$$\boldsymbol{C_6}=\begin{bmatrix}0 & 0 & 0 & 0 & 0 & 0\\ \frac{17}{120}\kappa & -\frac{\kappa L\tau}{240} & \frac{-240+\kappa^2L^2}{240L} & 0 & -\frac{\kappa L}{240} & 0\\ -\frac{3}{8}\frac{\kappa}{L} & \frac{\kappa\tau}{48} & \frac{96-\kappa^2L^2}{48L^2} & 0 & \frac{\kappa}{48} & 0\\ 0 & 0 & 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 0 & 0 & 0\\ \frac{\tau}{L} & \frac{7}{L^2} & 0 & \frac{1}{L} & 0 & 0\end{bmatrix} \quad \text{Eq. 2.70}$$

$$\boldsymbol{C_7}=\begin{bmatrix}-\frac{13\tau}{L^2} & \frac{66}{L^3} & 0 & -\frac{13}{L^2} & 0 & 0\\ \frac{12\tau}{L^3} & -\frac{68}{L^4} & 0 & \frac{12}{L^3} & 0 & 0\\ -\frac{4\tau}{L^4} & \frac{24}{L^5} & 0 & -\frac{4}{L^4} & 0 & 0\\ 0 & 0 & 0 & 0 & 0 & 1\\ \frac{73\kappa\tau}{120} & \frac{3\kappa(\tau^2L^2-80)}{80L} & -\frac{3\kappa^2L\tau}{80} & \kappa & \frac{3\kappa L\tau}{80} & -\frac{3}{\mathrm{L}}\\ -\frac{51\kappa\tau}{8L} & -\frac{25\kappa(\tau^2L^2-48)}{48L^2} & \frac{25\kappa^2\tau}{48} & -\frac{6\kappa}{L} & -\frac{25\kappa\tau}{48} & \frac{2}{L^2}\end{bmatrix} \quad \text{Eq. 2.71}$$

$$\boldsymbol{C_8} = \begin{bmatrix} -32\frac{\tau}{L^2} & -\frac{32}{L^3} & 0 & -\frac{32}{L^2} & 0 & 0 \\ 40\frac{\tau}{L^3} & \frac{16}{L^4} & 0 & \frac{40}{L^3} & 0 & 0 \\ -16\frac{\tau}{L^4} & 0 & 0 & -\frac{16}{L^4} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ \frac{8}{15}\kappa\tau & -\frac{\kappa(\tau^2L^2-24)}{6L} & \frac{1}{6}\kappa^2L\tau & 0 & -\frac{\kappa\,L\tau}{6} & \frac{4}{\mathrm{L}} \\ -8\frac{\kappa\tau}{L} & \frac{\kappa(\tau^2L^2-120)}{6L^2} & -\frac{1}{6}\kappa^2\tau & \frac{-8\kappa}{L} & \frac{\kappa\,\tau}{6} & -\frac{4}{L^2} \end{bmatrix} \quad \text{Eq. 2.72}$$

$$\boldsymbol{C_9} = \begin{bmatrix} -5\,\frac{\tau}{L^2} & -\frac{34}{L^3} & 0 & -\frac{5}{L^2} & 0 & 0 \\ 8\,\frac{\tau}{L^3} & \frac{52}{L^4} & 0 & \frac{8}{L^3} & 0 & 0 \\ -4\,\frac{\tau}{L^4} & -\frac{24}{L^5} & 0 & -\frac{4}{L^4} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ -\frac{17}{120}\,\kappa\,\tau & \frac{\kappa\,(\tau^2L^2-240)}{240L} & -\frac{\kappa^2L\tau}{240} & 0 & \frac{\kappa\,L\tau}{240} & -\frac{1}{L} \\ -\frac{5}{8}\frac{\kappa\,\tau}{L} & -\frac{\kappa\,(\tau^2L^2+240)}{48L^2} & \frac{\kappa^2\tau}{48} & -\frac{\kappa}{L} & -\frac{\kappa\,\tau}{48} & \frac{2}{L^2} \end{bmatrix} \quad \text{Eq. 2.73}$$

The displacements of the element can be defined through the interpolation of the nodal ones:

$$\boldsymbol{u} = \boldsymbol{A}\boldsymbol{q} = \boldsymbol{A}\boldsymbol{C}\boldsymbol{u}_{nodal} \quad \text{Eq. 2.74}$$

where:

- $\boldsymbol{u}^T = [u \quad v \quad w \quad \varphi_x \quad \varphi_y \quad \varphi_z]$;
- $\boldsymbol{A} = [\boldsymbol{A}_1 \quad \boldsymbol{A}_2 \quad \boldsymbol{A}_3]$.

$$\boldsymbol{A_1} = \begin{bmatrix} 1 & s & s^2 & s^3 & s^4 & s^5 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{\kappa\, s^3}{3} & \frac{\kappa\, s^4}{4} & \frac{\kappa\, s^5}{5} & \frac{\kappa\, s^6}{6} \\ -\tau & -s\tau & -s^2\tau & -s^3\tau & -\tau\, s^4 & -\tau\, s^5 \\ 0 & 1 & 2s + \frac{\kappa^2 s^3}{3} & 3s^2 + \frac{\kappa^2 s^4}{4} & 4s^3 + \frac{\kappa^2 s^5}{5} & 5s^4 + \frac{\kappa^2 s^6}{6} \\ 0 & 0 & -\frac{\kappa\, \tau\, s^3}{3} & -\frac{\kappa\, \tau\, s^4}{4} & -\frac{\kappa\, \tau\, s^5}{5} & -\frac{\kappa\, \tau\, s^6}{6} \end{bmatrix} \quad \text{Eq. 2.75}$$

$$\boldsymbol{A_2} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & s & s^2 \\ 1 & s & s^2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & -2\, s \\ \kappa & \kappa\, s & \kappa\, s^2 & -\tau & -s\tau & -s^2\tau \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \quad \text{Eq. 2.76}$$

$$\boldsymbol{A_3} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ s^3 & s^4 & s^5 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ -3\, s^2 & -4\, s^3 & -5\, s^4 & 0 & 0 & 0 \\ -s^3\tau & -\tau\, s^4 & -\tau\, s^5 & 0 & 0 & 0 \\ -\kappa\, s^3 & -\kappa\, s^4 & -\kappa\, s^5 & 1 & s & s^2 \end{bmatrix} \quad \text{Eq. 2.77}$$

The strain-displacement relations are given in the matrix form as:

$$\boldsymbol{\varepsilon^T} = [\varepsilon_z \quad \omega_x \quad \omega_y \quad \omega_z] \quad \text{Eq. 2.78}$$

$$\boldsymbol{\varepsilon = \partial u = \partial A q = \partial C A u_{nodal} = B C u_{nodal}} \quad \text{Eq. 2.79}$$

The strain-displacement matrix is given by:

$$\boldsymbol{B} = [\boldsymbol{B_1} \quad \boldsymbol{B_2} \quad \boldsymbol{B_3} \quad \boldsymbol{B_4} \quad \boldsymbol{B_5}] \quad \text{Eq. 2.80}$$

$$\boldsymbol{B_1} = \begin{bmatrix} -\kappa & -\kappa s & 0 \\ 0 & -2\tau & -\frac{2\tau\, s^3\kappa^2}{3} - 4\, s\tau \\ -\tau^2 & -\tau^2 s & (\kappa^2 - \tau^2)s^2 + 2 \\ \kappa\tau & \kappa\tau s & 0 \end{bmatrix} \qquad \text{Eq. 2.81}$$

$$\boldsymbol{B_2} = \begin{bmatrix} 0 & 0 & 0 \\ -\frac{\tau\, s^4\kappa^2}{2} - 6\, s^2\tau & -\frac{2\tau\, s^5\kappa^2}{5} - 8\, s^3\tau & -\frac{\tau\, \kappa^2 s^6}{3} - 10\, \tau\, s^4 \\ (\kappa^2 - \tau^2)s^3 + 6s & (\kappa^2 - \tau^2)\kappa^2 + 12s^2 & (\kappa^2 - \tau^2)s^5 + 20s^3 \\ 0 & 0 & 0 \end{bmatrix} \qquad \text{Eq. 2.82}$$

$$\boldsymbol{B_3} = \begin{bmatrix} 0 & 1 & 2\, s & 0 & 0 & 0 \\ -\kappa\, \tau & -\kappa\, s\tau & -\tau\, s^2\kappa & \tau^2 & s\tau^2 & s^2\tau^2 - 2 \\ 0 & \kappa & 2\, \kappa\, s & 0 & -2\, \tau & -4\, s\tau \\ 0 & 0 & 0 & 0 & \kappa & 2\, \kappa\, s \end{bmatrix} \qquad \text{Eq. 2.83}$$

$$\boldsymbol{B_4} = \begin{bmatrix} 0 & 0 & 0 \\ (\kappa^2 - \tau^2)s^3 - 6s & (\kappa^2 - \tau^2)s^4 - 12s^2 & (\kappa^2 - \tau^2)s^5 - 20s^3 \\ -6\tau s^2 & -8\tau s^3 & -10\tau s^4 \\ 0 & 0 & 0 \end{bmatrix} \qquad \text{Eq. 2.84}$$

$$\boldsymbol{B_5} = \begin{bmatrix} 0 & 0 & 0 \\ \kappa & \kappa s & \kappa\, s^2 \\ 0 & 0 & 0 \\ 0 & 1 & 2s \end{bmatrix} \qquad \text{Eq. 2.85}$$

The element stiffness matrix is then calculated by:

$$\boldsymbol{K}_{el} = \boldsymbol{C}^T\boldsymbol{B}^T\boldsymbol{E}\boldsymbol{B}\boldsymbol{C} \qquad \text{Eq. 2.86}$$

where:

- $\boldsymbol{E}$ – is the material elasticity matrix.

After defining the stiffness matrix on the local cross-section coordinate system (which is aligned with the Frénet triad), it must be rotated to the cylindrical coordinate system:

$$\boldsymbol{K} = \mathbb{T}^T \boldsymbol{K} \mathbb{T} \qquad \text{Eq. 2.87}$$

with:

$$\mathbb{T} = \begin{bmatrix} \boldsymbol{T} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{T} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{T} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{T} & \boldsymbol{0} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{T} & \boldsymbol{0} \\ \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{0} & \boldsymbol{T} \end{bmatrix} \qquad \text{Eq. 2.88}$$

where:

$$\boldsymbol{T} = \begin{bmatrix} -1 & 0 & 0 \\ 0 & -\cos\alpha & \sin\alpha \\ 0 & \sin\alpha & \cos\alpha \end{bmatrix} \qquad \text{Eq. 2.89}$$

2.3 Bridge Finite Macroelement for Contact of Nodes with Different Displacement Descriptions

The nodes of the orthotropic cylinder element (2.1) and the helical beam element (2.2) have different displacement descriptions: in the first case, each node has 3 degrees of freedom ($UR, U\theta, UZ$) expanded in Fourier series, dependent on an expansion order parameter; while in the second case the nodes are conventional, with 6 degrees of freedom ($UR, U\theta, UZ, RotR, Rot\theta, RotZ$). Thus, in order to simulate a rigid connection between these two elements, which could occur when modeling the interaction between a polymeric sheath and a tensile armor, for instance, (PROVASI & MARTINS, 2013-a) had to formulate a rigid-connection finite macroelement for nodes with different natures. As shown in Fig. 2.2, this is a *node-to-node* contact element, wherein each node is described in its correspondent nature.

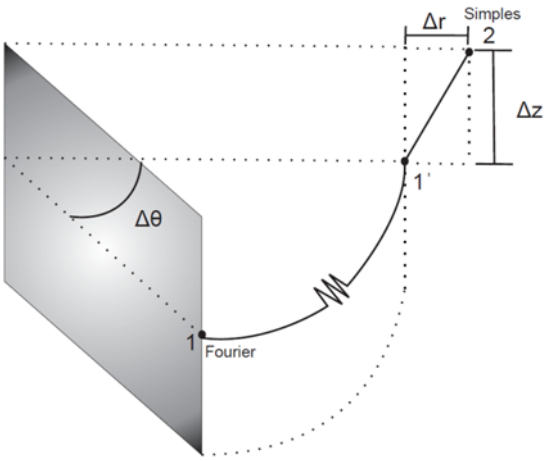

Fig. 2.2 – Bridge contact macroelement with different nodes displacements natures. Source: (PROVASI & MARTINS, 2013-a).

This element is ruled by the condition:

$$\boldsymbol{x}_2 - \boldsymbol{x}_1 = \boldsymbol{Q} = \begin{Bmatrix} \Delta r \\ \Delta\theta \\ \Delta z \end{Bmatrix} \quad \text{Eq. 2.90}$$

$$\boldsymbol{x}_2 - \boldsymbol{x}_{1'} = \boldsymbol{Q} = \begin{Bmatrix} \Delta r \\ 0 \\ \Delta z \end{Bmatrix} \quad \text{Eq. 2.91}$$

where:

- Δr – is the variation in radial direction;
- $\Delta\theta$ – is the variation in circumferential direction;
- Δz – is the variation in axial direction;
- $\boldsymbol{x}_1$ – denotes the coordinates of node **1** (Fourier);
- $\boldsymbol{x}_{1'}$ – denotes the coordinates of point **1'**, which is equal to the position of node **1** updated by the circumferential variation;
- $\boldsymbol{x}_2$ – denotes the coordinates of node **2** (conventional).

In the deformed configuration,

$$\boldsymbol{x}_2^d - \boldsymbol{x}_{1'}^d - \boldsymbol{Q} = \boldsymbol{0} \quad \text{Eq. 2.92}$$

The main condition governing this element is the inexistence of relative displacement between the nodes, given by:

$$\boldsymbol{u}_2 - \boldsymbol{u}_1' = 0 \qquad \text{Eq. 2.93}$$

$$\boldsymbol{u}_1' = \begin{bmatrix} \bar{u}_{1'}^0 & -\bar{\bar{v}}_{1'}^0 & \bar{w}_{1'}^0 \end{bmatrix}^T \boldsymbol{\Lambda}_0 + \sum_{i=1}^{n} \begin{bmatrix} \bar{u}_{1'}^i & \bar{v}_{1'}^i & \bar{w}_{1'}^i \end{bmatrix}^T \mathbb{C}_i + \sum_{i=1}^{n} \begin{bmatrix} \bar{\bar{u}}_{1'}^i & \bar{\bar{v}}_{1'}^i & \bar{\bar{w}}_{1'}^i \end{bmatrix}^T \mathbb{S}_i \qquad \text{Eq. 2.94}$$

$$\boldsymbol{u}_2 = \begin{bmatrix} u_r & u_\theta & u_z & \varphi_r & \varphi_\theta & \varphi_z \end{bmatrix}^T \qquad \text{Eq. 2.95}$$

where:

- u_r, u_θ and u_z – are the displacements for node 2 in radial, circumferential and axial direction, respectively;
- φ_r, φ_θ and φ_z – are the rotations for node 2 around radial, circumferential and axial axis, respectively;
- $\bar{u}_{1'}^0$, $-\bar{\bar{v}}_{1'}^0$ and $\bar{w}_{1'}^0$ are the order **0** displacements of point **1'**;
- $\begin{bmatrix} \bar{u}_{1'}^i & \bar{v}_{1'}^i & \bar{w}_{1'}^i \end{bmatrix}$ and $\begin{bmatrix} \bar{\bar{u}}_{1'}^i & \bar{\bar{v}}_{1'}^i & \bar{\bar{w}}_{1'}^i \end{bmatrix}$ – are the higher order single-barred and double-barred displacements in radial, circumferential and axial directions, respectively.

$$\mathbb{C}_i = \begin{bmatrix} \cos i\Delta\theta & 0 & 0 \\ 0 & \sin i\Delta\theta & 0 \\ 0 & 0 & \cos i\Delta\theta \end{bmatrix} \qquad \text{Eq. 2.96}$$

$$\mathbb{S}_i = \begin{bmatrix} \sin i\Delta\theta & 0 & 0 \\ 0 & -\cos i\Delta\theta & 0 \\ 0 & 0 & \sin i\Delta\theta \end{bmatrix} \qquad \text{Eq. 2.97}$$

$$\boldsymbol{\Lambda}_0 = \mathbb{C}_0 + \mathbb{S}_0 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \qquad \text{Eq. 2.98}$$

During the review of this paper, it was noted two signs typos in the original formulation. The first is at the order **0** circumferential displacement, $-\bar{\bar{v}}_{1'}^{0}$, which must be negative. The second one lies in Eq. 2.97, since the term $\mathbb{S}_i(2,2) = -\cos i\Delta\theta$ was positive in the original formulation, but it also needs to be negative.

The condition in Eq. 2.93 is then expressed as a constraint:

$$\boldsymbol{t}_{constraint} = [-\boldsymbol{\Lambda}_0 \quad -\mathbb{C}_1 \quad -\mathbb{S}_1 \quad \cdots \quad -\mathbb{C}_n \quad -\mathbb{S}_n \quad I] \begin{Bmatrix} \bar{\boldsymbol{u}}_{1'}^{0} \\ \bar{\boldsymbol{u}}_{1'}^{1} \\ \bar{\bar{\boldsymbol{u}}}_{1'}^{1} \\ \vdots \\ \bar{\boldsymbol{u}}_{1'}^{n} \\ \bar{\bar{\boldsymbol{u}}}_{1'}^{n} \\ \boldsymbol{u}_2 \end{Bmatrix} \qquad \text{Eq. 2.99}$$

$$\boldsymbol{t}_{constraint} = \boldsymbol{C}\boldsymbol{U} \qquad \text{Eq. 2.100}$$

In (PROVASI & MARTINS, 2013-a), Eq. 2.99 has also some sign typos, but here it is presented the corrected version. Applying a penalty methodology, the stiffness matrix is given by:

$$\boldsymbol{K} = \varepsilon \boldsymbol{C}^T \boldsymbol{C} \qquad \text{Eq. 2.101}$$

$$\boldsymbol{K} = \begin{bmatrix} \boldsymbol{\Lambda}_0\boldsymbol{\Lambda}_0 & \boldsymbol{\Lambda}_0\mathbb{C}_1 & \boldsymbol{\Lambda}_0\mathbb{S}_1 & \cdots & \boldsymbol{\Lambda}_0\mathbb{C}_n & \boldsymbol{\Lambda}_0\mathbb{S}_n & -\boldsymbol{\Lambda}_0 \\ \mathbb{C}_1\boldsymbol{\Lambda}_0 & \mathbb{C}_1\mathbb{C}_1 & \mathbb{C}_1\mathbb{S}_1 & \cdots & \mathbb{C}_1\mathbb{C}_n & \mathbb{C}_1\mathbb{S}_n & -\mathbb{C}_1 \\ \mathbb{S}_1\boldsymbol{\Lambda}_0 & \mathbb{S}_1\mathbb{C}_1 & \mathbb{S}_1\mathbb{S}_1 & \cdots & \mathbb{S}_1\mathbb{C}_n & \mathbb{S}_1\mathbb{S}_n & -\mathbb{S}_1 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ \mathbb{C}_n\boldsymbol{\Lambda}_0 & \mathbb{C}_n\mathbb{C}_1 & \mathbb{C}_n\mathbb{S}_1 & \cdots & \mathbb{C}_n\mathbb{C}_n & \mathbb{C}_n\mathbb{S}_n & -\mathbb{C}_n \\ \mathbb{S}_n\boldsymbol{\Lambda}_0 & \mathbb{S}_n\mathbb{C}_1 & \mathbb{S}_n\mathbb{S}_1 & \cdots & \mathbb{S}_n\mathbb{C}_n & \mathbb{S}_n\mathbb{S}_n & -\mathbb{S}_n \\ -\boldsymbol{\Lambda}_0 & -\mathbb{C}_1 & -\mathbb{S}_1 & \cdots & -\mathbb{C}_n & -\mathbb{S}_n & I \end{bmatrix} \qquad \text{Eq. 2.102}$$

where:

- ε – is the penalty coefficient, which has to be large for obtaining an accurate result, but not large enough to make the stiffness matrix ill-conditioned.

2.4 Standard Finite Macroelement for Contact of Nodes with Different Displacement Descriptions

(PROVASI & MARTINS, 2013-b) also proposed a contact finite macroelement for nodes with different displacement natures that considers normal and tangential displacements, as well as frictional effects. (TONI, F.G., 2014) noticed some signal typos in the original formulation and proposed a revised formulation to this element stiffness matrix.

This element considers the following modes concerning friction, which are also illustrated on Fig. 2.3, :

- ***Sticking*** – with no relative movement between nodes;
- ***Sliding*** – with relative movement between nodes.

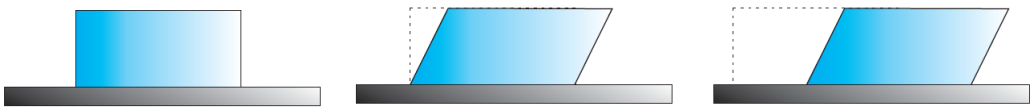

Fig. 2.3 – First case: block in initial condition; Second: sticking condition; Third: sliding condition. Source: (PROVASI, 2013).

It is important to note that this element applies only to small displacements due its *node-to-node* characteristic. For large displacements and deformations, a *node-to-surface* and *beam-to-surface* approaches are more suitable, but these elements have not yet been formulated to the present date. However, when the use limitations are not violated, this finite macroelement permits the simulation of more realistic flexible pipe behaviors, that include friction effects and nodal relative displacements.

As illustrated in Fig. 2.4, this contact element possesses two nodes: the first one is of the Fourier type, with displacements expanded in Fourier series; the second one is a conventional node, with six degrees-of-freedom.

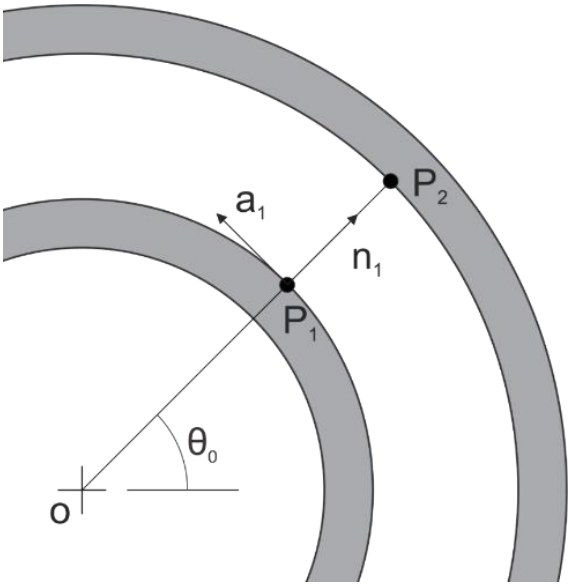

Fig. 2.4 – Node-to-node contact: node 1 (Fourier) and node 2 (conventional). Source: (TONI, F.G., 2014).

The following parameters are known in advance:

- θ_0 – the angle difference between both nodes, measured in a cylindrical coordinate system;
- $\boldsymbol{P}_1$ and $\boldsymbol{P}_2$ – are the nodal coordinates;
- $\vec{\boldsymbol{n}}_1$ – is the surface normal at node 1;
- $\vec{\boldsymbol{a}}_1$ and $\vec{\boldsymbol{a}}_2$ – are the directions tangential to the surface at node 1.

The following vectors are defined:

$$\overline{OP}_1 = \boldsymbol{X}_1^r \quad \text{and} \quad \overline{OP}_1' = \boldsymbol{x}_1 \qquad \text{Eq. 2.103}$$

$$\overline{OP}_2 = \boldsymbol{X}_2^r \quad \text{and} \quad \overline{OP}_2' = \boldsymbol{x}_2 \qquad \text{Eq. 2.104}$$

where:

- $\boldsymbol{X}_1^r$ – is the coordinate of node 1 in the reference configuration;
- $\boldsymbol{X}_2^r$ – is the coordinate of node 2 in the reference configuration;
- $\boldsymbol{x}_1$ – is the coordinate of node 1 in the deformed configuration;
- $\boldsymbol{x}_2$ – is the coordinate of node 2 in the deformed configuration.

The displacements are defined by:

$$\boldsymbol{u_1} = \boldsymbol{x}_1 - \boldsymbol{X}_1 = \begin{Bmatrix} u_1 \\ v_1 \\ w_1 \end{Bmatrix} = \begin{Bmatrix} \sum_{i=0}^{n} \bar{u}_1^i \cos i\theta_0 + \sum_{i=0}^{n} \bar{\bar{u}}_1^i \sin i\theta_0 \\ \sum_{i=0}^{n} \bar{v}_1^i \sin i\theta_0 - \sum_{i=0}^{n} \bar{\bar{v}}_1^i \cos i\theta_0 \\ \sum_{i=0}^{n} \bar{w}_1^i \cos i\theta_0 + \sum_{i=0}^{n} \bar{\bar{w}}_1^i \sin i\theta_0 \end{Bmatrix} \quad \text{Eq. 2.105}$$

$$\boldsymbol{u_2} = \boldsymbol{x}_2 - \boldsymbol{X}_2 = \begin{Bmatrix} u_2 \\ v_2 \\ w_2 \end{Bmatrix} \quad \text{Eq. 2.106}$$

$$\boldsymbol{u_1} = \begin{Bmatrix} \bar{u}_1^0 + \sum_{i=1}^{n} \bar{u}_1^i \cos i\theta_0 + \sum_{i=1}^{n} \bar{\bar{u}}_1^i \sin i\theta_0 \\ -\bar{\bar{v}}_1^0 + \sum_{i=1}^{n} \bar{v}_1^i \sin i\theta_0 - \sum_{i=1}^{n} \bar{\bar{v}}_1^i \cos i\theta_0 \\ \bar{w}_1^0 + \sum_{i=1}^{n} \bar{w}_1^i \cos i\theta_0 + \sum_{i=1}^{n} \bar{\bar{w}}_1^i \sin i\theta_0 \end{Bmatrix} \quad \text{Eq. 2.107}$$

$$\boldsymbol{u}_1 = \boldsymbol{u}_1^0 + \sum_{i=1}^{n} \left[\boldsymbol{C}_i \bar{\boldsymbol{u}}_1^i + \boldsymbol{S}_i \bar{\bar{\boldsymbol{u}}}_1^i \right] \quad \text{Eq. 2.108}$$

with:

$$\boldsymbol{u}_1 = \begin{Bmatrix} u_1 \\ v_1 \\ w_1 \end{Bmatrix} \quad \text{Eq. 2.109}$$

$$\boldsymbol{u}_1^0 = \begin{Bmatrix} \bar{u}_1^0 \\ -\bar{\bar{v}}_1^0 \\ \bar{w}_1^0 \end{Bmatrix} \ , \ \bar{\boldsymbol{u}}_1^i = \begin{Bmatrix} \bar{u}_1^i \\ \bar{v}_1^i \\ \bar{w}_1^i \end{Bmatrix} \text{ and } \bar{\bar{\boldsymbol{u}}}_1^i = \begin{Bmatrix} \bar{\bar{u}}_1^i \\ \bar{\bar{v}}_1^i \\ \bar{\bar{w}}_1^i \end{Bmatrix} \quad \text{Eq. 2.110}$$

$$\boldsymbol{C}_i = \begin{bmatrix} \cos i\theta_0 & 0 & 0 \\ 0 & \sin i\theta_0 & 0 \\ 0 & 0 & \cos i\theta_0 \end{bmatrix} \quad \text{Eq. 2.111}$$

$$\boldsymbol{S}_i = \begin{bmatrix} \sin i\theta_0 & 0 & 0 \\ 0 & -\cos i\theta_0 & 0 \\ 0 & 0 & \sin i\theta_0 \end{bmatrix} \quad \text{Eq. 2.112}$$

A normal gap function is then defined:

$$g_N = (\boldsymbol{x}_2 - \boldsymbol{x}_1) \,.\, \vec{\boldsymbol{n}}_{\boldsymbol{1}} \quad \text{Eq. 2.113}$$

And also a tangential gap function:

$$\boldsymbol{g}_T = g_{T_1}\, \vec{\boldsymbol{a}}_{\boldsymbol{1}} + g_{T_2}\, \vec{\boldsymbol{a}}_{\boldsymbol{2}} \quad \text{Eq. 2.114}$$

with:

$$g_{T_\alpha} = (\boldsymbol{x}_2 - \boldsymbol{x}_1).\vec{\boldsymbol{a}}_{\boldsymbol{\alpha}} \quad \text{for } \alpha = 1, 2 \quad \text{Eq. 2.115}$$

From the Principle of Virtual Work, the virtual work is given by:

$$\delta W_{contact} = \varepsilon_N g_N \,.\, \delta g_N + \varepsilon_T \boldsymbol{g_T} \,.\, \boldsymbol{\delta g_T} \quad \text{Eq. 2.116}$$

Deriving Eq. 2.116,

$$\delta(\delta W_{contact}) = \varepsilon_N \delta g_N \,.\, \delta g_N + \varepsilon_T \boldsymbol{\delta g_T} \,.\, \boldsymbol{\delta g_T} \quad \text{Eq. 2.117}$$

Rewriting it in matrix form:

$$\delta(\delta W_{contato}) = \varepsilon_N \delta g_N \delta g_N + \varepsilon_T \boldsymbol{\delta g_T}^{\boldsymbol{T}} \boldsymbol{\delta g_T} \quad \text{Eq. 2.118}$$

with:

$$\boldsymbol{\delta g_T} = \delta g_{T_1}\, \boldsymbol{a_1} + \delta g_{T_2}\, \boldsymbol{a_2} \quad \text{Eq. 2.119}$$

$$\boldsymbol{n} = \begin{Bmatrix} 1 \\ 0 \\ 0 \end{Bmatrix} \quad , \quad \boldsymbol{a_1} = \begin{Bmatrix} 0 \\ 1 \\ 0 \end{Bmatrix} \text{ and } \boldsymbol{a_2} = \begin{Bmatrix} 0 \\ 0 \\ 1 \end{Bmatrix} \quad \text{Eq. 2.120}$$

$$\begin{aligned}\delta\delta g_N &= \boldsymbol{n^T}\delta\boldsymbol{x_2} - \boldsymbol{n^T}\delta\boldsymbol{x_1}\\ \delta g_{T_1} &= \boldsymbol{a_1^T}\delta\boldsymbol{x_2} - \boldsymbol{a_1^T}\delta\boldsymbol{x_1}\\ \delta g_{T_2} &= \boldsymbol{a_2^T}\delta\boldsymbol{x_2} - a_2^T\delta\boldsymbol{x_1}\end{aligned} \qquad \text{Eq. 2.121}$$

The following relations are valid:

$$\boldsymbol{\delta u_2 = \delta x_2} \;\;,\;\; \boldsymbol{\delta\bar{u}_1^i = \delta\bar{x}_1^i} \;\;\;,\;\;\; \boldsymbol{\delta\bar{\bar{u}}_1^i = \delta\bar{\bar{x}}_1^i} \qquad \text{Eq. 2.122}$$

However, the derivatives for the order 0 displacements of node 1 are not trivial:

$$\boldsymbol{\delta x_1^0 \neq \delta u_1^0} \qquad \text{Eq. 2.123}$$

Remembering that $\boldsymbol{u_1^0} = [\bar{u}_1^0 \quad -\bar{\bar{v}}_1^0 \quad \bar{w}_1^0]^T$, where $-\bar{\bar{v}}_1^0$ is the first term of sum $-\sum_{i=0}^{n}\bar{\bar{v}}_1^i \cos i\theta_0$ for when $i = 0$, the following transformation is required:

$$\boldsymbol{u_1^0} = \begin{bmatrix}1 & 0 & 0\\ 0 & -1 & 0\\ 0 & 0 & 1\end{bmatrix}\begin{Bmatrix}\bar{u}_1^0\\ \bar{\bar{v}}_1^0\\ \bar{w}_1^0\end{Bmatrix} = B\begin{Bmatrix}\bar{u}_1^0\\ \bar{\bar{v}}_1^0\\ \bar{w}_1^0\end{Bmatrix} \qquad \text{Eq. 2.124}$$

Then,

$$B\,\boldsymbol{\delta u_1^0} = \boldsymbol{\delta x_1^0} \qquad \text{Eq. 2.125}$$

$$B = \begin{bmatrix}1 & 0 & 0\\ 0 & -1 & 0\\ 0 & 0 & 1\end{bmatrix} \qquad \text{Eq. 2.126}$$

$$(\; B = B^T \quad \text{e} \quad BB^T = I \;)$$

Then expressions in Eq. 2.121 are then calculated by:

$$\delta g_N = \boldsymbol{n^T\delta u_2} - \boldsymbol{n^T}B\boldsymbol{\delta u_1^0} - \sum_{i=1}^{n}\left[\boldsymbol{n^T}C_i\boldsymbol{\delta\bar{u}_1^i} + \boldsymbol{n^T}S_i\boldsymbol{\delta\bar{\bar{u}}_1^i}\right] \qquad \text{Eq. 2.127}$$

$$\delta g_{T_1} = \boldsymbol{a_1^T \delta u_2} - \boldsymbol{a_1^T B \delta u_1^0} - \sum_{i=1}^{n} \left[\boldsymbol{a_1^T C_i \delta \bar{u}_1^i} + \boldsymbol{a_1^T S_i \delta \bar{\bar{u}}_1^i} \right]$$

$$g_{T_2} = \boldsymbol{a_2^T \delta u_2} - \boldsymbol{a_2^T B \delta u_1^0} - \sum_{i=1}^{n} \left[\boldsymbol{a_2^T C_i \delta \bar{u}_1^i} + \boldsymbol{a_2^T S_i \delta \bar{\bar{u}}_1^i} \right]$$

The stiffness matrix for the *sticking* case are given by:

$$\boldsymbol{K}_{sticking} = \varepsilon_N \boldsymbol{M}_n + \varepsilon_T \boldsymbol{M}_{a1} + \varepsilon_T \boldsymbol{M}_{a2} \qquad \text{Eq. 2.128}$$

With:

$$\boldsymbol{M}_n = \begin{bmatrix} \boldsymbol{nn^T} & \boldsymbol{Bnn^TC_1} & \boldsymbol{Bnn^TS_1} & \dots & \boldsymbol{Bnn^TC_n} & \boldsymbol{Bnn^TS_n} & \boldsymbol{-Bnn^T} \\ \boldsymbol{BC_1nn^T} & \boldsymbol{C_1nn^TC_1} & \boldsymbol{C_1nn^TS_1} & \dots & \boldsymbol{C_1nn^TC_n} & \boldsymbol{C_1nn^TS_n} & \boldsymbol{-C_1nn^T} \\ \boldsymbol{BS_1nn^T} & \boldsymbol{S_1nn^TC_1} & \boldsymbol{S_1nn^TS_1} & \dots & \boldsymbol{S_1nn^TC_n} & \boldsymbol{S_1nn^TS_n} & \boldsymbol{-S_1nn^T} \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ \boldsymbol{BC_nnn^T} & \boldsymbol{C_nnn^TC_1} & \boldsymbol{C_nnn^TS_1} & \dots & \boldsymbol{C_nnn^TC_n} & \boldsymbol{C_nnn^TS_n} & \boldsymbol{-C_nnn^T} \\ \boldsymbol{BS_nnn^T} & \boldsymbol{S_nnn^TC_1} & \boldsymbol{S_nnn^TS_1} & \dots & \boldsymbol{S_nnn^TC_n} & \boldsymbol{S_nnn^TS_n} & \boldsymbol{-S_nnn^T} \\ \boldsymbol{-Bnn^T} & \boldsymbol{-nn^TC_1} & \boldsymbol{-nn^TS_1} & \dots & \boldsymbol{-nn^TC_n} & \boldsymbol{-nn^TS_n} & \boldsymbol{nn^T} \end{bmatrix} \qquad \text{Eq. 2.129}$$

$$\boldsymbol{M}_{a1} = \begin{bmatrix} \boldsymbol{a_1a_1^T} & \boldsymbol{Ba_1a_1^TC_1} & \boldsymbol{Ba_1a_1^TS_1} & \dots & \boldsymbol{Ba_1a_1^TC_n} & \boldsymbol{Ba_1a_1^TS_n} & \boldsymbol{-Ba_1a_1^T} \\ \boldsymbol{BC_1a_1a_1^T} & \boldsymbol{C_1a_1a_1^TC_1} & \boldsymbol{C_1a_1a_1^TS_1} & \dots & \boldsymbol{C_1a_1a_1^TC_n} & \boldsymbol{C_1a_1a_1^TS_n} & \boldsymbol{-C_1a_1a_1^T} \\ \boldsymbol{BS_1a_1a_1^T} & \boldsymbol{S_1a_1a_1^TC_1} & \boldsymbol{S_1a_1a_1^TS_1} & \dots & \boldsymbol{S_1a_1a_1^TC_n} & \boldsymbol{S_1a_1a_1^TS_n} & \boldsymbol{-S_1a_1a_1^T} \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ \boldsymbol{BC_na_1a_1^T} & \boldsymbol{C_na_1a_1^TC_1} & \boldsymbol{C_na_1a_1^TS_1} & \dots & \boldsymbol{C_na_1a_1^TC_n} & \boldsymbol{C_na_1a_1^TS_n} & \boldsymbol{-C_na_1a_1^T} \\ \boldsymbol{BS_na_1a_1^T} & \boldsymbol{S_na_1a_1^TC_1} & \boldsymbol{S_na_1a_1^TS_1} & \dots & \boldsymbol{S_na_1a_1^TC_n} & \boldsymbol{S_na_1a_1^TS_n} & \boldsymbol{-S_na_1a_1^T} \\ \boldsymbol{-Ba_1a_1^T} & \boldsymbol{-a_1a_1^TC_1} & \boldsymbol{-a_1a_1^TS_1} & \dots & \boldsymbol{-a_1a_1^TC_n} & \boldsymbol{-a_1a_1^TS_n} & \boldsymbol{a_1a_1^T} \end{bmatrix} \qquad \text{Eq. 2.130}$$

$$\boldsymbol{M}_{a2} = \begin{bmatrix} \boldsymbol{a_2a_2^T} & \boldsymbol{Ba_2a_2^TC_1} & \boldsymbol{Ba_2a_2^TS_1} & \dots & \boldsymbol{Ba_2a_2^TC_n} & \boldsymbol{Ba_2a_2^TS_n} & \boldsymbol{-Ba_2a_2^T} \\ \boldsymbol{BC_1a_2a_2^T} & \boldsymbol{C_1a_2a_2^TC_1} & \boldsymbol{C_1a_2a_2^TS_1} & \dots & \boldsymbol{C_1a_2a_2^TC_n} & \boldsymbol{C_1a_2a_2^TS_n} & \boldsymbol{-C_1a_2a_2^T} \\ \boldsymbol{BS_1a_2a_2^T} & \boldsymbol{S_1a_2a_2^TC_1} & \boldsymbol{S_1a_2a_2^TS_1} & \dots & \boldsymbol{S_1a_2a_2^TC_n} & \boldsymbol{S_1a_2a_2^TS_n} & \boldsymbol{-S_1a_2a_2^T} \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots \\ \boldsymbol{BC_na_2a_2^T} & \boldsymbol{C_na_2a_2^TC_1} & \boldsymbol{C_na_2a_2^TS_1} & \dots & \boldsymbol{C_na_2a_2^TC_n} & \boldsymbol{C_na_2a_2^TS_n} & \boldsymbol{-C_na_2a_2^T} \\ \boldsymbol{BS_na_2a_2^T} & \boldsymbol{S_na_2a_2^TC_1} & \boldsymbol{S_na_2a_2^TS_1} & \dots & \boldsymbol{S_na_2a_2^TC_n} & \boldsymbol{S_na_2a_2^TS_n} & \boldsymbol{-S_na_2a_2^T} \\ \boldsymbol{-Ba_2a_2^T} & \boldsymbol{-a_2a_2^TC_1} & \boldsymbol{-a_2a_2^TS_1} & \dots & \boldsymbol{-a_2a_2^TC_n} & \boldsymbol{-a_2a_2^TS_n} & \boldsymbol{a_2a_2^T} \end{bmatrix} \qquad \text{Eq. 2.131}$$

The stiffness matrices for the sliding condition were omitted here, but can be found in (PROVASI & MARTINS, 2013-b).

3 Element-by-Element Method

This chapter consists of a review on the application of the element-by-element method (EBE) to the conventional finite element method. When developing a program for large-scale model, it is important to use proper algorithms and data structures due to limitations of computational resources. The EBE method fits in this context, since it is characterized by the global stiffness matrix elimination, so that most calculations are performed in an element basis using a proper indexing system which relates the local degrees-of-freedom of the elements with the global ones. Thus, the storage cost increases linearly with model size in the EBE method, being, therefore, an efficient alternative to the conventional sparse formulation.

Furthermore, the main advantage of the EBE formulation regards on the scalability and ease of parallelization of the numerical solution. When compared to the sparse formulation, the EBE requires a larger number of operations to execute the same algorithm. However, this is rapidly compensated by techniques of parallel programming and element based domain decompositions, taking advantage from clusters and modern processors with several processing cores.

The EBE allows a fully customized solution for the elements described in Chapter 2, aiming high computational performance. The ease of adding new types of elements also must be highlighted in the EBE method, providing the necessary flexibility for future works. As the calculations are carried out in a local basis, it is necessary to implement only a matrix-vector multiplication and a scattering method between the local and global degrees-of-freedom for this element.

Other advantages of the EBE methods are: optimized cache usage by the allocation and management of blocks of similar elements; simpler algorithm implementation; domain subdivision procedure of reduced complexity in comparison to the sparse formulation.

According to (WINGET & HUGHES, 1985), there are three main ingredients in an EBE iterative linear equation solution algorithm:

1) An iterative driver algorithm;
2) A matrix which approximates the global implicit matrix and is amenable to EBE approximation;

3) The EBE approximation scheme itself.

Regarding the first ingredient, the vast majority of the EBE implementations utilizes iterative methods for solving linear systems of equations. These iterative algorithms are adapted versions of the conventional ones, such as *the preconditioned conjugate gradient method*, and will be presented in section 3.1. The choice of iterative methods is justified by the fact that they are easier to be implemented and require matrix-vector multiplications, which can trivially be done in the local basis. There are some reasons for not using direct methods for solving linear system with the EBE method. First, because they change the sparsity pattern of the global matrix, what can significantly increase the number of nonzero entries. If not properly implemented, direct methods could 'explode' memory consumption. Second, the pivoting process, common in direct methods, would create many crossed numerical terms between elements, so that the indexing would be extremely complex, becoming almost inevitably a standard sparse method, losing all the advantages of the EBE method.

For finite element structural problems, the matrix from the second ingredient is the global stiffness matrix. In the EBE method, an approximation of the global stiffness matrix is computed and used as preconditioner of the iterative methods aforementioned. The quality of this approximation has direct influence on the convergence rate of the iterative algorithm. Several approximation techniques were developed for the EBE method and they will be presented and discussed in section 3.2.

3.1 Iterative Algorithms for Linear System Solution

Iterative methods for solving linear system of equations are a very extensive research field, with many published books about this subject. Therefore, this section presents only the most common iterative methods which are used in conjunction with the element-by-element method. This section is subdivided into the sections 3.1.1 and 3.1.2 that refer to methods for symmetric linear systems.

3.1.1 Preconditioned Conjugate Gradient Method (PCG)

The preconditioned conjugate gradient method is by far the most widely used iterative method for solving symmetric linear system of equations, due its simplicity and efficiency. Its standard version is shown in Table 3.1.

Table 3.1 – Standard version of the Preconditioned Conjugate Gradient Method (PCG).

Linear System: $\boldsymbol{A\,x = b}$

1. $\boldsymbol{r_0 = b - A\,x_0}$
2. $\boldsymbol{z_0 = M^{-1} r_0}$
3. $\boldsymbol{p_0 = z_0}$
4. for $\boldsymbol{k = 0, 1, 2}$, ...
5. $\quad \boldsymbol{\alpha_k = \frac{r_k^T z_k}{p_k^T A\, p_k}}$
6. $\quad \boldsymbol{x_{k+1} = x_k + \alpha_k\, p_k}$
7. $\quad \boldsymbol{r_{k+1} = r_k - \alpha_k\, A\, p_k}$
8. $\quad$ if $\boldsymbol{\|r_{k+1}\| \leq}$ **tolerance**
9. $\quad\quad$ *Solution converged!*
10. $\quad$ end if
11. $\quad \boldsymbol{z_k = M^{-1} r_{k+1}}$
12. $\quad \boldsymbol{\beta_k = \frac{z_{k+1}^T r_{k+1}}{z_k^T r_k}}$
13. $\quad \boldsymbol{p_{k+1} = z_{k+1} + \beta_k\, p_k}$
14. end for

Source: (SAAD, 2003).

where:

- k – is the iteration count;
- $\boldsymbol{A}$ – is the global stiffness matrix;
- $\boldsymbol{r}$ – is the linear residue;
- $\boldsymbol{x_0}$ – is the initial guess or a prediction;
- $\boldsymbol{x}$ – is the trial displacement vector;
- $\boldsymbol{M}$ – is the preconditioning matrix;

- $\boldsymbol{p}$ – denotes the step direction;
- α – is the step length;
- β – defines the correction factor.

Usually, the initial guess, $\boldsymbol{x_0}$, is a null vector, simplifying the initial residual expression to $\boldsymbol{r_0 = b}$. The preconditioning matrix, $\boldsymbol{M}$, is a matrix designed to improve the rate of convergence of the method, being, therefore, very important for its efficiency, especially for ill-conditioned linear systems.

As the global stiffness matrix, $\boldsymbol{A}$, is not assembled in the EBE method, the matrix-vector product $\boldsymbol{A\,p_k}$, necessary for the computations of $\boldsymbol{\alpha_k}$ and $\boldsymbol{r_{k+1}}$, is not a trivial operation. Thus, the EBE method requires an adapted version of the PCG method.

The following works have employed the PCG algorithm on the development of the EBE method: (WINGET & HUGHES, 1985), (CAREY & JIANG, 1986), (HUGHES & FERENCZ, 1987), (LEVIT, 1987), (KING & SONNAD, 1987), (HUGHES & FERENCZ, 1988), (ADELI & KUMAR, 1995), (GULLERUD & DODDS JR, 2001), (THIAGARAJAN & ARAVAMUTHAN, 2002), (LIU, ZHOU, & YANG, 2007), (MARTÍNEZ-FRUTOS & HERRERO-PÉREZ, 2015) and (MARTÍNEZ-FRUTOS, MARTÍNEZ-CASTEJÓN, & HERRERO-PÉREZ, 2015).

Despite the differences on notations, these works share the same concept, i.e., the global matrix is defined as a sum of a series of element matrices and the matrix-vector multiplication can be performed as follows:

$$\boldsymbol{A} = \sum_{e=1}^{N_{el}} \boldsymbol{A}^e \qquad \text{Eq. 3.1}$$

$$\boldsymbol{v} = \boldsymbol{A}\,\boldsymbol{p}_k = \left(\sum_{e=1}^{N_{el}} \boldsymbol{A}^e\right)\boldsymbol{p}_k = \sum_{e=1}^{N_{el}} \boldsymbol{A}^e \boldsymbol{p}_k^e \qquad \text{Eq. 3.2}$$

$$\boldsymbol{v}^e = \boldsymbol{A}^e \boldsymbol{p}_k^e \qquad \text{Eq. 3.3}$$

$$\boldsymbol{v} = \sum_{e=1}^{N_{el}} \boldsymbol{v}^e \qquad \text{Eq. 3.4}$$

where:

- e – is the element counter;
- N_{el} – refers to the total number of elements;
- $\boldsymbol{A}^e$ – is the stiffness matrix of element e;
- $\boldsymbol{v}^e$ – is the local product $\boldsymbol{A}^e \boldsymbol{p}_k^e$ for the element e;
- $\boldsymbol{v}$ – is the global product of $\boldsymbol{A}\,\boldsymbol{p}_k$.

$\boldsymbol{A}^e$ is the eth element contribution to $\boldsymbol{A}$. Globalized element matrices $\boldsymbol{A}^e$ are used in order to simplify the notation, but in practice, only local element matrices $\boldsymbol{A}^e$ and their corresponding indexing arrays are stored.

When performed in a parallel way, this operation is not trivial. Gathering the $\boldsymbol{p}_k^e$ values and the local matrix-vector products $\boldsymbol{A}^e \boldsymbol{p}_k^e$ are independent operations and easily parallelizable. However, as the elements share nodes and degrees-of-freedom, the spreading operation from Eq. 3.4 represents a bottleneck to the EBE method and requires syncronization, once the terms of $\boldsymbol{v}$ are accessed and updated several times during the complete operation. As this syncronization is highly dependent on the hardware architecture and on the programming techniques, it will be presented and discussed in greater detail in the section 3.3.

3.1.2 Lanczos Biorthogonalization (Lanczos)

The Lanczos biorthogonalization method is an alternative to the PCG. Its standard version is shown in Table 3.2.

Table 3.2 – The Lanczos biorthogonalization procedure (SAAD, 2003).

1. Choose two vectors $\boldsymbol{v_1}$ and $\boldsymbol{w_1}$ such that $\boldsymbol{v_1 . w_1 = 1}$
2. Set $\boldsymbol{\beta_1 = \delta_1 = 0}$ and $\boldsymbol{v_0 = w_0 = 0}$
3. For j=1, 2, …, m
4. $\boldsymbol{\alpha_j = A v_j . w_j}$
5. $\boldsymbol{\hat{v}_{j+1} = A v_j - \alpha_j v_j - \beta_j v_{j-1}}$
6. $\boldsymbol{\hat{w}_{j+1} = A^T w_j - \alpha_j w_j - \delta_j w_{j-1}}$
7. $\boldsymbol{\delta_{j+1} = \left(\hat{v}_{j+1} . \hat{w}_{j+1}\right)^{1/2}}$. If $\boldsymbol{\delta_{j+1} = 0}$ Stop
8. $\boldsymbol{\beta_{j+1} = \left(\hat{v}_{j+1} . \hat{w}_{j+1}\right) / \delta_{j+1}}$
9. $\boldsymbol{w_{j+1} = \hat{w}_{j+1} / \beta_{j+1}}$
10. $\boldsymbol{v_{j+1} = \hat{v}_{j+1} / \delta_{j+1}}$
11. Endfor

(COUTINHO, ALVES, LANDAU, LIMA, & EBECKEN, 1987) applied the EBE procedure and developed an EBE version of the symmetric diagonal preconditioned Lanczos method. This algorithm was applied in large-scale offshore engineering structural problems which can be ill-conditioned in some cases. The authors of the aforementined work concluded that *the EBE Lanczos achieved, without loss of accuracy, a better computer perfomance than the EBE Conjugate Gradient* for the noticeably ill-conditioned analysed problems.

(NOUR-OMID, PARLETT, & RAEFSKY, 1987) compared the EBE preconditioned versions of Lanczos against the CG algorithms. They tested both algorithms for fluid and structural ill-conditioned problems. The Lanczos versions showed a better convergence rate, requiring, in some cases, a significantly smaller number of iterations. However, as the simulation times were not provided, the efficiency comparison becomes compromised, since the PCG iterations are faster.

(COUTINHO, ALVES, LANDAU, EBECKEN, & TROINA, 1991) implemented and compared the EBE versions of the Preconditioned Lanczos and PCG algorithms, and concluded that *although both algorithms are theoretically related, they present important implementational differences. Lanczos based procedures require a lot of I/O operations to save and restore the Lanczos vectors. The operations are of course non-vectorizable and the correspondent overhead makes Lanczos algorithms slower than their conjugate gradient counterparts*.

Therefore, it can be concluded that the Lanczos biorthogonalization is an important alternative to be considered for ill-conditioned problems. However, as only linear finite elements and material models will be used (which have been already extensively tested in MacroFEM), ill-conditioning and slow convergence rate problems are not expected. Thus, EBE-PCG is the most logical alternative for a first implementation, due to its simpler implementation, slightly lower memory requirements and high efficiency for well-conditioned problems.

3.2 EBE Preconditioners

Preconditioners are mathematical transformations developed to improve the numerical solution of a given problem. For linear systems, preconditioners usually act by reducing the condition number of the matrices, thus, increasing the rate of convergence of iterative methods. Condition number is a measure of how much a function is sensitive to errors in the input, given by the ratio between maximum and minimum (in modulus) eigenvalues of the global matrix. Consider the following linear system:

$$\boldsymbol{A}\,\boldsymbol{x} = \boldsymbol{b} \qquad \text{Eq. 3.5}$$

This linear system can be multiplied by the preconditioning matrix $\boldsymbol{M}$,

$$\boldsymbol{M}^{-1}\boldsymbol{A}\,\boldsymbol{x} = \boldsymbol{M}^{-1}\boldsymbol{b} \qquad \text{Eq. 3.6}$$

For a better convergence rate, the product $\boldsymbol{M}^{-1}\boldsymbol{A}$ should have a lower condition number than the original matrix $\boldsymbol{A}$. The preconditioner matrix $\boldsymbol{M}$ is an approximation of $\boldsymbol{A}^{-1}$ and the convergence rate is directly associated to the quality of this approximation.

There are many types of numerical preconditioners, such as: *Jacobi*, *Successive Over relaxation* (SOR), *Symmetric SOR* (SSOR), *Cholesky and LU factorizations*, Block and Multilevel Block-Matrix Preconditioners, Polynomial Preconditioners, among many others.

Not all of these preconditioners are available in the EBE forms, which are adapted or derived versions of the conventional ones. This is because EBE preconditioners must be computed without the assemblage of a global preconditioner matrix using the element-based indexing system, which, for computational efficiency, restricts the creation of

crossed numerical terms between elements. In this section are presented the two most important EBE preconditioners: the Jacobi Diagonal Preconditioner (item 3.2.1) and the Hughes-Winget Preconditioner (item 3.2.2).

3.2.1 Jacobi Diagonal Preconditioner

The Jacobi Diagonal Preconditioner is given by the diagonal of the global matrix:

$$\boldsymbol{M} = \boldsymbol{A}_{diag} = \sum_{e=1}^{N_{el}} \boldsymbol{A}_{diag}^{e} \qquad \text{Eq. 3.7}$$

In the EBE method, this preconditioner is given by the global sum of the diagonal values of each element stiffness matrix. A scattering operation between the local and global degrees-of-freedom values must be performed, which requires synchronization if it is performed in parallel. In practice, only a vector with the inverse values of the global diagonal must be stored.

3.2.2 Hughes-Winget Preconditioner

The Hughes-Winget preconditioner was developed in (WINGET & HUGHES, 1985) and has been employed in many of the subsequent works in this research field. Proceeding to (WINGET & HUGHES, 1985), the same authors have published two other works that were the basis for the development of the Hughes-Winget preconditioner. These works will be briefly described as follows in order to contextualize the reader.

In the first one, (HUGHES, LEVIT, & WINGET, 1983-A) introduced the EBE method for implicit and unconditionally stable solution of heat conduction problems, which typically required the assemblage of a global matrix of conductivity coefficients. It was proposed an element-by-element splitting algorithm, which eliminated the global matrix and allowed a solution procedure with arrays of element size. The storage requirements were equivalent to explicit methods for heat conduction, with considerable reduction of memory consumption, but with stability and accuracy of implicit methods.

Based on the potential of this technique, the authors extended EBE method to problems of structural and solid mechanics (HUGHES, LEVIT, & WINGET, 1983-B). An iterative time-discretization algorithm to solve a linear system of equations by using

approximate factorization techniques was developed, allowing it to occur on an element-by-element basis. The aforementioned authors replaced the linear system (a discrete elliptic problem) by a differential equation (an associated equivalent parabolic problem), used a trapezoidal integration algorithm and defined others auxiliary variables in order to solve this differential equation. The complete resolution is found in (HUGHES, LEVIT, & WINGET, 1983-B). When compared with direct elimination algorithms, this EBE version showed a reduced number of operations and I/O (disk input / output) advantages. Despite the good numerical results, this approach of transforming the linear system into a parabolic problem is not usual and was not taken forward in the subsequent works of the same authors.

The greatest contribution of these two publications comes from an approximation of a global matrix by a series of element matrices products:

$$\boldsymbol{A} \approx \prod_{e=1}^{N_{el}} \boldsymbol{A}^e = \boldsymbol{A}^1 \boldsymbol{A}^2 \dots \boldsymbol{A}^{N_{el}} \qquad \text{Eq. 3.8}$$

This approximate factorization scheme was further improved in (WINGET & HUGHES, 1985), originating the Hughes-Winget preconditioner.

Following the same strategy from (WINGET & HUGHES, 1985), these approximation techniques will be here firstly presented generically and, after that, particularized to the EBE method, evidencing its potential.

A two-stage factorization is performed for ease in representation. The first stage is a reduction of $\boldsymbol{A}$ into a form $\widetilde{\boldsymbol{A}}$, which is 'close' to $\boldsymbol{A}$, has a known form and can be easily factored:

$$\boldsymbol{A} \approx \widetilde{\boldsymbol{A}} = \boldsymbol{W}^{1/2}(\boldsymbol{I} + \varepsilon\, \overline{\boldsymbol{A}})\boldsymbol{W}^{1/2} \qquad \text{Eq. 3.9}$$

$$\boldsymbol{W} = \boldsymbol{A}_{diag} \qquad \text{Eq. 3.10}$$

where:

- $\boldsymbol{W}$ – is a scaling or normalizing diagonal, symmetric, positive-definite matrix. It reduces the order of $\boldsymbol{A}$ to O(1);
- ε – is a positive real number that should be a "small" parameter;

- $\overline{\boldsymbol{A}}$ – is a pre-scaled approximation and that maintains the same sparsity structure of $\boldsymbol{A}$.

Appropriate choices for the parameters $\boldsymbol{W}$, ε and $\overline{\boldsymbol{A}}$ will be discussed at the end of this section.

The second stage is the definition of the splitting matrix $\boldsymbol{M}$ (preconditioner matrix) as an approximation of $\widetilde{\boldsymbol{A}}$.

$$\boldsymbol{M} = \boldsymbol{W}^{1/2}\,\boldsymbol{C}\,\boldsymbol{W}^{1/2} \qquad \text{Eq. 3.11}$$

$$\boldsymbol{C} \approx \boldsymbol{I} + \varepsilon\,\overline{\boldsymbol{A}} \qquad \text{Eq. 3.12}$$

For computational reasons, the matrix $\boldsymbol{C}$ should be easily factorable and its inverse matrix, $\boldsymbol{C}^{-1}$**,** should be well behaved and storable in a compact form.

(WINGET & HUGHES, 1985) "*considered definitions of* $\boldsymbol{C}$ *based on sum-to-product approximations. A sum-to-product approximation approximates the sum of a number of terms by the product of scaled terms augmented by the identity".*

3.2.2.1 Two-component splitting

The two-component splitting is the easiest of the sum-to-product type approximation and is given by:

$$\overline{\boldsymbol{A}} = \overline{\boldsymbol{A}}_1 + \overline{\boldsymbol{A}}_2 \qquad \text{Eq. 3.13}$$

Using this sum-decomposition of $\overline{\boldsymbol{A}}$, the matrix $\boldsymbol{C}$ could be defined as:

$$\boldsymbol{C} = (\boldsymbol{I} + \varepsilon\,\overline{\boldsymbol{A}}_1)(\boldsymbol{I} + \varepsilon\,\overline{\boldsymbol{A}}_2) \qquad \text{Eq. 3.14}$$

$$\boldsymbol{C} = \boldsymbol{I} + \varepsilon\,\overline{\boldsymbol{A}} + \varepsilon^2\overline{\boldsymbol{A}}_1\overline{\boldsymbol{A}}_2 = \boldsymbol{I} + \varepsilon\,\overline{\boldsymbol{A}} + O(\varepsilon^2) \qquad \text{Eq. 3.15}$$

"Computational simplicity is gained if $\overline{\boldsymbol{A}}_1$ *and* $\overline{\boldsymbol{A}}_2$ *are very sparse and are easier to factor than* $\overline{\boldsymbol{A}}$*. Note that if* $\overline{\boldsymbol{A}}_1$ *and* $\overline{\boldsymbol{A}}_2$ *do not commute,* $\boldsymbol{C}$ *will not in general be symmetric*

even if $\overline{\boldsymbol{A}}$ *is symmetric. In addition, the ordering of terms in the product approximation influences the error in the approximation"* (WINGET & HUGHES, 1985).

3.2.2.2 Multi-component splitting

The multi-component splitting is a generalization of the two-component splitting, in which $\overline{\boldsymbol{A}}$ is defined by a multi-component sum of *N* operators:

$$\overline{\boldsymbol{A}} = \sum_{i=1}^{N} \overline{\boldsymbol{A}}_i \qquad \text{Eq. 3.16}$$

Analogously to the previously presented procedure, $\boldsymbol{C}$ can be approximated by a product formed of its *N* components:

$$\boldsymbol{C} = \prod_{i=1}^{N} (\boldsymbol{I} + \varepsilon\, \overline{\boldsymbol{A}}_i) \qquad \text{Eq. 3.17}$$

$$\boldsymbol{C} = (\boldsymbol{I} + \varepsilon\, \overline{\boldsymbol{A}}_1)(\boldsymbol{I} + \varepsilon\, \overline{\boldsymbol{A}}_2) \ldots (\boldsymbol{I} + \varepsilon\, \overline{\boldsymbol{A}}_N) \qquad \text{Eq. 3.18}$$

$$\boldsymbol{C} = \boldsymbol{I} + \varepsilon\, \overline{\boldsymbol{A}}_i + \varepsilon^2 \sum_{i=1}^{N-1} \left(\overline{\boldsymbol{A}}_i \sum_{j=i+1}^{N} \overline{\boldsymbol{A}}_j \right) + O(\varepsilon^2) \qquad \text{Eq. 3.19}$$

$$\boldsymbol{C} = \boldsymbol{I} + \varepsilon\, \overline{\boldsymbol{A}} + O(\varepsilon^2) \qquad \text{Eq. 3.20}$$

This approximation is known as *one-pass* multi-component splitting, the simplest of the multi-component splitting category. The quality of this approximation is influenced by the form and order of the terms in the product.

The one-pass procedure can be generalized for the two-pass and multi-pass multi-component splitting approximations. The most important expressions for these three cases are summarized in Table 3.3.

Table 3.3 – Multi-component splitting. Adapted from: (WINGET & HUGHES, 1985).

One-pass:

$$\boldsymbol{C} = \prod_{i=1}^{N} (\boldsymbol{I} + \varepsilon\, \overline{\boldsymbol{A}}_{\boldsymbol{i}}) \qquad \text{Eq. 3.21}$$

Two-pass:

$$\boldsymbol{C} = \prod_{i=1}^{N} \left(\boldsymbol{I} + \frac{1}{2}\, \varepsilon\, \overline{\boldsymbol{A}}_{\boldsymbol{i}}\right) \prod_{i=N}^{1} \left(\boldsymbol{I} + \frac{1}{2}\, \varepsilon\, \overline{\boldsymbol{A}}_{\boldsymbol{i}}\right) \qquad \text{Eq. 3.22}$$

Multi-pass:

$$\boldsymbol{C} = \prod_{j=1}^{N_{pass}} \left[\prod_{i=1}^{N} \left(\boldsymbol{I} + \frac{\varepsilon}{N_{pass}}\, \overline{\boldsymbol{A}}_{k_j(i)}\right)\right] \qquad \text{Eq. 3.23}$$

Where:

- $\boldsymbol{N_{pass}}$ is the number of passes;
- $\boldsymbol{k_j(i)}$ defines the order of the components for pass $\boldsymbol{j}$.

When $\boldsymbol{C}$ is a symmetric matrix, the multi-pass multi-component splitting can by simplified by:

$$\boldsymbol{C_{sym}} = \prod_{j=1}^{N_{pass}/2} \left[\prod_{i=1}^{N} \left(\boldsymbol{I} + \frac{\varepsilon}{N_{pass}}\, \overline{\boldsymbol{A}}_{\boldsymbol{i}}\right) \prod_{i=N}^{1} \left(\boldsymbol{I} + \frac{\varepsilon}{N_{pass}}\, \overline{\boldsymbol{A}}_{\boldsymbol{i}}\right)\right] \qquad \text{Eq. 3.24}$$

Despite being a better qualitative approximation for some cases, the multi-pass multi-component splitting demands a significantly larger number of operations, so that only the one-pass and two-pass procedures are used in practice.

3.2.2.3 Element-by-element splits

Particularizing these approximations to the element-by-element method, the components of the matrix are the pre-scaled finite element contributions to the global matrix:

$$\overline{\boldsymbol{A}} = \sum_{e=1}^{N_{el}} \overline{\boldsymbol{A}}_e \qquad \text{Eq. 3.25}$$

$\bar{\boldsymbol{A}}_e$ is the eth element contribution to $\bar{\boldsymbol{A}}$. Globalized element arrays $\bar{\boldsymbol{A}}_e$ are used in order to simplify the notation, but in practice, only local element arrays $\boldsymbol{a}_e$ and their corresponding indexing arrays are stored. Substituting the definition of $\bar{\boldsymbol{A}}_e$ into the expressions of Table 3.3, the one-pass and two-pass EBE multi-component splitting for $\boldsymbol{C}$ can be defined. These expressions are shown in Table 3.4.

Table 3.4 – EBE multi-component splitting. Adapted from (WINGET & HUGHES, 1985).

One-pass:

$$\boldsymbol{C} = \prod_{e=1}^{N_{el}} (\boldsymbol{I} + \varepsilon\, \bar{\boldsymbol{A}}_e) \qquad \text{Eq. 3.26}$$

Two-pass:

$$\boldsymbol{C} = \prod_{e=1}^{N_{el}} \left(\boldsymbol{I} + \frac{\varepsilon}{2}\, \bar{\boldsymbol{A}}_e\right) \prod_{N_{el}}^{1} \left(\boldsymbol{I} + \frac{\varepsilon}{2}\, \bar{\boldsymbol{A}}_e\right) \qquad \text{Eq. 3.27}$$

The pseudo-residual, $\boldsymbol{z}_{k-1}$, from the iterative methods (Table 3.1, for example) can be calculated directly multiplying the inverse of $\boldsymbol{M}$ by the linear residual $\boldsymbol{r}_{k-1}$:

$$\boldsymbol{z}_{k-1} = \boldsymbol{M}^{-1}\boldsymbol{r}_{k-1} = \left(\boldsymbol{W}^{1/2}\, \boldsymbol{C}\, \boldsymbol{W}^{1/2}\right)^{-1} \boldsymbol{r}_{k-1} \qquad \text{Eq. 3.28}$$

or solving the linear system:

$$\left(\boldsymbol{W}^{1/2}\, \boldsymbol{C}\, \boldsymbol{W}^{1/2}\right) \boldsymbol{z}_{k-1} = \boldsymbol{r}_{k-1} \qquad \text{Eq. 3.29}$$

This operation becomes much easier if a factored form of $\boldsymbol{C}$ is found. In this case, $\boldsymbol{z}_{k-1}$ is computed using standard direct solution techniques that require diagonal scaling, forward reductions and back substitutions. Based on this, (WINGET & HUGHES, 1985) developed three factored forms of the one-pass EBE multi-component splitting, which are in Table 3.5. The expressions for the symmetric case are summarized in Table 3.6.

Table 3.5 – One-pass EBE multi-component splitting.

Standard:

$$\boldsymbol{C} = \prod_{e=1}^{N_{el}} (\boldsymbol{I} + \varepsilon\, \bar{\boldsymbol{A}}_{\boldsymbol{e}}) \qquad \text{Eq. 3.30}$$

Crout factored form:

$$\boldsymbol{C} = \prod_{e=1}^{N_{el}} \boldsymbol{L}_{\pi}(\boldsymbol{I} + \varepsilon\bar{\boldsymbol{A}}_{e})\boldsymbol{D}_{\pi}(\boldsymbol{I} + \varepsilon\bar{\boldsymbol{A}}_{e})\boldsymbol{U}_{\pi}(\boldsymbol{I} + \varepsilon\bar{\boldsymbol{A}}_{e}) \qquad \text{Eq. 3.31}$$

Cholesky factored form:

$$\boldsymbol{C} = \prod_{e=1}^{N_{el}} \tilde{\boldsymbol{L}}_{\pi}(\boldsymbol{I} + \varepsilon\bar{\boldsymbol{A}}_{e})\tilde{\boldsymbol{U}}_{\pi}(\boldsymbol{I} + \varepsilon\bar{\boldsymbol{A}}_{e}) \qquad \text{Eq. 3.32}$$

Gauss-Seidel approximate factored form:

$$\boldsymbol{C} = \prod_{\boldsymbol{e=1}}^{N_{el}} \left(\boldsymbol{I} + \boldsymbol{\varepsilon}\tilde{\boldsymbol{L}}_{\boldsymbol{\sigma}}(\bar{\boldsymbol{A}}_{\boldsymbol{e}})\right)\left(\boldsymbol{I} + \boldsymbol{\varepsilon}\tilde{\boldsymbol{U}}_{\boldsymbol{\sigma}}(\bar{\boldsymbol{A}}_{\boldsymbol{e}})\right) \qquad \text{Eq. 3.33}$$

Adapted from: (WINGET & HUGHES, 1985).

Table 3.6 – Symmetric factorizations for one-pass EBE multi-component splitting.

Crout ($\boldsymbol{\pi}$)

$$\boldsymbol{C} = \prod_{e=1}^{N_{el}} \boldsymbol{L}_{\pi}(\boldsymbol{I} + \varepsilon\bar{\boldsymbol{A}}_{e})\boldsymbol{D}_{\pi}(\boldsymbol{I} + \varepsilon\bar{\boldsymbol{A}}_{e})\boldsymbol{L}_{\pi}^{t}(\boldsymbol{I} + \varepsilon\bar{\boldsymbol{A}}_{e}) \qquad \text{Eq. 3.34}$$

Cholesky ($\tilde{\boldsymbol{\pi}}$)

$$\boldsymbol{C} = \prod_{e=1}^{N_{el}} \tilde{\boldsymbol{L}}_{\pi}(\boldsymbol{I} + \varepsilon\bar{\boldsymbol{A}}_{e})\tilde{\boldsymbol{L}}_{\pi}^{t}(\boldsymbol{I} + \varepsilon\bar{\boldsymbol{A}}_{e}) \qquad \text{Eq. 3.35}$$

Symmetric Gauss-Seidel ($\tilde{\boldsymbol{\sigma}}$)

$$\boldsymbol{C} = \prod_{e=1}^{N_{el}} \left(\boldsymbol{I} + \varepsilon\tilde{\boldsymbol{L}}_{\sigma}(\bar{\boldsymbol{A}}_{e})\right)\left(\boldsymbol{I} + \varepsilon\tilde{\boldsymbol{L}}_{\sigma}^{t}(\bar{\boldsymbol{A}}_{e})\right) \qquad \text{Eq. 3.36}$$

Source: (WINGET & HUGHES, 1985).

The factored forms of the two-pass EBE multi-component splitting are shown in Table 3.7.

Table 3.7 – Two-pass EBE multi-component splitting.

Standard

$$\boldsymbol{C} = \prod_{e=1}^{N_{el}} \left(\boldsymbol{I} + \frac{\varepsilon}{2}\,\overline{\boldsymbol{A}}_{\boldsymbol{e}}\right) \prod_{N_{EL}}^{1} \left(\boldsymbol{I} + \frac{\varepsilon}{2}\,\overline{\boldsymbol{A}}_{\boldsymbol{e}}\right) \qquad \text{Eq. 3.37}$$

Crout factored form

$$\boldsymbol{C} = \prod_{e=1}^{N_{el}} \boldsymbol{L}_{\pi}\left(\boldsymbol{I} + \frac{\varepsilon}{2}\overline{\boldsymbol{A}}_{e}\right) \boldsymbol{D}_{\pi}\left(\boldsymbol{I} + \frac{\varepsilon}{2}\overline{\boldsymbol{A}}_{e}\right) \boldsymbol{U}_{\pi}\left(\boldsymbol{I} + \frac{\varepsilon}{2}\overline{\boldsymbol{A}}_{e}\right) \times \prod_{e=N_{EL}}^{1} \boldsymbol{L}_{\pi}\left(\boldsymbol{I} + \frac{\varepsilon}{2}\overline{\boldsymbol{A}}_{e}\right) \boldsymbol{D}_{\pi}\left(\boldsymbol{I} + \frac{\varepsilon}{2}\overline{\boldsymbol{A}}_{e}\right) \boldsymbol{U}_{\pi}\left(\boldsymbol{I} + \frac{\varepsilon}{2}\overline{\boldsymbol{A}}_{e}\right) \qquad \text{Eq. 3.38}$$

Cholesky factored form

$$\boldsymbol{C} = \prod_{e=1}^{N_{el}} \tilde{\boldsymbol{L}}_{\pi}\left(\boldsymbol{I} + \frac{\varepsilon}{2}\overline{\boldsymbol{A}}_{e}\right) \tilde{\boldsymbol{U}}_{\pi}\left(\boldsymbol{I} + \frac{\varepsilon}{2}\overline{\boldsymbol{A}}_{e}\right) \times \prod_{e=N_{el}}^{1} \tilde{\boldsymbol{L}}_{\pi}\left(\boldsymbol{I} + \frac{\varepsilon}{2}\overline{\boldsymbol{A}}_{e}\right) \tilde{\boldsymbol{U}}_{\pi}\left(\boldsymbol{I} + \frac{\varepsilon}{2}\overline{\boldsymbol{A}}_{e}\right) \qquad \text{Eq. 3.39}$$

Gauss-Seidel approximate factored form

$$\boldsymbol{C} = \prod_{e=1}^{N_{el}} \left(\boldsymbol{I} + \frac{\varepsilon}{2}\tilde{\boldsymbol{L}}_{\sigma}(\overline{\boldsymbol{A}}_{e})\right)\left(\boldsymbol{I} + \frac{\varepsilon}{2}\tilde{\boldsymbol{U}}_{\sigma}(\overline{\boldsymbol{A}}_{e})\right) \times \prod_{e=N_{el}}^{1} \left(\boldsymbol{I} + \frac{\varepsilon}{2}\tilde{\boldsymbol{L}}_{\sigma}(\overline{\boldsymbol{A}}_{e})\right)\left(\boldsymbol{I} + \frac{\varepsilon}{2}\tilde{\boldsymbol{U}}_{\sigma}(\overline{\boldsymbol{A}}_{e})\right) \qquad \text{Eq. 3.40}$$

Adapted from: (WINGET & HUGHES, 1985).

(WINGET & HUGHES, 1985) also noticed that the ordering of the factors influences how well $\boldsymbol{C}$ approximates $\boldsymbol{I} + \varepsilon\,\overline{\boldsymbol{A}}$ and therefore developed a reordered version of the EBE split factorizations, which expressions are in for the one-pass can be found in Table 3.8.

Table 3.8 – Reordered one-pass EBE multi-component splitting.

Crout factored form:

$$\boldsymbol{C} = \left[\prod_{e=1}^{N_{el}} \boldsymbol{L}_{\pi}(\boldsymbol{I} + \varepsilon\overline{\boldsymbol{A}}_{e})\right]\left[\prod_{e=1}^{N_{el}} \boldsymbol{D}_{\pi}(\boldsymbol{I} + \varepsilon\overline{\boldsymbol{A}}_{e})\right]\left[\prod_{e=N_{el}}^{1} \boldsymbol{U}_{\pi}(\boldsymbol{I} + \varepsilon\overline{\boldsymbol{A}}_{e})\right] \quad \text{Eq. 3.41}$$

Cholesky factored form:

$$\boldsymbol{C} = \left[\prod_{e=1}^{N_{el}} \tilde{\boldsymbol{L}}_{\pi}(\boldsymbol{I} + \varepsilon\overline{\boldsymbol{A}}_{e})\right]\left[\prod_{e=N_{el}}^{1} \widetilde{\boldsymbol{U}}_{\pi}(\boldsymbol{I} + \varepsilon\overline{\boldsymbol{A}}_{e})\right] \quad \text{Eq. 3.42}$$

Gauss-Seidel approximate factored form:

$$\boldsymbol{C} = \left[\prod_{\boldsymbol{e=1}}^{N_{el}} \left(\boldsymbol{I} + \varepsilon\tilde{\boldsymbol{L}}_{\boldsymbol{\sigma}}(\overline{\boldsymbol{A}}_{\boldsymbol{e}})\right)\right]\left[\prod_{\boldsymbol{e}=N_{el}}^{\mathbf{1}} \left(\boldsymbol{I} + \varepsilon\widetilde{\boldsymbol{U}}_{\boldsymbol{\sigma}}(\overline{\boldsymbol{A}}_{\boldsymbol{e}})\right)\right] \quad \text{Eq. 3.43}$$

Adapted from: (WINGET & HUGHES, 1985).

In order to complete the formulation, parameters $\boldsymbol{W}$, ε and $\overline{\boldsymbol{A}}$ from Eq. 3.9 must be considered. There are two options for the parameters $\boldsymbol{W}$ and $\overline{\boldsymbol{A}}$, and they are summarized in Table 3.9.

Table 3.9 – Choice of parameters $\boldsymbol{W}$ and $\overline{\boldsymbol{A}}$.

Parabolic regularization parameters:

$$\boldsymbol{W} = \boldsymbol{D}_{\sigma} \quad \text{Eq. 3.44}$$

$$\overline{\boldsymbol{A}} = \frac{1}{\varepsilon}\,\boldsymbol{W}^{-1/2}\boldsymbol{A}\,\boldsymbol{W}^{-1/2} \quad \text{Eq. 3.45}$$

Optimum parameters:

$$\boldsymbol{W} = \boldsymbol{D}_{\sigma} \quad \text{Eq. 3.46}$$

$$\overline{\boldsymbol{A}} = \frac{1}{\varepsilon}\,\boldsymbol{W}^{-1/2}\big(\boldsymbol{A} - \boldsymbol{D}_{\sigma}(\boldsymbol{A})\big)\,\boldsymbol{W}^{-1/2} \quad \text{Eq. 3.47}$$

Adapted from: (WINGET & HUGHES, 1985).

According to (WINGET & HUGHES, 1985), *the approximate factorizations under consideration have all had error terms of order* ε^2 *and higher. The "quality" of the approximation is governed by the size of these error terms*.

Based on obtained numerical results, (WINGET & HUGHES, 1985) concluded that *the one-pass, reordered, Crout EBE factorization, applied to the optimal definition of* $\widetilde{\boldsymbol{A}}$ *and coupled with a preconditioned conjugate gradient iteration algorithm is particularly effective for solving symmetric positive-definite matrix systems*. This version corresponds to the Hughes-Winget Preconditioner (HW).

3.3 Parallelization of the EBE method

The scalability and ease of parallelization characteristhics are among the main reasons that explain the success of the EBE method. At the same time, as parallel computing has evolved a lot on the last three decades, both in terms of programming and hardware technologies, the first EBE-PCG implementations have become outdated.

(HUGHES, LEVIT, & WINGET, 1983-A) introduced the EBE method. Given its potential, these authors have further developed the research line, publishing also (HUGHES, LEVIT, & WINGET, 1983-B), (WINGET & HUGHES, 1985) and (HUGHES & FERENCZ, 1987). In these works, significative advances have been achieved in order to establish and develop the EBE method, that led, for example, in the development of the Hughes-Winget preconditioner. However, in terms of parallelization, the employed techniques are obsolete in relation to modern computers and programming language, since they were oriented to CRAY vectorial supercomputers from the 80's, using, for example, specific data structures of blocks of 64 elements.

(CAREY & JIANG, 1986) proposed an element-by-element scheme that employs preconditioned conjugate gradient algorithms for nonlinear problems. Their implementation was tested on several computers of that time and compared it with a direct solution method, obtaining great results regarding the processing time and memory consumption. However, despite discussing the necessity of synchronization, they have not presented or discussed the aspects involving the parallelism of the solution.

(KING & SONNAD, 1987) introduced a parallel implementation of the Crout EBE preconditioned conjugate gradient method. The architecture of the computer used to run the code consisted of an array of loose processors, each one with a large memory (to the stardards of that time) and all of them connected to a shared memory via communication

bus. Despite the outdate hardware, the synchronization logics and challenges resulting from this architecture are similar to those found with MPI programming. (KING & SONNAD, 1987) subdivided the element mesh into N_P regions (where N_P is the number of available processors), each of which was handled by a separate processor, as shown in Fig. 3.1.

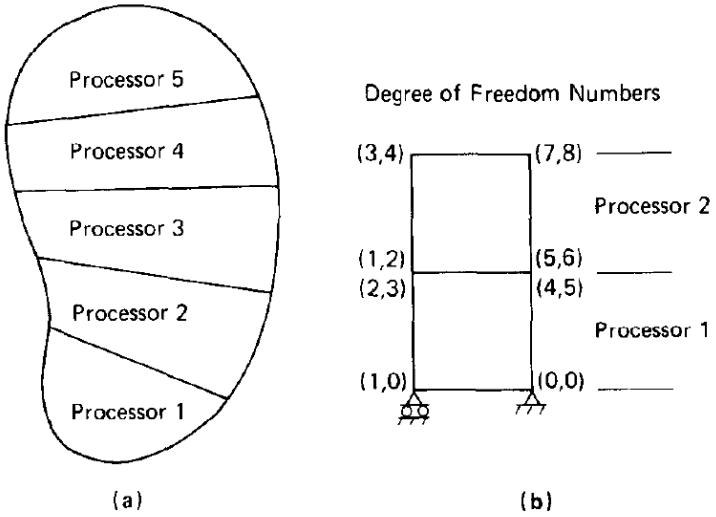

Fig. 3.1 – Schematic diagram of parallel implementation of EBE-PCG algorithm. Source: (KING & SONNAD, 1987).

The backward product from Crout preconditioner, for example, *was carried in each processor over the all elements except to those touching the top boundary of each processor*. *The shared memory was then used to pass the resulting vector components at the degrees of freedom on the bottom boundary.* This procedure is repeated until the end, concentrating the maximum possible amount of operations in each processor and using the shared memory to synchronize informations between neighbor regions or, in some cases, globally. (KING & SONNAD, 1987) found an ingenius solution with the limited computational tools available that time to generate an element-based domain decomposition in order to apply the EBE procedure. This allowed them to obtain very high speedup values in numerical experiments, as shown in Fig. 3.2.

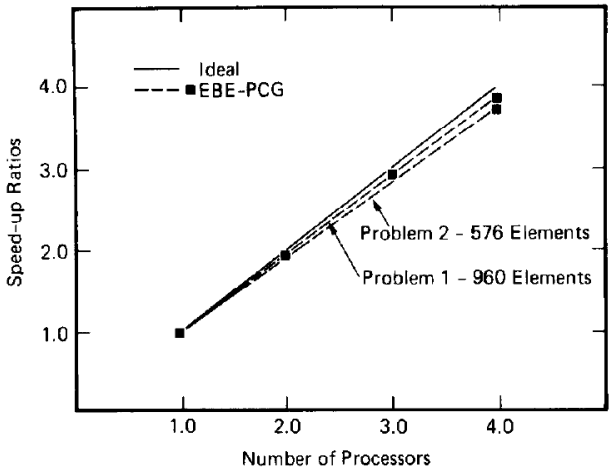

Fig. 3.2 – Parallel speedup ratios achieved on the 1CAP computer, corrected for effect of element reordering on convergence. Source: (KING & SONNAD, 1987).

(ADELI & KUMAR, 1995) presented a distributed algorithm for solving linear system of equations with a coarse-grained element-by-element preconditioned conjugate gradient method. They proposed an algorithm that divides the problem into different subdomains with overlapping regions, with the objective of reducing comunnication costs through redundant computations. Fig. 3.3 is an example with four overlapping subdomains, such that the element 31 belongs to all sets in this case.

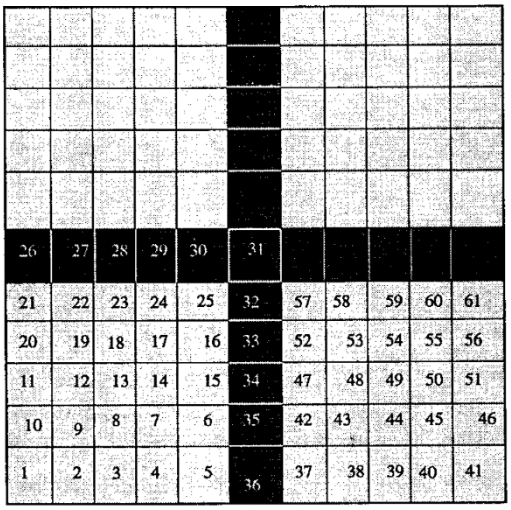

Fig. 3.3 – Distribution of elements among subdomains. Source: (ADELI & KUMAR, 1995).

(GULLERUD & DODDS JR, 2001) developed a MPI-based implementation of a PCG solver using an EBE architecture and preconditioner for implicit 3D finite element analysis for shared, distributed and distributed-shared memory computers. In this

implementation, the mesh is divided into subdomains, each of which assigned to a different processor, responsible for the local computations. In order to achieve a load balanced solution and minimize boundaries and communication costs, (GULLERUD & DODDS JR, 2001) used the graph partitioning software *METIS* to perform this task. As shown in Fig. 3.4, a second level of partitioning was performed inside of each subdomain, grouping similar elements, i.e. same type, properties, constitutive model, etc… for further improvement of computational performance, once *this framework creates inner loops with significant work loads, which expose opportunites for local parallelism*.

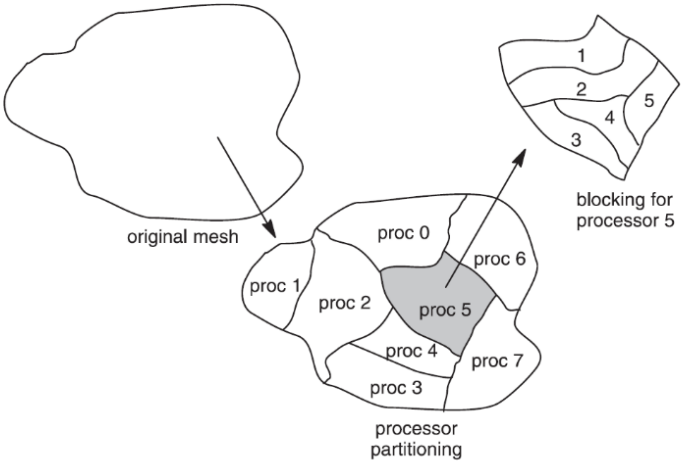

Fig. 3.4 – Two level partitioning scheme. Mesh is first partitioned into subdomains for the processors, then each subdomain is further divided into blocks of elements with the same type, constitutive model, etc. Source: (GULLERUD & DODDS JR, 2001).

In a very complete work, (GULLERUD & DODDS JR, 2001) detail several aspects regarding parallelization of the implementation, which was performed in WARP3D, a research code developed by the aforementioned authors. According to them, "*the WARP3D architecture employs a manager-work approach to organize and drive parallel execution. The manager (root) processor serves as the controller for the computations by conducting necessary serial calculations and initializing parallel computation through notification of the worker processors. The worker processors, when not conducting computations, wait for the manager processor to initiate a new set of calculations. Each processor stores the data for the elements within its domain, and conducts the corresponding local element calculations, including tangent stiffness computation, strain/stress/internal force resolution, contact evaluation. The manager processor*

computes and stores most nodal quantities (e.g. the applied load vector). For nodal quantities derived from element values (e.g. diagonal terms of structure stiffness), each processor stores values for all nodes connected to the elements within its domain and employs special MPI communication datatypes to update values for nodes shared between processors. Every processor stores the basic geometry data for the model (e.g. node coordinates, element connectivity, displacement constraints). The manager processor currently conducts all input activities while each worker processor generates output for the domain data". The aspects involving possibility of parallelization and communication requirements for a load increment are displayed in Fig. 3.5.

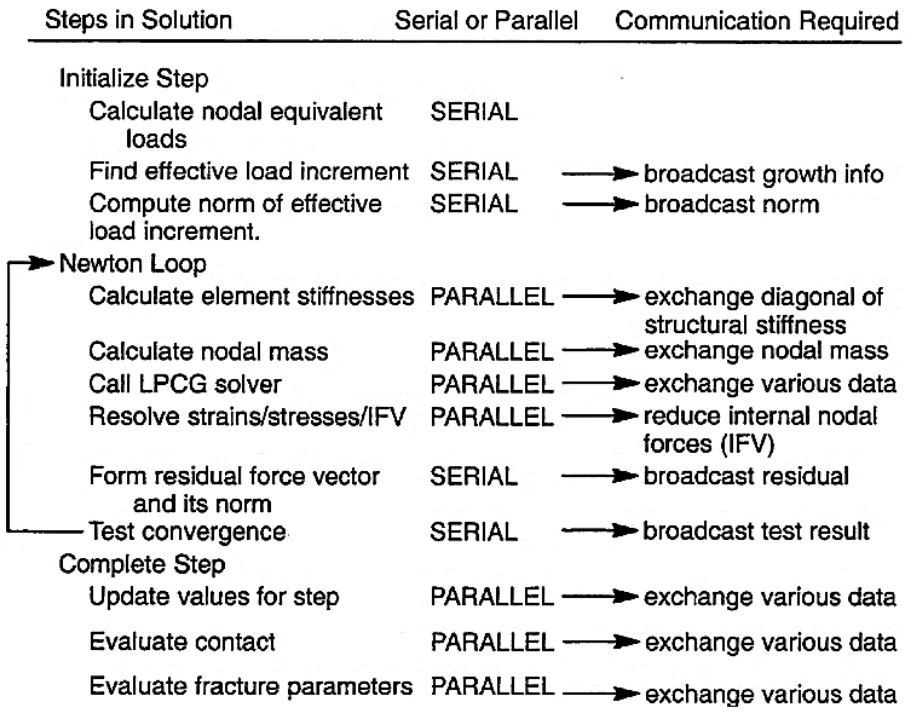

Fig. 3.5 – Parallel solution of a load increment. Source: (GULLERUD & DODDS JR, 2001).

The EBE preconditioned conjugate gradient algorithm implemented by (GULLERUD & DODDS JR, 2001) is summarized in Table 3.10. With respect to the preconditioning matrix, $\boldsymbol{C}$, they implemented and tested the diagonal (Jacobi) and the Hughes-Winget (HW) versions. (GULLERUD & DODDS JR, 2001) also proposed a new parallel implementation of the Hughes-Winget EBE preconditioner, *that couples an unstructured dependency graph with a new balanced graph-coloring algorithm to schedule parallel computations within and across domains*.

Table 3.10 – EBE Preconditioned Conjugate Gradient Algorithm.

Linear System: $\boldsymbol{K_T\, x = R}$

Step 1 - Initialization:

1. $\boldsymbol{k = 1}$, $\boldsymbol{x_0 = x_{predicted}}$
2. for $\boldsymbol{j = 1}$, ... $\boldsymbol{N_{eq}}$
3. if $\boldsymbol{j}$ is a constrained dof,
4. $\boldsymbol{r_j = 0}$
5. else
6. $\boldsymbol{r_j = R_j}$
7. end if
8. end for

Step 2 - Iterations:

1. for $\boldsymbol{k = 1, 2, 3}$, ...
2. $\boldsymbol{z_{k-1} = M^{-1}\, r_{k-1}}$
3. $\boldsymbol{\beta_k = \frac{z_{k-1}^T r_{k-1}}{z_{k-2}^T r_{k-2}}}$ $\qquad (\boldsymbol{\beta_1 = 0})$
4. $\boldsymbol{p_k = z_{k-1} + \beta_k\, p_{k-1}}$ $\qquad (\boldsymbol{p_0 = 0})$
5. $\boldsymbol{\alpha_k = \frac{z_{k-1}^T\, p_{k-1}}{p_k^T\, K_T\, p_k}}$
6. $\boldsymbol{x_k = x_{k-1} + \alpha_k\, p_k}$
7. $\boldsymbol{r_k = r_{k-1} - \alpha_k K_T\, p_k}$
8. end for

Step 3 - Convergence Check:

1. if $\|\boldsymbol{r_k}\| \leq$ tolerance
2. Solution converged
3. else
4. if $\boldsymbol{k >}$ **max. iteration limit**
5. Solution did not converged
6. else
7. $\boldsymbol{k = k + 1}$
8. Return to Step 2
9. end if
10. end if

Source: (GULLERUD & DODDS JR, 2001).

where:

- k – is the iteration count;
- $\boldsymbol{K_T}$ – is the global stiffness matrix;
- $\boldsymbol{r}$ – is the linear residual;
- $\boldsymbol{x}$ – is the trial displacement vector;
- $\boldsymbol{M}$ – is the preconditioning matrix;
- $\boldsymbol{p}$ – denotes the step direction;
- α – is the step length;
- β – defines the correction factor.

In this implementation, the calculation of the preconditioning matrix $\boldsymbol{M}$ and the matrix vector product $[\mathbf{K}_\mathrm{T}]\{\mathbf{p}_\mathrm{k}\}$ are responsible for most of the computation time. As in the EBE formulation the global stiffness matrix assembling is avoided, the contributions of each element $\left(\boldsymbol{K}_T^{(e)}\,\boldsymbol{p}_{k\,(e)}\right)$ are summed together to form the global result.

According to (GULLERUD & DODDS JR, 2001), "*the parallel version of the LPCG algorithm uses a domain decomposition of both element and nodal data. Processors own all relevant data for elements in their domain and the data for internal nodes; data is shared between processors for nodes on domain boundaries. Between each step in the LPCG algorithm, communications synchronize terms for nodes on the boundary between domains for nodal vectors. Fig. 3.6 illustrates the computation of the matrix-vector product* $[\boldsymbol{K}_T]\{\boldsymbol{p}_k\}$ *using the previously described blocking of the elements. A gather-compute-scatter cycle, as shown within the shaded region, defines the computational kernel. This process collects terms required for the multiplication of an entire block into data structures contiguous in memory. The blocking approach provides a simple means to tune for optimal cache memory utilization by altering block sizes and takes full advantage of platforms that provide vector processors*".

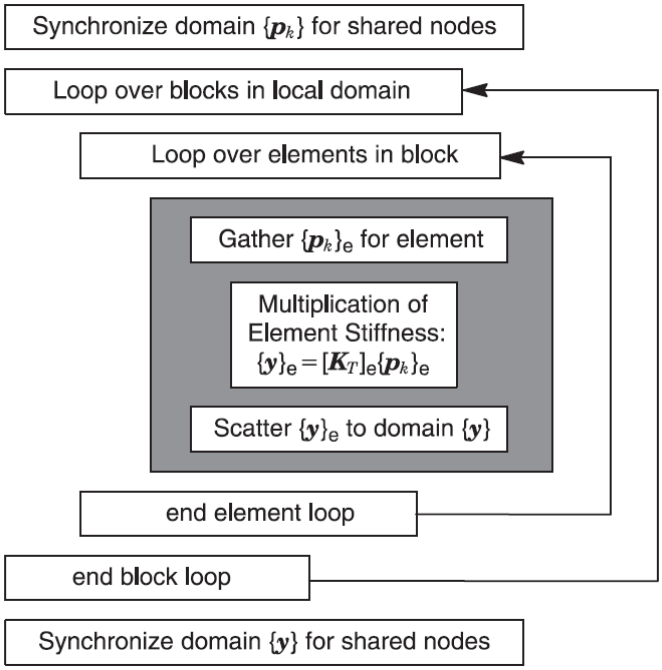

Fig. 3.6 – Multiplication of $[K_T]\{p_k\}$ for a block of elements. Source: (GULLERUD & DODDS JR, 2001).

(GULLERUD & DODDS JR, 2001) have presented by far the most detailed and didactic implementation of the EBE method. They have discussed several aspects regarding parallelization, synchronization and how the gather and scatter operations should be performed in order to achieve a better performance. In addition, they used MPI to parallelize their solution, which is still widely used in the academy and industry for parallelization in distributed processing hardware. Their domain subdivision methodology is also fully applicable to the finite element problem of a flexible pipe, which, for its simplified cylindrical element, has simplified logics of domain subdivision, discarding the use of dedicated auxiliary software (such as METIS) for this. Therefore, this very efficient strategy adopted by (GULLERUD & DODDS JR, 2001) will be the basis for the development of a parallel implementation customized to the finite element analysis of a flexible pipe.

(LIU, ZHOU, & YANG, 2007) developed a distributed memory parallel EBE scheme for tridimensional finite element analysis that employs the Jacobi preconditioned conjugate gradient method. The flowchart from Fig. 3.7 illustrates the implemented procedure to parallelize the solution, in which each processor stores data and performs calculation only to the elements assigned to them.

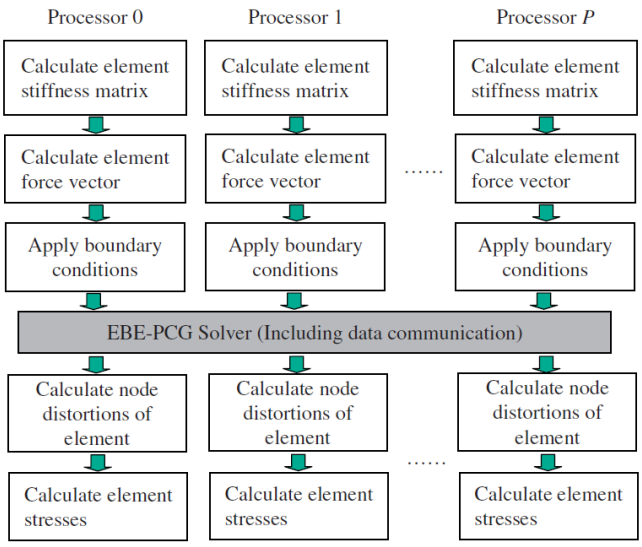

Fig. 3.7 – Flowchart of FEM method based on EBE policy. Source: (LIU, ZHOU, & YANG, 2007).

More recent works, for example (KISS, GYIMOTHY, BADICS, & PAVO, 2012), (KISS, BADICS, GYIMOTHY, & PAVO, 2012), (MARTÍNEZ-FRUTOS, MARTÍNEZ-CASTEJÓN, & HERRERO-PÉREZ, 2015) and (MARTÍNEZ-FRUTOS & HERRERO-PÉREZ, 2015), have focused on GPU applications to the EBE method, extrapolating the scope of this work.

4 PipeFEM

In general, the final solution of the system of equations is the main bottleneck of a finite element simulation. However, when dealing with large scale models, it is important to look for a balanced tool in terms of computational performance, since other processes may also become bottlenecks, such as geometry creation, mesh generation and element stiffness matrices computation. In this context, aiming the simulation of large scale models of flexible pipes, a new analysis tool was developed, named as PipeFEM, entirely written in C++ and that explores parallelism in the geometry and mesh generation and in the numerical solution. Its functioning follows the standard flowchart of the finite element method, illustrated in Fig. 4.1. After the numerical solution, the program exports the nodal results into an output ".txt" file, which is then used to post process the results and to generate the displacements graphs.

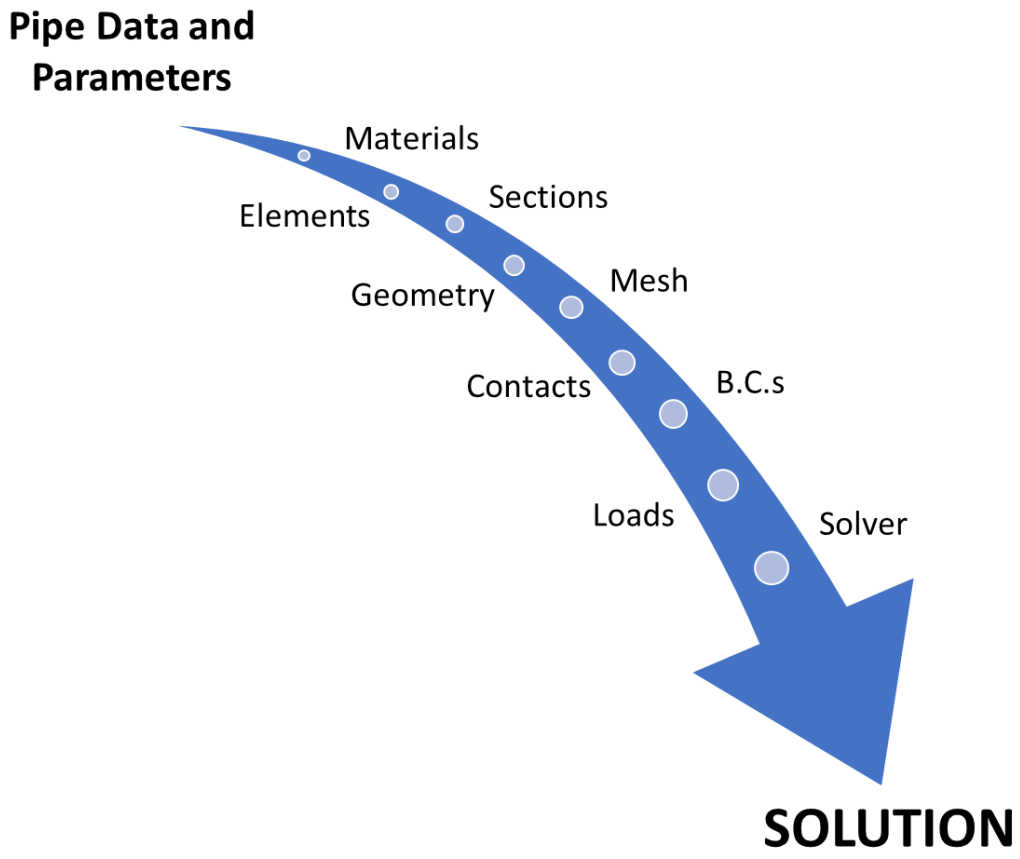

Fig. 4.1 – Flowchart of PipeFEM. Source: own authorship.

In order to implement the flowchart of Fig. 4.1, it was necessary to develop a series of additional modules and libraries, shown in Fig. 4.2.

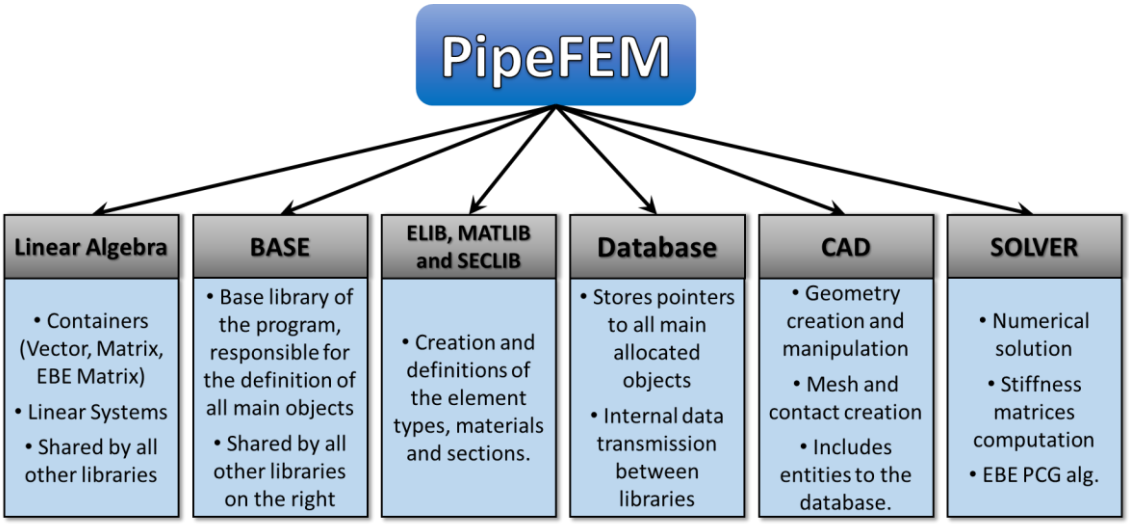

Fig. 4.2 – Libraries that compose the PipeFEM program. Source: own authorship.

A **Linear Algebra** library was developed for PipeFEM. It includes a series of containers for data storage and manipulation, such as *Vector* and *Matrix*, and which are presented in greater detail in Chapter 5. The linear algebra library also contains the EBE-PCG algorithm, presented in Chapter 9.

As the name suggests, **Base** is responsible for the definition of most of the objects of the program, including, for instance, geometric and mesh entities, coordinate systems (CSYS), among others. For this reason, it can be considered the base library of the program and directly employed by the following ones. The most important items of the BASE library are presented in detail in chapters 6 and 7.

Aiming high computational performance, the element definitions of MacroFEM were converted to the C++ language, parallelized, optimized and implemented in the element library, **ELIB**. This library also contains a useful feature, called "*element type*", which is responsible for standardizing and systematizing the storage of user-defined element parameters, besides acting as element allocators during mesh generations. The materials and sections definitions were also converted from MacroFEM and implemented in the **MATLIB** and **SECLIB** libraries, respectively. These three libraries are presented in the following sections.

Database library is responsible for the definition of the *"database"* object, which is employed to store and transmit data in an organized and encapsulated way. Basically, it

consists of a structured container of pointers to all entities that comprise the model (which are presented in the next chapters). It also facilitates data manipulation, by allowing the selection of specific items or the iteration along all items of a desired type. In addition, the *database* object provides all necessary statistics of the model, such as the total number of nodes or elements.

CAD library is responsible for the construction of the flexible pipe models. It provides methods for geometry creation and manipulation, and is also responsible for mesh generations and contact definitions between layers of the pipe. It is also responsible for applying the loads and boundary conditions to the modeled flexible pipe. For these reasons, it is the library with the highest degree of interaction with the user. Internally, the CAD library allocates the objects defined in the BASE library and includes them to the *database* object.

Solver, fully described in Chapter 7.3, is responsible for the numerical solution of the problem. The complete model data is transmitted from CAD to the solver through the *database* object. The solver is responsible for computing the element stiffness matrices, global degrees-of-freedom numbering and for employing the developed EBE-PCG algorithm.

4.1 ELIB – Element Library

ELIB implements the finite macroelements from chapter 2. It is subdivided into two branches, that are related to each other:

- **Elements:** responsible for the definitions of the element objects;
- **Element Types (or Element Allocators):** stores user-defined parameters regarding the finite elements and works as their allocators.

4.1.1 Element

The finite elements are the basis of the method. Each element object stores its nodal connectivity, material, section and other parameters, which are later used to compute the element stiffness matrices and to relate the global and local connectivity basis. Aiming higher computational performance, the finite macroelements from MacroFEM were converted to the C++ language, parallelized, optimized and implemented in PipeFEM using polymorphism. The derived classes from the main class Element are illustrated in

Fig. 4.3. Element, as well as the classes that directly derive from it, are abstract classes. This allows the manipulation of the objects from the lowest level derived classes as if they were Element objects.

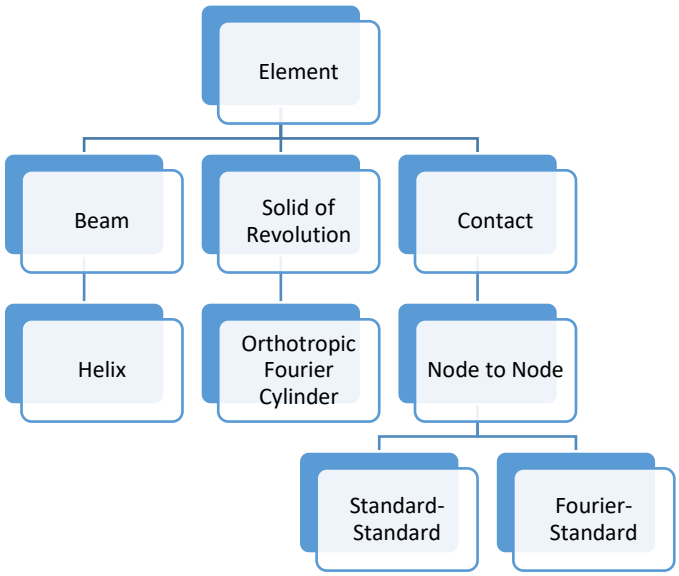

Fig. 4.3 – Finite elements. Source: own authorship.

The branch *Beam* includes the beam elements, in special the helical element; while the branch *Solid of Revolution* is responsible for the Fourier cylindrical element; and finally Contact relates to the implementation of the contact elements.

4.1.2 Element Type

As each derived element has its own *Element Type* object, both of them follow exactly the same hierarchy from Fig. 4.3. Element Type has two important functions. The first is to store both intrinsic and user-defined parameters from each type of element, such as the number of degrees-of-freedom, nodal type (Standard or Fourier), the element Fourier order, element applicability (if it can be used to mesh lines or areas, for example), and others. The second function is to act as an element allocator: during the mesh methods, the element type is employed to allocate new elements, which can be done in a concise way, since all necessary parameters and properties are already encapsulated in it.

4.2 MATLIB – Material Library

MATLIB is responsible for the material definitions. The hierarchical relationships of the material objects are illustrated in Fig. 4.4. So far, only linear elastic materials are contemplated, although the basis for a future implementation of nonlinear materials has been left.

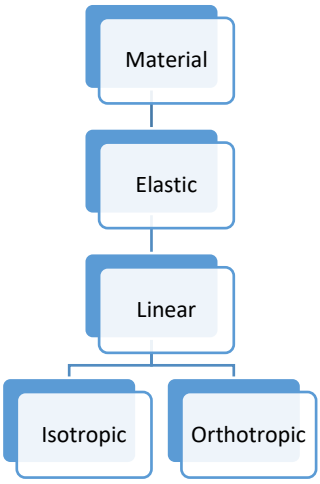

Fig. 4.4 – Material library. Source: own authorship.

4.3 SECLIB – Section Library

SECLIB is responsible for the sections definitions, by creating their objects and storing the user-defined section parameters, being a relatively simple class in comparison to the other. It is subdivided into circular and rectangular beam cross-sections, as illustrated in Fig. 4.5. Object-oriented concepts were applied in order to facilitate future inclusions of new types of cross-sections.

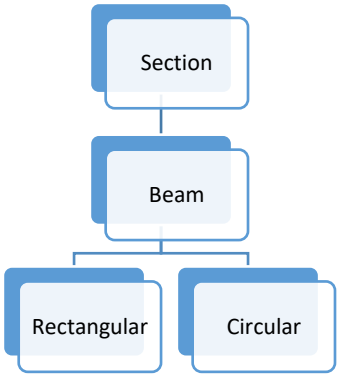

Fig. 4.5 – Section library. Source: own authorship.

5 Data Containers

In order to store and manipulate data in PipeFEM in an efficient way, it was necessary the development of the following data containers:

- **Vector:** a single-dimensional data container;
- **Matrix:** a two-dimensional data container;
- **Symmetric Matrix:** a symmetric two-dimensional data container;
- **EBE Matrix:** a two-dimensional data container composed of several other smaller matrices.

When designing these containers, generic programming techniques of C++ were explored by employing *Templates*, in which the developed code is independent of any particular type. In other words, the code is generic and can be applied to different types. Vector, for instance, can be used to store from simple types such as *int* and *double*, to more complex objects (geometry, mesh and element objects), pointers to objects and so on.

5.1 Vector

Vector is a sequential container that can stores n user-defined objects of the same type. It is dynamically and contiguously allocated, i.e., the memory is allocated in consecutive address memory blocks, making the iteration process a very fast and efficient procedure. All the work relative to memory allocation and management is encapsulated within the vector object, making very simple its usage.

Resizing a vector can be an expensive procedure. When the desired size exceeds the total amount of allocated memory, there is no alternative other than to dynamically allocate a new array and in sequence move the contents of the old array to the new one. In order to minimize its cost, additional memory is also pre-allocated, as illustrated in Fig. 5.1, being possible the rapid inclusion of some new elements. With an acceptable additional memory cost, this strategy reduces significantly the total number of resizing operations, bringing performance to the implementation. The amounts of allocated and

pre-allocated memories can be controlled by the user through the *resize* and *reserve* methods, respectively.

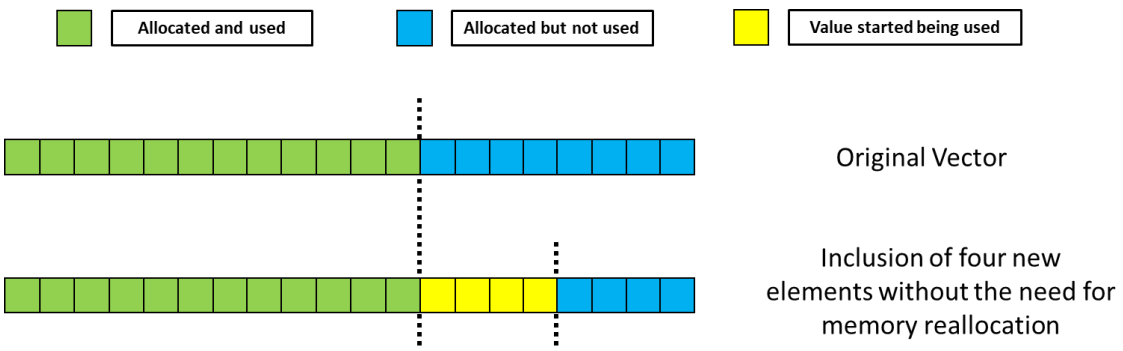

Fig. 5.1 – Vector memory management. Source: own authorship.

The functioning of the implemented Vector container is very similar to the available in the C++ Standard Library (STD). Nevertheless, it was decided to develop a proprietary vector template instead of using the one from STD due to the following reasons: absolute control of memory allocation (especially in "*resizing*" and "*push back*" operations), since the STD version uses its internal allocator, whose behavior is not completely clear in the specifications, which can be harmful for parallel applications; and also due to some observed unexpected behaviors of the STD version for parallel manipulations of large-scale vectors. Additionally, the implemented Vector explores an interesting feature of the C++ language called "*template specialization*", which allows the customization of the algorithm for a particular type. In this case, it was done a specialization for the *double* type, and specific methods (such as vector norms) and mathematical operators were implemented.

5.2 Matrix

Matrix template is an extension of the concept of vector to the two-dimensional field, in which the stored data is accessed via row and column indexes.

In its first implementation, a double starred pointer was used to allocate and manipulate the memory, as exemplified in Fig. 5.2 for the *double* type. In this case, an array of pointers is dynamically allocated. Then, for each of its elements, a new array is dynamically allocated. This approach has the advantage of the direct indexing (for instance, the position (2,3) in the matrix is accessed directly by $p[2][3]$), as shown in

Fig. 5.3, but leaves data scattered in memory, since each row of the matrix is allocated independently, losing memory contiguity.

```
int n, m;

double** p = new double*[n];

for (int i = 0; i < n; i++)
{
    p[i] = new double[m];
}
```

Fig. 5.2 – Example of double starred pointer for matrix allocation. Source: own authorship.

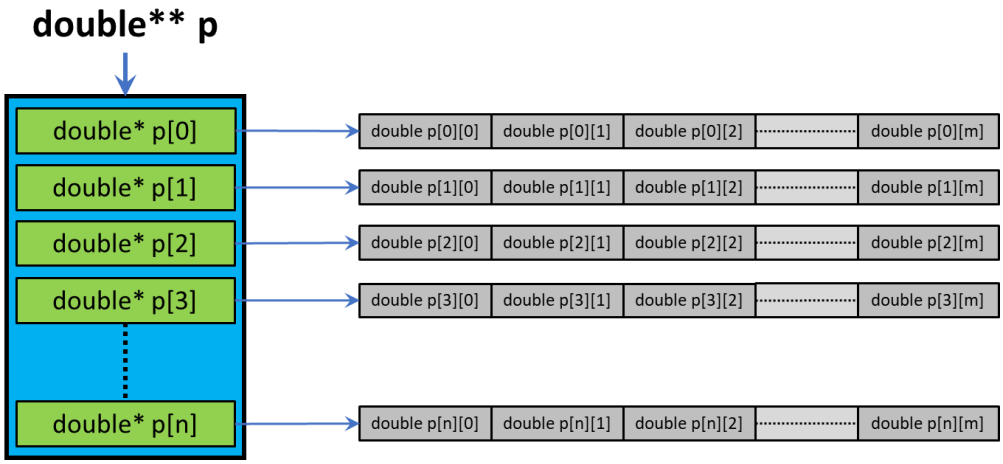

Fig. 5.3 – Memory hierarchy and indexing for the double starred pointer allocation. Source: own authorship.

This way, in order to take advantage of the benefits of contiguous memory allocation, the single array methodology was employed, as illustrated in Fig. 5.4. In this case, a single large array is dynamically allocated, concatenating all the lines of the matrix. The position in memory of the pair (i, j) is given by mathematical operation:

$$Memory\ position\ of\ A(i,j) = i\ .\ n_{columns} + j \qquad \text{Eq. 5.1}$$

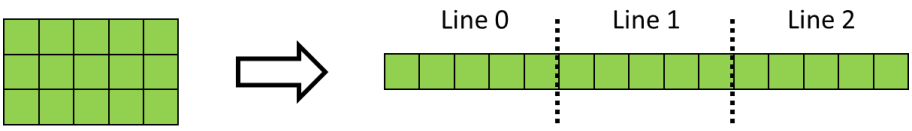

Fig. 5.4 – Single array scheme of storage. Source: own authorship.

In order to optimize the resizing capability of the matrix, it was developed a procedure of additional pre-allocated memory, as illustrated in Fig. 5.5. In this example, the useful size of the matrix is $(3x3)$, but the pre-allocated size is $(5x8)$. It means that up to two rows or four columns can be added without any memory reallocation.

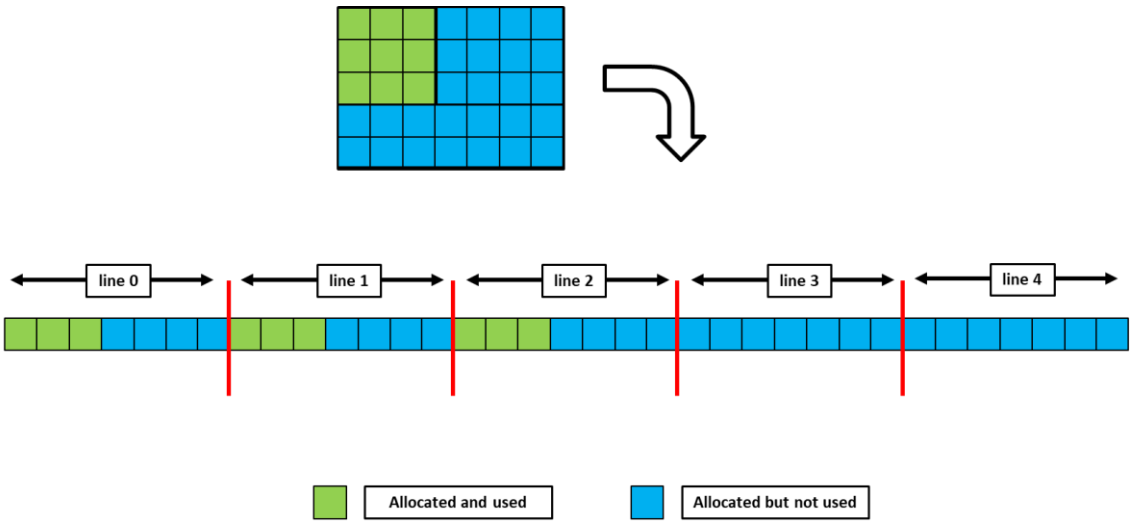

Fig. 5.5 – Memory management for fast resizing capability. Source: own authorship.

Fig. 5.6 shows the pre-allocated memory regions that start being used when a new line is added to the matrix.

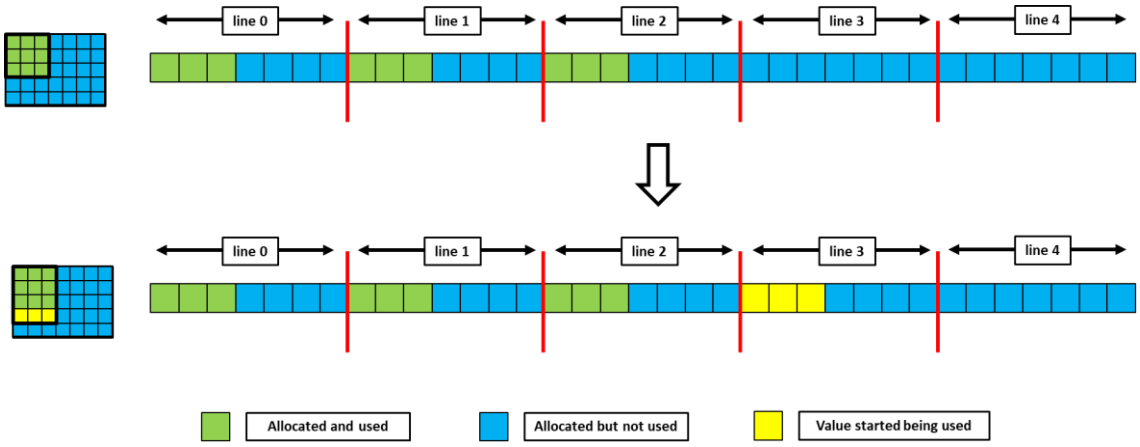

Fig. 5.6 – Addition of a new line. Source: own authorship.

Fig. 5.7 shows the pre-allocated memory regions that start being used when a new column is added to the matrix.

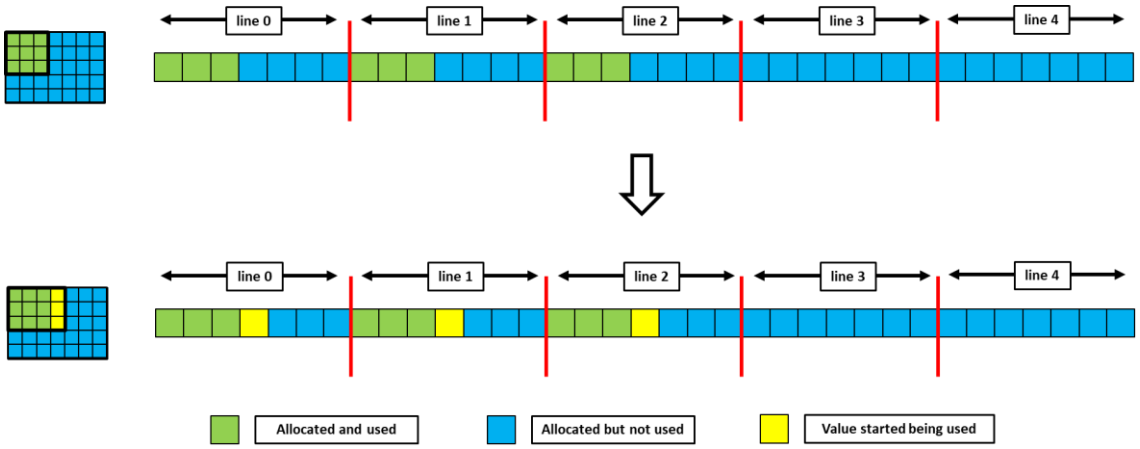

Fig. 5.7 – Addition of a new column. Source: own authorship.

In this scheme, the position in memory does not depend on the number of columns of the matrix, but on the total number of pre-allocated columns and is given by:

$$Memory\ position\ of\ A(i,j) = i\ .\ n_{pre-alloc.\ columns} + j \qquad \text{Eq. 5.2}$$

Although this methodology reduces the number of memory reallocations, there are still necessary situations, as the exemplified in Fig. 5.8.

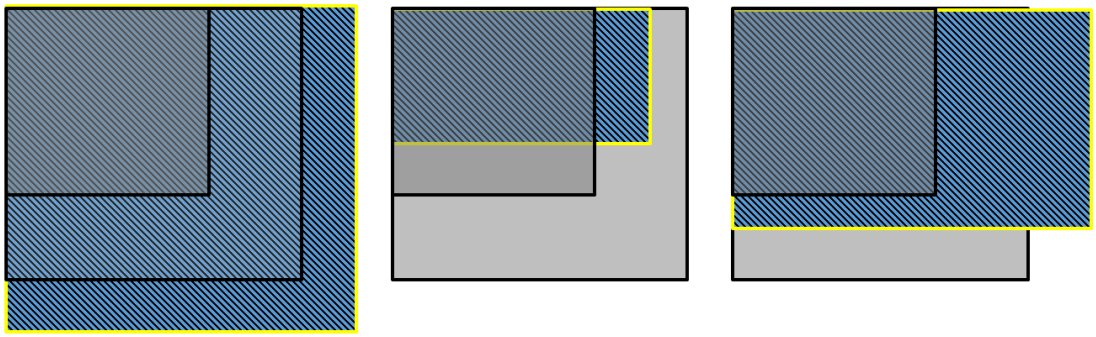

Fig. 5.8 – Examples of resizing cases with necessary memory reallocation. Source: own authorship.

Fig. 5.9 shows the procedure for the addition of four pre-allocated lines. In this case, a new array must be dynamically allocated and after that the useful entries of the old array must be moved to this new one.

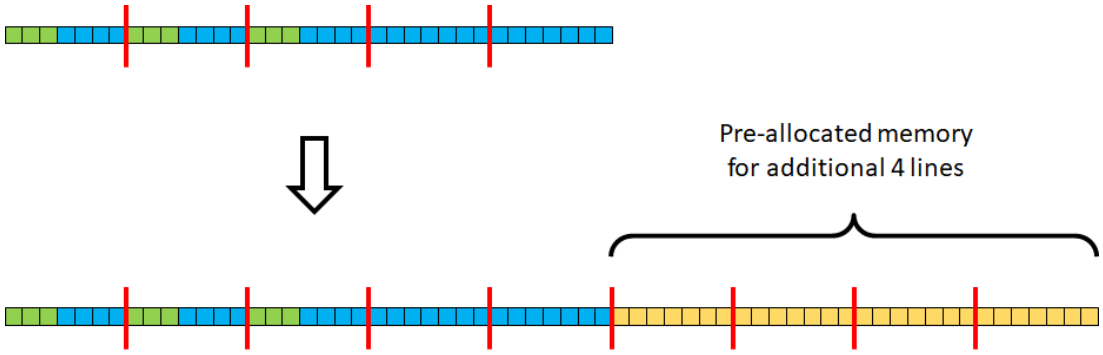

Fig. 5.9 – Memory management for additional pre-allocated lines. Source: own authorship.

Analogous, Fig. 5.10 shows the procedure for the addition of two pre-allocated columns, and Fig. 5.11 shows the procedures for the inclusion in both directions.

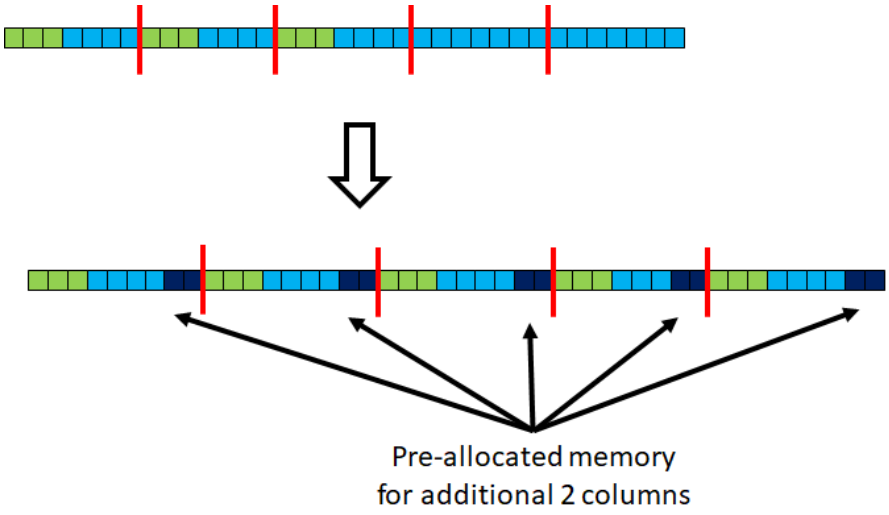

Fig. 5.10 – Memory management for additional pre-allocated columns. Source: own authorship.

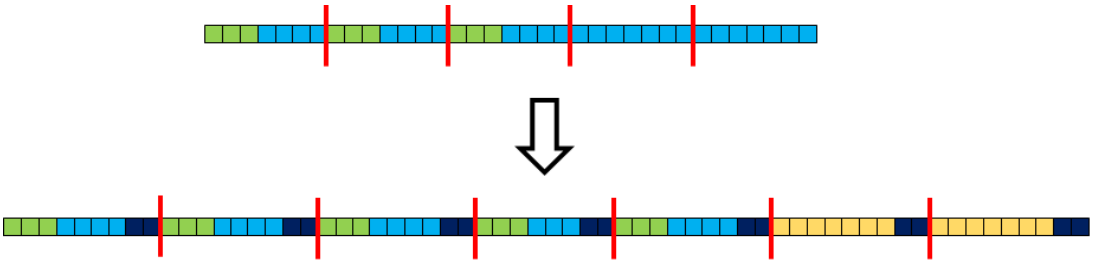

Fig. 5.11 – Memory management for both additional pre-allocated lines and columns. Source: own authorship.

Template specialization was also explored for the *double* type, enabling the implementation of specific methods (such as matrix determinant) and numerical operators (such as matrix-vector and matrix-matrix products).

5.3 Symmetric Matrix

The Symmetric Matrix is a particular case of the Matrix, in which the following relation is always valid:

$$A(i,j) = A(j,i) \qquad \text{Eq. 5.3}$$

This property is explored to reduce memory consumption in almost 50%, since the symmetric values are stored only once. As illustrated in Fig. 5.12, a single array is dynamically allocated to store the elements of the symmetric matrix.

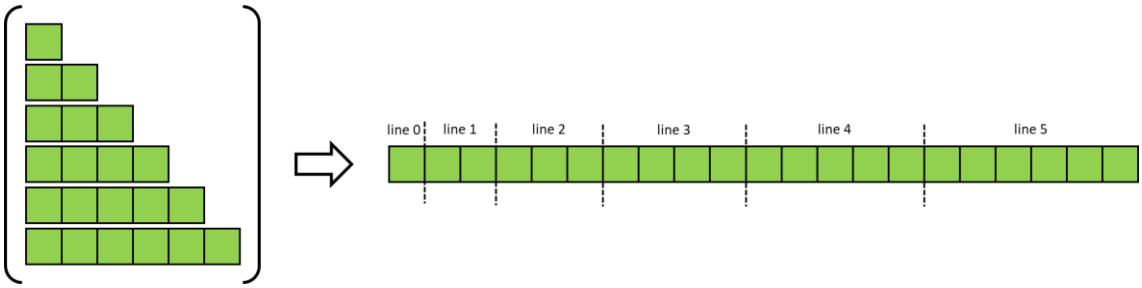

Fig. 5.12 – Single array memory allocation. Source: own authorship.

The memory indexing is given by the arithmetic progression from Eq. 5.4. In the case of accessing an element situated above the main diagonal, the values of i and j are swapped, as shown in this equation.

$$Memory\ position\ of\ A(i,j) = \begin{cases} j + \dfrac{i\,.\,(1+i)}{2}, & i \geq j \\ i + \dfrac{j\,.\,(1+j)}{2}, & j > i \end{cases} \qquad \text{Eq. 5.4}$$

As can be seen in Eq. 5.4, every indexing operation requires a costly "*if/else*" operation to verify if the input pair is situated below or above the main diagonal. This results in a trade-off between improved memory consumption and higher indexing costs.

If the number of indexing operations is very high during the manipulation of the stored data, it might be better to use conventional matrices instead of the symmetric ones.

In order to minimize this indexing costs, the implementation counts also with non-safe accessing methods and operators that do not perform this "*if/else*" verification, giving the user full responsibility for their correct use.

The development of customized algorithms was also another explored alternative to improve the efficiency of the symmetric matrix implementation, which is clear in the following example, the product operation between two symmetric matrices. This operation is very important for finite elements, since it can be performed several times during the evaluation of each of the element stiffness matrices depending on the formulations. The matrix-matrix product is a costly operation, for both the high number of mathematical operations and memory accesses. However, as the indexing follows well-defined patterns, it was possible to develop an optimized logic using only indexes pairs situated exclusively below the main diagonal, with direct access and eliminating all "*if/else*" verifications. It is illustrated in Fig. 5.13 and Fig. 5.14, and the complete algorithm is found in Table 5.1.

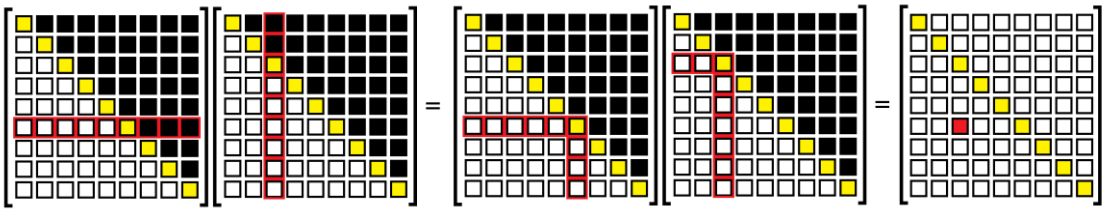

Fig. 5.13 – Cache optimized product between two symmetric matrices, for (i, j) indexes where $i \leq j$.

Source: own authorship.

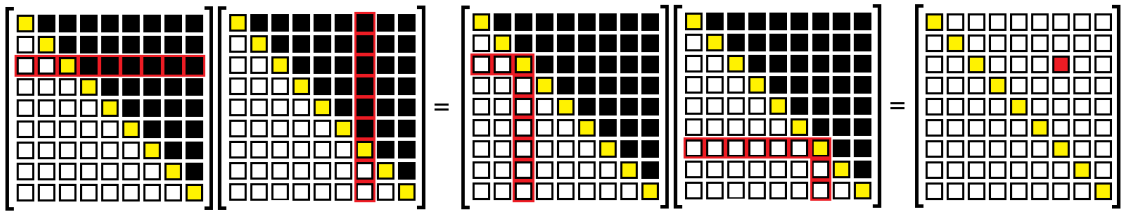

Fig. 5.14 – Cache optimized product between two symmetric matrices, for (i, j) indexes where $j > i$.

Source: own authorship.

Table 5.1 – Optimized algorithm for the product between two symmetric matrices.

Conventional Algorithm	*Optimized Algorithm*

`for (int i = 0; i < n; i++)`
`{`
`  for (int j = 0; j < n; j++)`
`  {`
`  for (int k = 0; k < n; k++)`
`  C(i, j) = A(i, k) * B(k, j);`
`  }`
`}`
	`// BELOW MAIN DIAGONAL`
	`for (int i = 0; i < n - 1; i++)`
	`{`
	`  for (int j = 0; j <= i; j++)`
	`  {`
	`  for (int k = 0; k < j; k++)`
	`  C(i, j) += A(i, k) * B(j, k);`
	`  for (int k = j; k <= i; k++)`
	`  C(i, j) += A(i, k) * B(k, j);`
	`  for (int k = i + 1; k < n; k++)`
	`  C(i, j) += A(k, i) * B(k, j);`
	`  }`
	`}`
	`// ABOVE MAIN DIAGONAL`
	`for (int i = 0; i < n; i++)`
	`{`
	`  for (int j = i + 1; j < n; j++)`
	`  {`
	`  for (int k = 0; k < i; k++)`
	`  C(i, j) += A(i, k) * B(j, k);`
	`  for (int k = i; k <= j; k++)`
	`  C(i, j) += A(k, i) * B(j, k);`
	`  for (int k = j + 1; k < n; k++)`
	`  C(i, j) += A(k, i) * B(k, j);`
	`  }`
	`}`

Source: own authorship.

In order to evaluate the efficiency of the implementations, a parametric analysis of the simulation time of the matrix-matrix product in function of the size of the matrix for four different methodologies is available at Fig. 5.15. These results show that the optimized logic for symmetric matrices (Table 5.1) have reduced the simulation time by 11% in comparison to the standard algorithm. However, despite these advances in performance, the product between two symmetric matrices is still slower than the product between two standard (non-symmetric) matrices, especially with the increase of the dimensions of the involved matrices. The choice of the most appropriate container depends then on how the matrix is used. If the purpose is merely to store data, then the symmetric container is the best alternative, for its lower memory consumption. However, if the symmetric matrix is very used in the product with other matrices or vectors, then it is more efficient to store it through a standard matrix container.

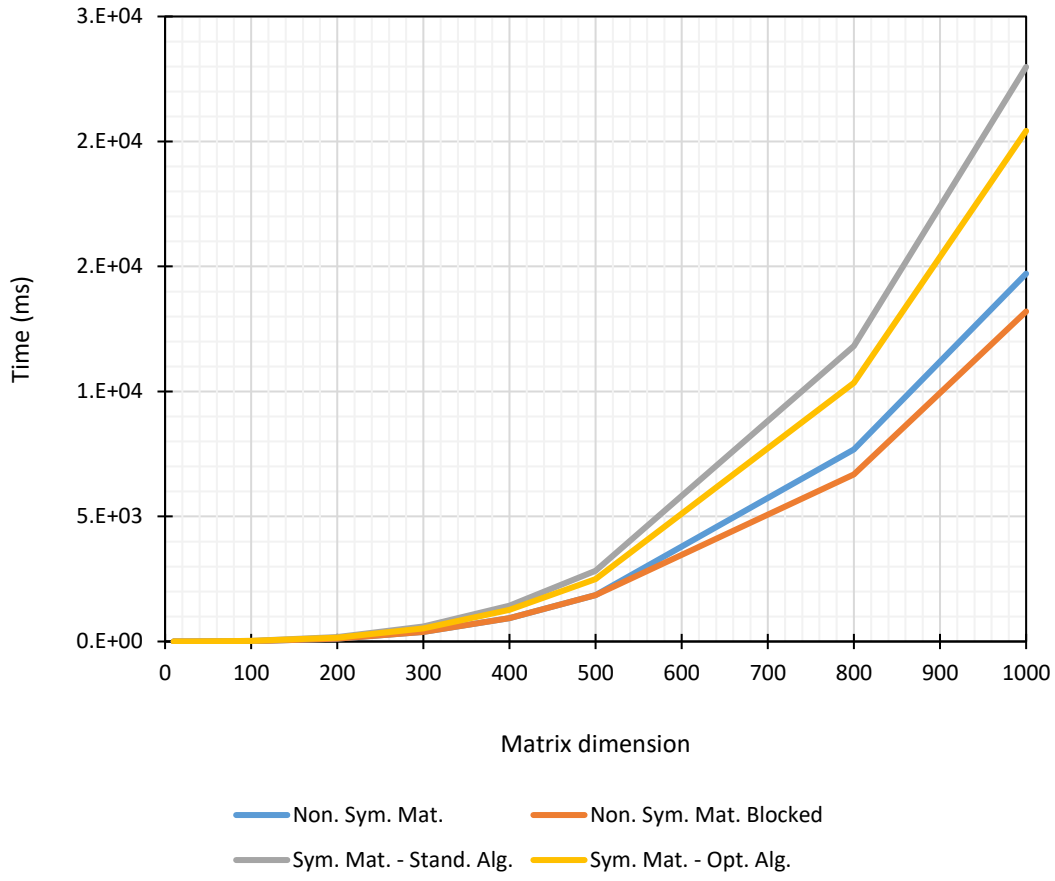

Fig. 5.15 – Performance comparison of the product between two matrices. Source: own authorship.

5.4 EBE Matrix

The element-by-element method consists of solving the linear system of equations in a local basis with the element stiffness matrices, so that the assembly of the global stiffness matrix is no longer necessary. It means that, instead of a single, very large and almost always sparse matrix, the EBE requires the manipulation of hundreds of thousands (or even millions) of small matrices. The computational performance of the EBE algorithm is directly related to how these matrices are manipulated and managed.

The previously presented containers proved to be very important and efficient in the mathematical operations of the computations of the element stiffness matrices. They could even be used directly in the EBE algorithm, through vectors of pointers to dynamically allocated matrices or vectors, for example. However, this would not be the most suitable strategy. As each matrix dynamically allocates its own array, it would result in several arrays randomly allocated in memory, losing data contiguity. In addition, the implemented algorithm would be extensive and difficult to understand.

In this way, it was decided to implement a container exclusively to the EBE method, which ensures contiguous memory allocation and also encapsulates internal operations, making the code as concise as possible and easy to use. The term *Block* was defined in this implementation as the set composed by one element stiffness matrix and one vector of integers that relates local degrees-of-freedom with the global basis.

The EBE Matrix container is created by specifying the global dimension and a vector of integers with the sizes from each block. Internally, it sums the sizes of each block and dynamically allocates a single large array, as illustrated in Fig. 5.16, ensuring thus, the contiguous memory allocation. After that, the EBE Matrix is fulfilled with each of the blocks, using a block definition method.

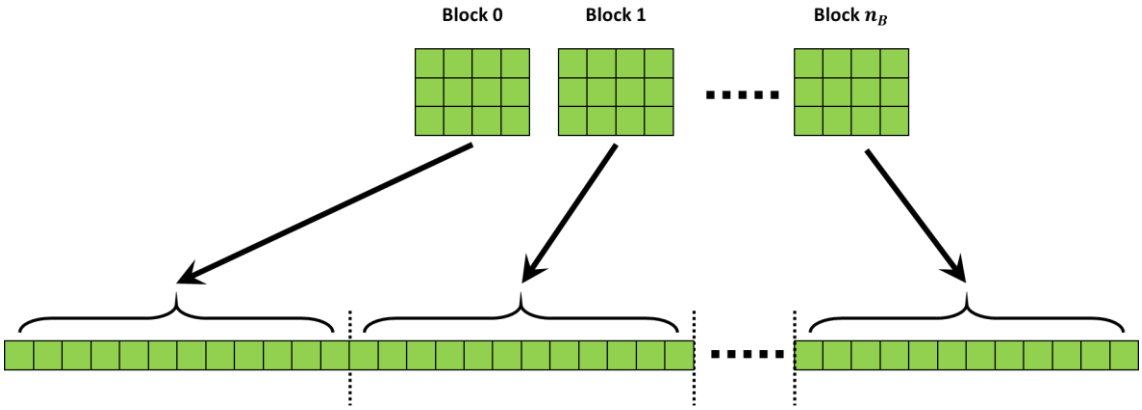

Fig. 5.16 – One single large dynamically allocated array ensures the contiguous memory allocation.
Source: own authorship.

The EBE Matrix object, Fig. 5.17, has four groups of properties:

- **Global Properties:** they are related to the global aspect of the matrix;
- **Block Properties:** properties related to or arrays of size equal to the number of blocks;
- **1D Properties:** properties related to or arrays of size equal to the sum of the dimensions of the blocks;
- **2D Properties:** properties related to or arrays of size equal to the sum of the square of dimensions of the blocks;

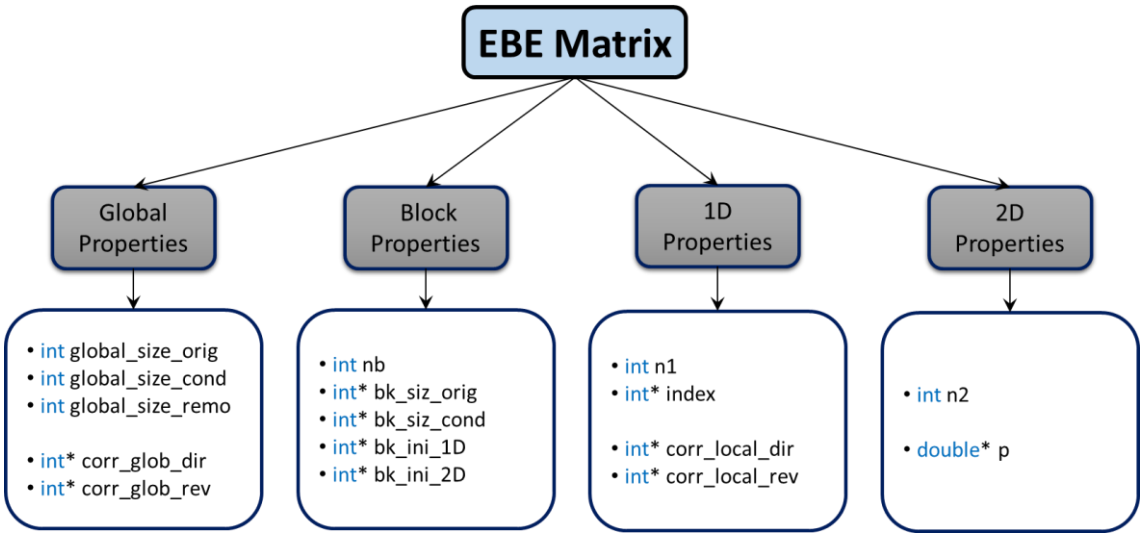

Fig. 5.17 – EBE Matrix object. Source: own authorship.

The *global properties* branch has the following items:

- ***global_size_orig*:** it is the original size of the global matrix;
- ***global_size_cond*:** it is the final size of the global matrix after the removal of the imposed degrees-of-freedom;
- ***global_size_remo*:** it specifies how many degrees-of-freedom have been removed;
- ***corr_glob_dir*** and ***corr_glob_rev*:** the renumbering of the degrees-of-freedom is necessary after the removal of the imposed ones; these two properties are arrays of integers that correlates the original with the new ordered d.o.f.s in direct and reverse way, respectively.

The *block properties* branch has the following items:

- ***nb*:** the total number of blocks;
- ***bk_siz_orig*:** an array of integers that stores the original sizes from each block;
- ***bk_siz_cond*:** an array of integers that stores the final sizes from each block (after the removal of the imposed degrees-of-freedom);
- ***bk_ini_1D*:** an array of integers that stores the 1D initial position in memory for each block, for direct indexing;
- ***bk_ini_2D*:** an array of integers that stores the 2D initial position in memory for each block, for direct indexing;

The *1D properties* branch has the following items:

- ***n1*:** the sum of dimensions of all blocks;
- ***indexes*:** a dynamically allocated array of integers that stores for each of the local degrees-of-freedom their respective and correspondent global ones.
- ***corr_local_dir*** and ***corr_local_rev*:** after the removal of the imposed d.o.f.s, a local reordering is performed on each block. These arrays of integers store the new ordering sequence in direct and reverse way, so that the reordering can be undone in the future if desired.

The *2D properties* branch has the following items:

- ***n2*:** the sum of the square of the dimensions of all blocks;
- ***p*:** a dynamically allocated array corresponding to the values of the stiffness matrices of each of the blocks.

In the finite element method, the imposed degrees-of-freedom need to be removed from the global stiffness matrix before the solution of the linear system, otherwise it would result in null determinant, and it is still necessary in the EBE method. In order to accomplish this, the first procedure is to renumber all degree-of-freedom, shifting the imposed ones to the end of the queue, as illustrated in Fig. 5.18.

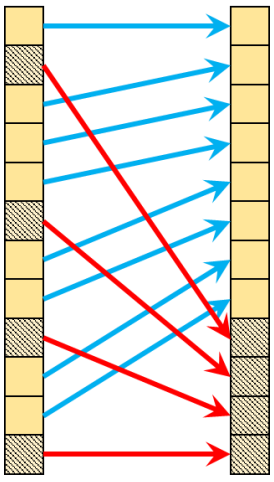

Fig. 5.18 – Global degrees-of-freedom renumbering, the imposed ones are shifted to the end of the queue. Source: own authorship.

The second procedure is to update the indexes from each block. And finally, a data rearrangement is performed for each block (including indexes and stiffness values), as

shown in Fig. 5.19. It is important to note that this image is merely illustrative, the data is concentrated in the single allocated array, so that this rearrangement operation will act in the region that corresponds to the addressed block.

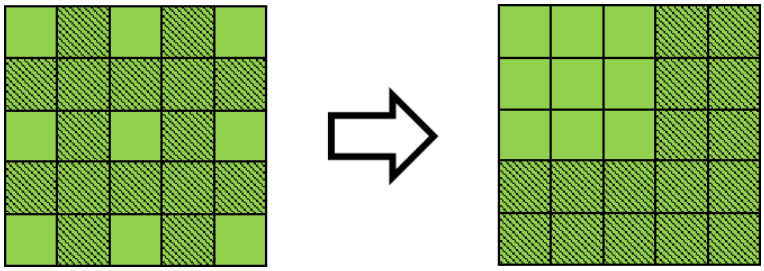

Fig. 5.19 – Data rearrangement for the block, moving to the extremities the values corresponding to the imposed degrees-of-freedom. Source: own authorship.

This data structure of EBE Matrix container is responsible for storing all blocks in an organized pattern, allowing their efficient access and manipulation. Instead of a large number of objects, the implemented EBE-PCG algorithm receives only one object, which already encapsulates and organizes all necessary data.

6 Geometry and Mesh

This chapter presents the complete implementation and data structure in what concerns geometry and mesh.

6.1 Geometry

The development of the geometry data structure was inspired by the boundary representation techniques from (STROUD, 2006). Obviously, since the objective of this work is not the development of a complete CAD, but rather a specific analysis tool for flexible pipes, it was necessary to focus only on the essential data structure and methods.

The geometry is subdivided into hierarchical levels, illustrated in Fig. 6.1. The highest one consists of volumes, which are delimited by areas, which in turn are delineated by lines, and, finally, points are the lowest level of the geometric hierarchy. Fig. 6.2 exemplifies this hierarchy for a simple cube. In the next items, each of these levels are presented in detail.

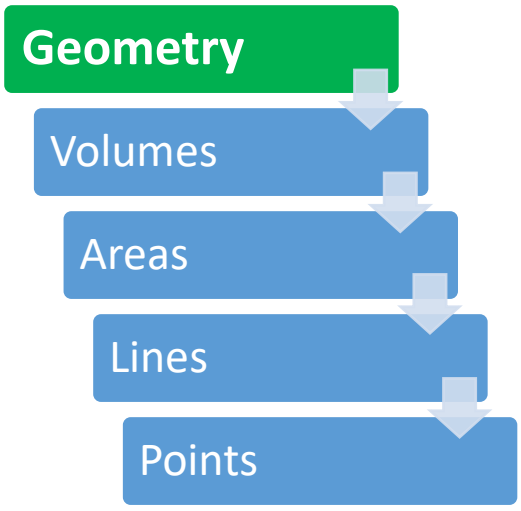

Fig. 6.1 – Hierarchical relations at the geometric level. Source: own authorship.

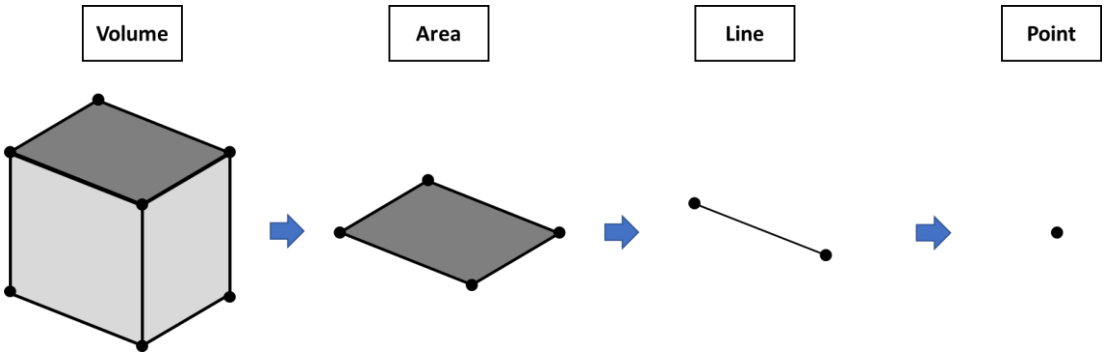

Fig. 6.2 – Volume, area, line and point of a cube. Source: own authorship.

6.1.1 Point

The point object, Fig. 6.3, is very simple to be represented, since it does not have any length, area, volume or any other dimensional attribute other than its coordinate.

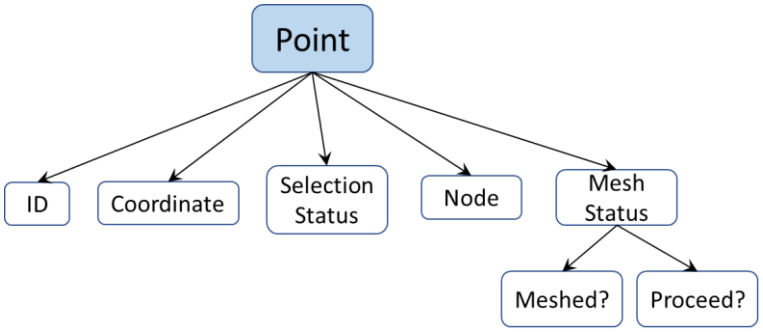

Fig. 6.3 – Point object. Source: own authorship.

The attributes of the **Point** are:

- **ID:** an integer that stores its identification;
- **Coordinate:** it is an object that encapsulates an array with the coordinates, and that converts and return the coordinates to some requested coordinate system (cartesian or cylindrical).
- **Selection Status:** used for global selection of entities;
- **Node:** a pointer that stores the address of the node attached to this point;
- **Mesh Status:** used for parallel meshing, it checks whether the point already has a node and, if not, only one thread proceeds to meshing.

6.1.2 Line

The **Line** object is illustrated in Fig. 6.4 and consists of:

- **ID:** an integer that stores its identification;
- **Length:** the line length;
- **Curve:** it is the geometric curve that defines the line (straight, arc, splines, etc.);
- **Half Lines:** the positive and negative half lines;
- **Points:** a vector of pointers to the points;
- **Mesh:** vectors of pointers to nodes, edges and elements;
- **Mesh Status:** used for parallel meshing, it checks whether the line is already meshed and for permission to proceed.

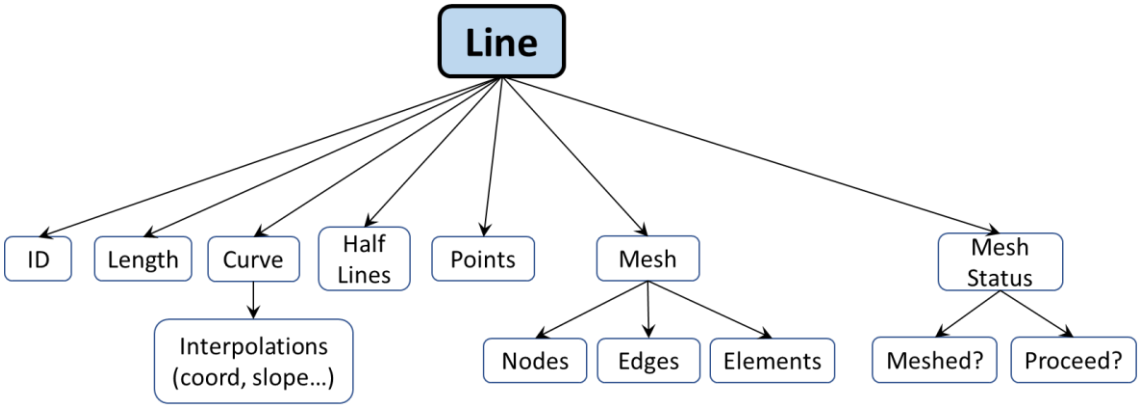

Fig. 6.4 – Abstract line object. Source: own authorship.

The curve is responsible for the geometric definition of the line. Polymorphism was employed to implement the relationships represented in Fig. 6.5. As the name suggests, the common lines are the most used, since they are composed of straight, arc and quadratic (interpolated with second order isoparametric shape functions) curves. Primitive helical curves were also necessary, since they are very used to construct the tendons of the tensile armor layers. Due to the way that it was implemented, the code is also ready to receive more complex curves, such as BSPLINES or NURBS. The specific definitions for each type of curve are encapsulated. To the line object, it does not matter how simple or complex the curve definition is, whether it is defined by just two points or by an interpolation of several, it is all encapsulated in the curve object and the polymorphism allows the manipulation between the various derived types.

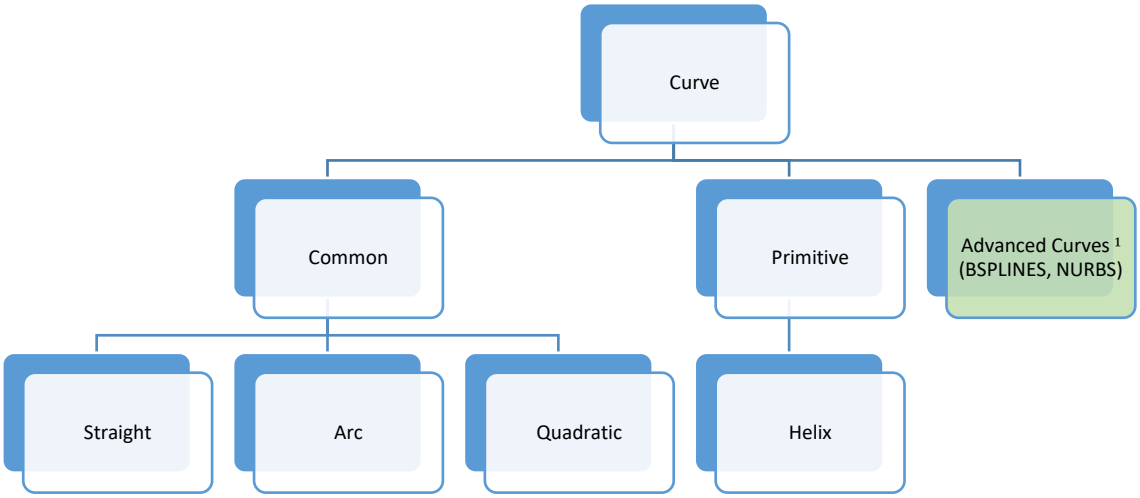

Fig. 6.5 – Class hierarchy from curve. Source: own authorship.

1: not implemented, but the code is ready to receive it.

Half-Line is an adaptation of a well-defined method in the literature called "*Half-Edges*". The change in name is justified by the fact that, in this case, the term "*edges*" is employed in the mesh context. Each line possesses a pair of half-lines, as exemplified in Fig. 6.6 for a straight line. The purposes of half-lines are: to define two orientations to a line, allowing its manipulation regardless of the orientation in which it was created; to point to other half-lines, creating well defined paths when connecting two or more lines, as illustrated in Fig. 6.7.

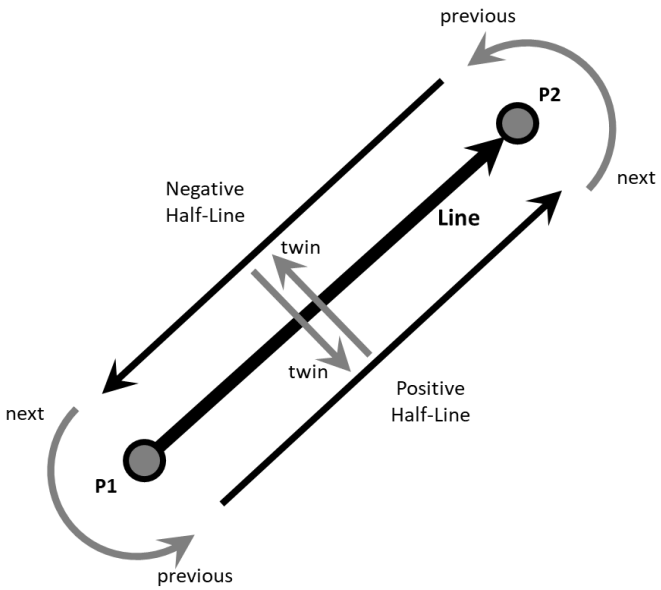

Fig. 6.6 – Half-lines indexing for a straight line. Source: own authorship.

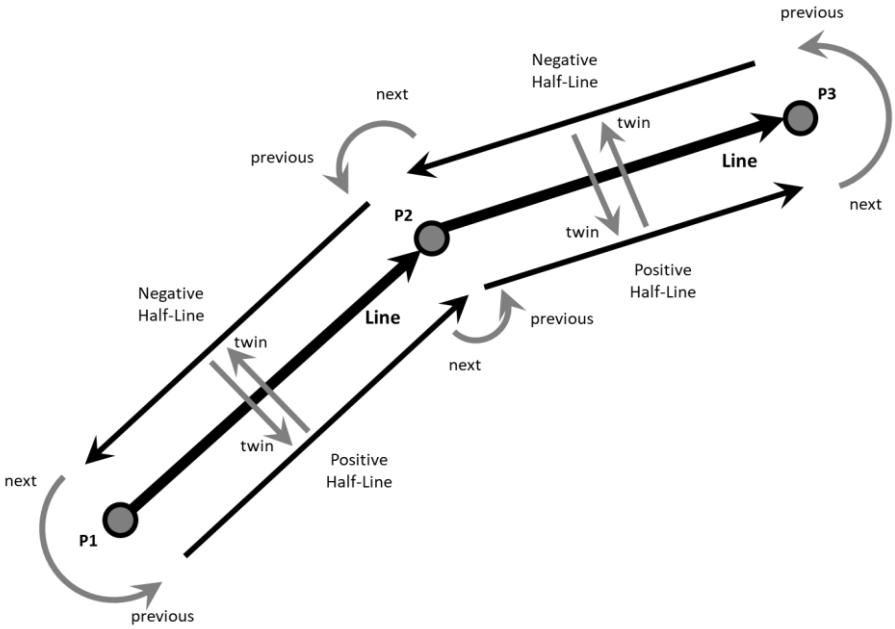

Fig. 6.7 – Indexing changes when connecting two lines. Source: own authorship.

The half-line object is shown in Fig. 6.8. In addition to the conventional twin, previous and next half-lines indexing, the implementation also incorporated geometric and mesh information, stored in the form of vectors of pointers to the objects. In this case, the positive and negative half-lines of the same line are differentiated by the fact that their vectors of pointers are defined in a reverse order in relation to the same from the other half-line, maintaining coherence in orientation and allowing the immediate iteration in both directions of the line.

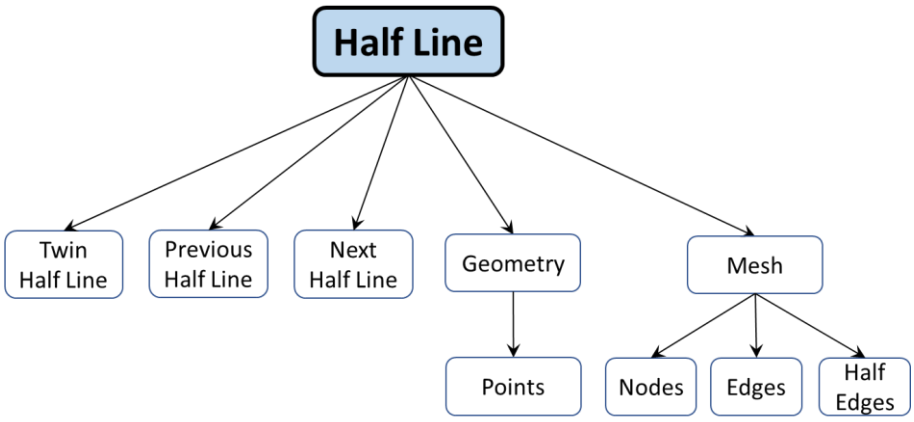

Fig. 6.8 – Half-line object. Source: own authorship.

6.1.3 Area

The Area object, Fig. 6.9, consists of:

- **ID:** an integer that stores its identification;
- **Area:** a double that stores the area value;
- **Surface:** it is the geometric surface that defines the area;
- **Half Area:** the positive and negative half areas (defined next);
- **Geometry:** pointers to the points, lines and half-lines;
- **Mesh:** vectors of pointers to nodes, edges, faces and elements;
- **Mesh Status:** used for parallel meshing, it checks whether the area is already meshed and for permission to proceed.

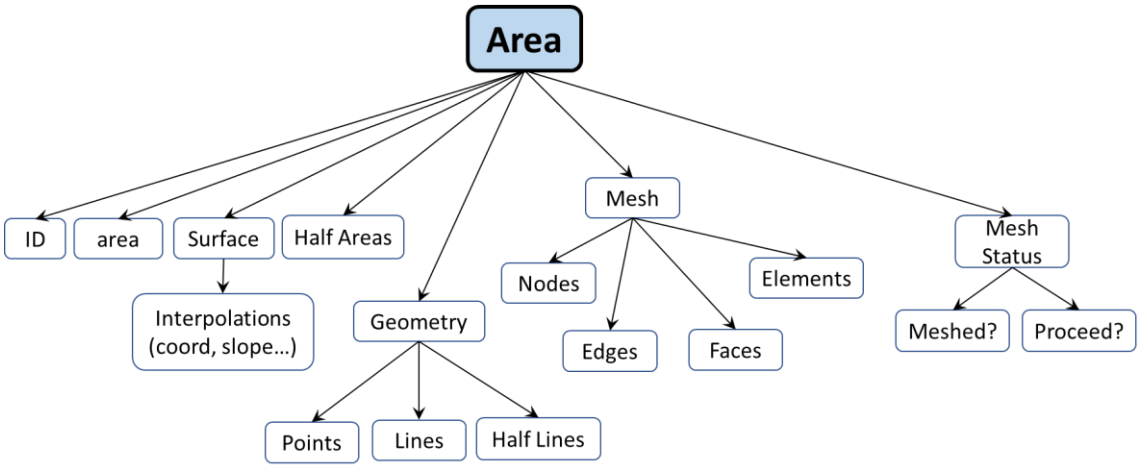

Fig. 6.9 – Area object. Source: own authorship.

Again, polymorphism was employed to implement the surfaces, as illustrated in Fig. 6.10. A flat surface is created when the area is composed exclusively of straight lines, otherwise it will be a quadratic surface, i.e., interpolated with the quadratic isoparametric shape functions. Until the present moment, the implementation allows only triangular and rectangular shapes, without any voids. The development of area generic shapes is a very complex task, that requires a lot of verifications, and may be included in the future if necessary, as well as more complex surfaces.

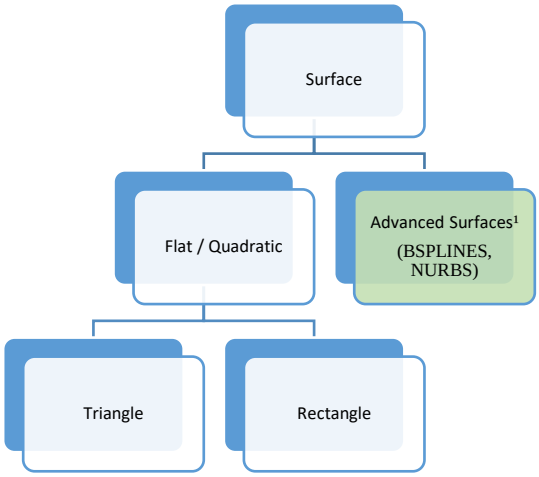

Fig. 6.10 – Class hierarchy from Area. Source: own authorship.

1: not implemented, but the code is ready to receive it.

Half-Area is an extrapolation of the concept of Half-Line. Each half-area is defined by a counter-clock wise, closed and continuously connected set of half-lines, as illustrated in Fig. 6.11. In this case, a rectangular area is given by the lines L1, L2, L3 and L4. As the lines L2 and L4 were defined with opposite orientation, the positive half-area was defined with their respective negative half-lines, maintaining thus the orientation coherence. The negative half-area is given by the twins of the half-lines that defines the positive half-area, and, by observing this image from the other side, it is concluded that the counter-clockwise orientation is also maintained for the negative half-area. When creating the area, the compatibility and connectivity between the lines are easily checked by verifying if the given lines share the correspondent points.

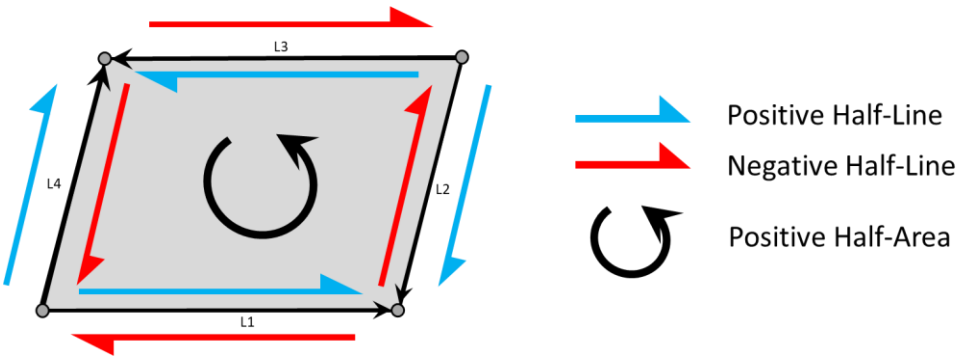

Fig. 6.11 – Half-Area. Source: own authorship.

The half-area concept is very useful for constructing the geometries, since each pair corresponds to a side of the area, and also for finite element applications, since this orientation is important for shell elements, for example. The Half-Area object, Fig. 6.12,

besides the geometric links, also stores pointers to the mesh entities attached to the area. This direction indexing system makes the selection process for contact applications a very simple task.

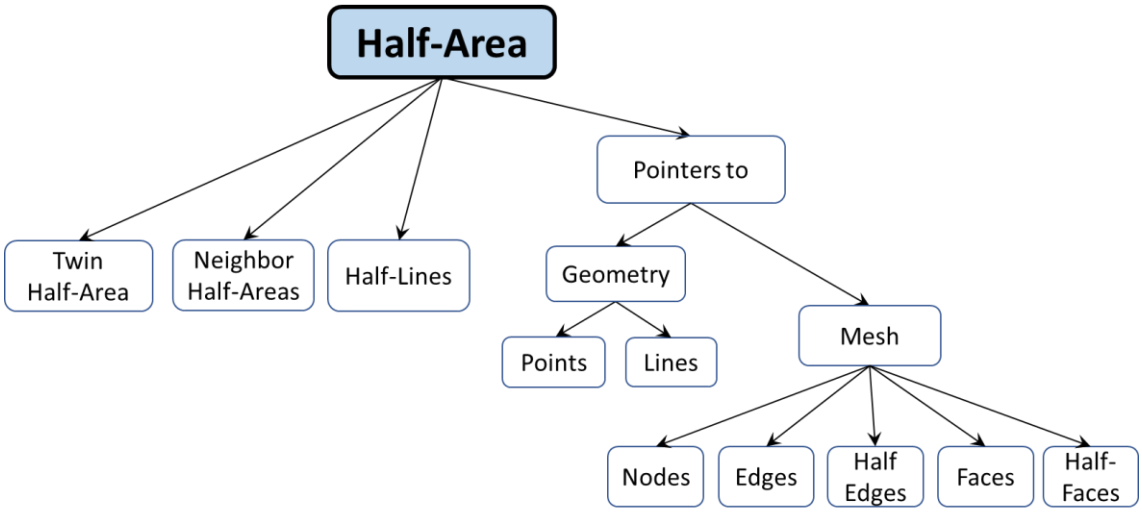

Fig. 6.12 – Half-Area object. Source: own authorship.

6.1.4 Volume

Since there are still no solid finite macroelements for modelling flexible pipes, volumetric entities have not been developed in this work.

6.2 Mesh

The mesh hierarchical levels are represented in Fig. 6.13. Cells are employed for solid finite elements. Faces are used for shells or solids of revolution, and also in surface contact applications. Edges are utilized for beam elements and in contacts involving lines. Nodes are the basis of the finite element and are also employed in node-to-node contact elements. In the next items, each of these levels are presented in detail.

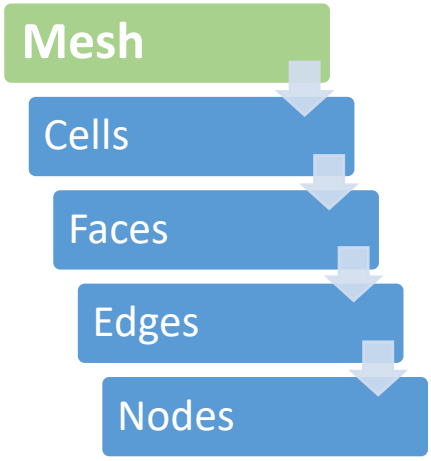

Fig. 6.13 – Hierarchical relations at mesh level. Source: own authorship.

6.2.1 Node

The **Node** object is illustrated in Fig. 6.14 and consists of:

- **ID:** an integer that stores its identification;
- **Order:** an integer the stores the order of the node (for standard nodes it is always equals to zero);
- **Node Type:** an enumerator that specifies if the node is of standard of Fourier type;
- **Nr of DOFs:** an integer that stores the total number of degrees-of-freedom;
- **DOF indexes:** a vector of integer with the global values for each local degree-of-freedom;
- **DOF statuses:** a vector of enumerators that specifies the status from each local degree-of-freedom (free, imposed, etc.);
- **DOF values:** a series of vectors that stores important values of the nodal degrees-of-freedom, such as initial displacements, applied loads, etc.

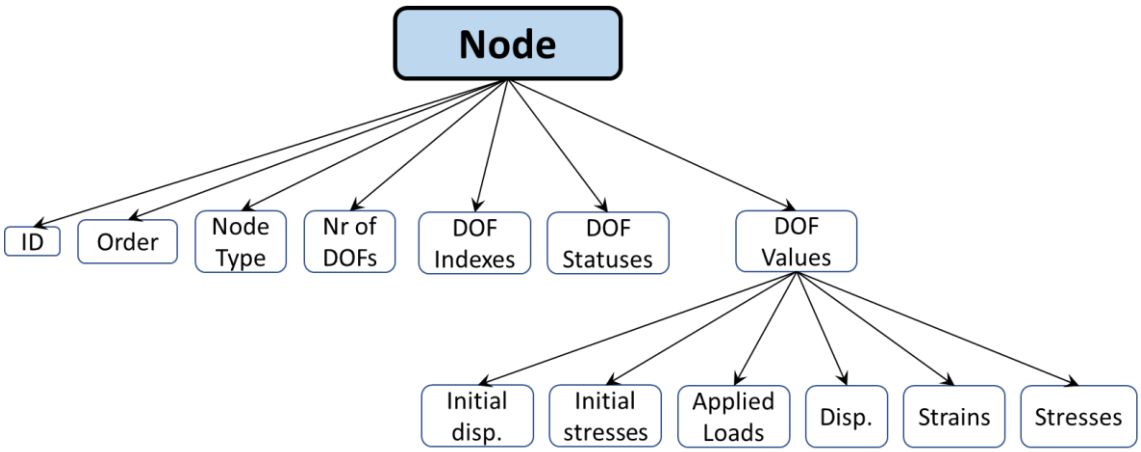

Fig. 6.14 – Node object. Source: own authorship.

As shown in Fig. 6.15, polymorphism was employed to derive the abstract node class into the Standard and Fourier classes, so that the program is able to handle and manipulate these two types of nodes without major problems.

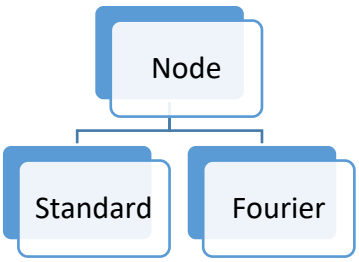

Fig. 6.15 – Node polymorphism. Source: own authorship.

6.2.2 Edge

The Edge object, Fig. 6.16, consists of:

- **ID:** an integer that stores its identification;
- **Edge Order:** an enumerator that specifies if the edge is linear or quadratic;
- **Half-Edges:** the positive and negative half-edges from this edge;
- **Nodes:** two nodes, in the linear case, or three, in the quadratic one;
- **Selection Status:** used for global selection of entities.

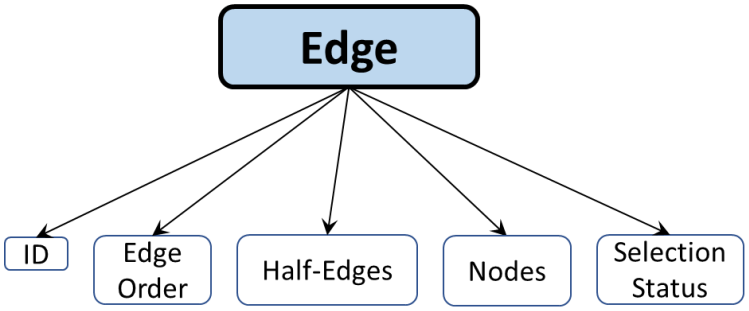

Fig. 6.16 – Edge object. Source: own authorship.

As illustrated in Fig. 6.17, linear and quadratic edges can be created. Higher-order edges can be included in the future if necessary.

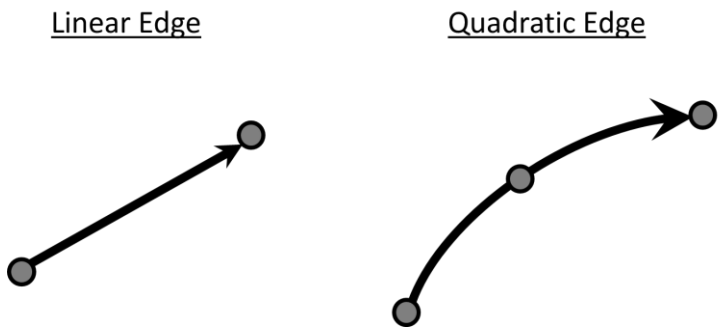

Fig. 6.17 – Linear and quadratic edges. Source: own authorship.

Half-Edges act very similar to the half-lines, since their purpose is justified by orientation and indexing reasons.

6.2.3 Face

The faces are used to mesh areas, being directly applied in FEM for shell and solid of revolution elements. They are also used to form the cells, which are then applied for solid elements. A face is defined by a continuous and closed set of edges. As illustrated in Fig. 6.18, linear and quadratic versions of the triangular and rectangular shaped faces were implemented in PipeFEM, which is enough to cover most of the shape functions used for finite elements.

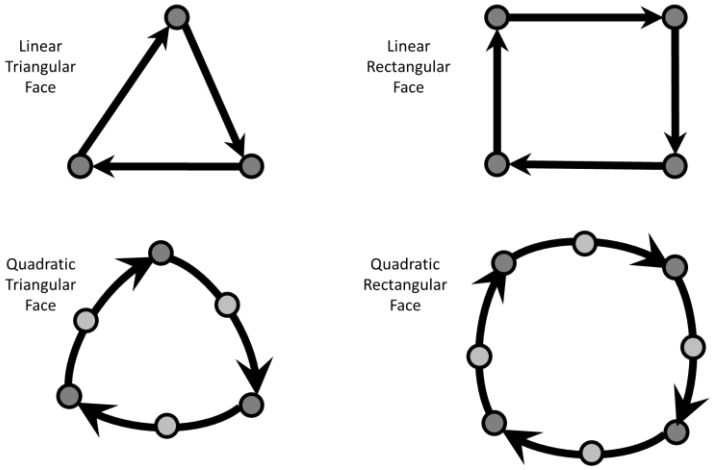

Fig. 6.18 – Linear and quadratic versions of the triangular and rectangular shaped faces. Source: own authorship.

The **Face** object is illustrated in Fig. 6.19 and consists of:

- **ID:** an integer that stores its identification;
- **Edge Order:** an enumerator that specifies if the face is linear or quadratic. Faces with mixed edge order are not possible;
- **Face Type:** specifies whether the face is triangular or quadratic;
- **Half-Faces:** the positive and negative half faces (defined next);
- **Nodes:** vector of pointers to the nodes attached to this face;
- **Edges:** vector of pointers to the edges attached to this face;
- **Half-Edges:** the positive and negative half-edges;
- **Selection Status:** used for global selection of entities.

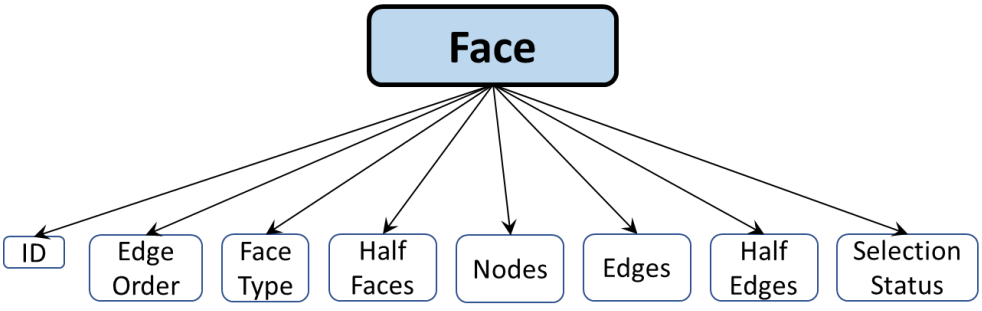

Fig. 6.19 – Face object. Source: own authorship.

Half-faces are very similar to the half-areas previously presented. They are very useful for selecting the sets of nodes and edges in the already correct sequence,

eliminating the large amount of verifications that would be necessary for a not indexed data structure.

6.2.4 Cell

Tetrahedral and hexahedral versions of the cells were implemented in PipeFEM. However, since there are still no solid finite macroelements for modelling flexible pipes, the cell objects have not been completely developed.

6.3 Parallel Mesh Generation

PipeFEM provides a series of methods for mesh generation, in which the user must specify the input parameters, such the list of geometric entities to be meshed, element types, materials and sections. Then, the meshing procedure follows the sequence of steps from Fig. 6.20. The first one consists of a series of compatibility verifications, which can be a check of the consistency of the input data (for example, whether the specified material exists or not) or if the specified element really applies to the type of geometry that is being meshed. If any incompatibility is found, the method is aborted, a message is printed to the user and no elements are created. Otherwise, it proceeds to the next step, which is a geometric meshing, in this case, the items of the aforementioned data structure (nodes, edges and faces). The last step consists in creating the finite elements and associating them with their geometric mesh (edges for beam elements and faces for solids of revolution, for example), resulting then in the final meshed geometry.

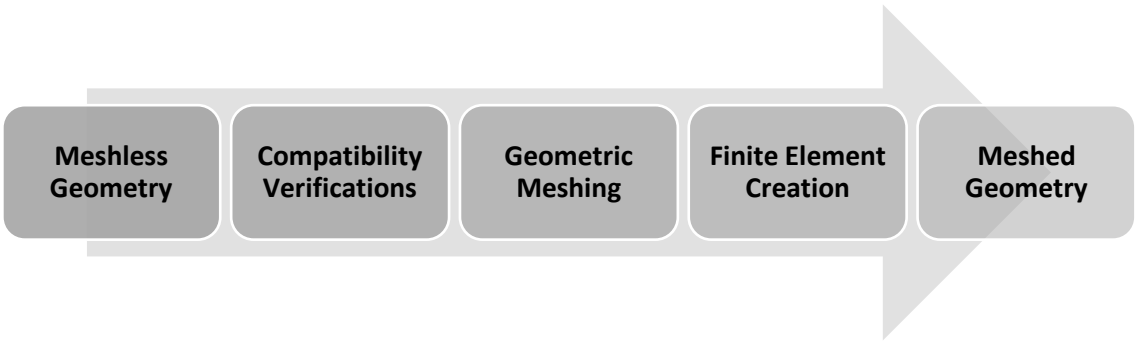

Fig. 6.20 – Meshing processes. Source: own authorship.

The geometric meshing is parallelized with OpenMP, a set of compiler directives and routines for shared memory multiprocessing programming in C++. A cascade

methodology was developed in order to avoid conflicts between threads and to allow the parallel accomplishment of this task. This cascade methodology is based in the hierarchical levels of the geometry, Fig. 6.21. Before a level starts the generation of its own geometric mesh, it must certify that the immediately below level is already meshed, as illustrated in Fig. 6.22.

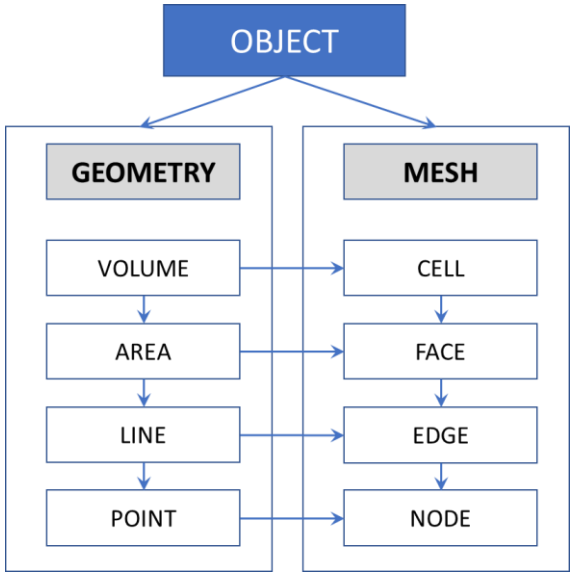

Fig. 6.21 – Hierarchical levels of geometry and mesh and their relationships. Source: own authorship.

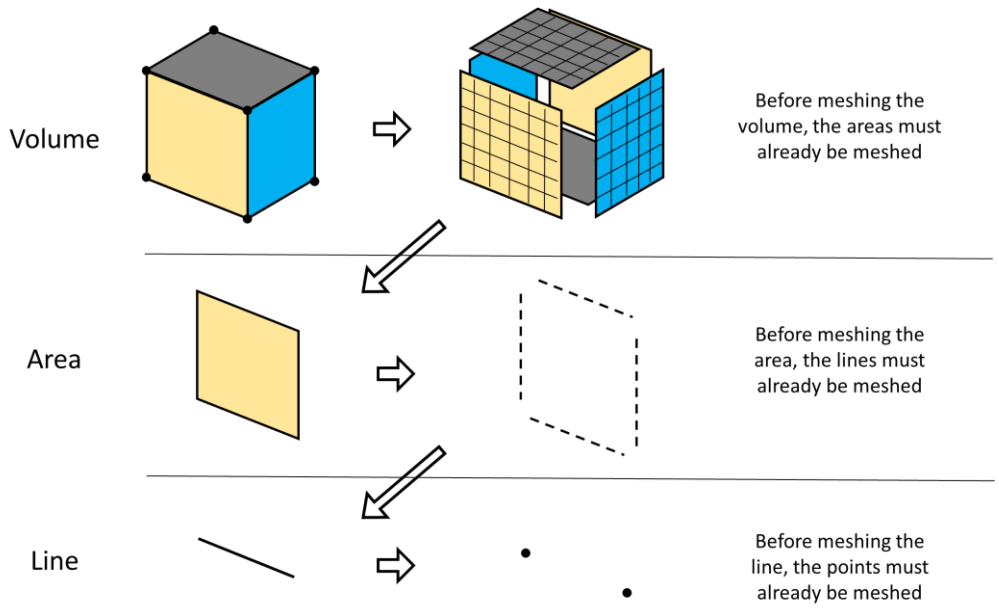

Fig. 6.22 – Cascade methodology of the geometric meshing. Source: own authorship.

This cascade methodology ensures that the entity creation always starts from the lowest level that is not meshed yet. When two different areas share the same line, for

example, and the mesh operation is performed in parallel, it may occur the situation in which two or more threads try to mesh the same line more than once, and worst, at the same time. In this case, if the implementation is not correctly synchronized, it would result in duplicate and poorly connected meshes. This problem was solved with the inclusion of aforementioned "*Mesh Status*" object for each geometric entity of the program. As the name suggests, it stores the information of whether the entity has already been meshed or not. Right at the beginning of the method it is consulted and, if the mesh already exists, nothing new is created. In addition, the Mesh Status also works similarly to a lock. When two or more threads simultaneously check the inexistence of the mesh and try to create it, only one thread receives the permission to continue and the others remain idle until the operation is completed.

6.4 Indexed Data Structure

The hierarchical levels from Geometry (Fig. 6.1) and Mesh (Fig. 6.13) enable a high degree of modularization of the data structure. Volumes are made of areas, which in turn are delimited by lines, which are defined by points. Analogously, cells are composed of faces, which are delimited by edges, which are defined by nodes.

In addition to this, as it can be seen in the descriptions of the objects along items 6.1 and 6.2, every entity can directly access all other lower-level entities that belong to it. In this case, the direct accesses are done via *pointers*, which stores the memory addresses of the objects in question, conferring high computational performance. The geometry creation methods are responsible for the initialization of the pointers related to indexing between different hierarchical levels of geometry. The meshing methods are responsible for indexing between geometry and mesh levels, besides the indexing between different mesh hierarchical levels.

The combination of these two characteristics, modularization and direct access, resulted in a fully integrated and indexed data structure. One of the main advantages of this relies in the ease of selecting items. As illustrated in Fig. 6.23, it is a trivial task to select specific nodes, edges or faces from a meshed area. Therefore, PipeFEM presents the same facilities and features found in the multi-purpose finite element package ANSYS® for items selections, which are extremely useful for contact and load applications, and, at the same time, it has an internally very organized and stratified data

structure that takes advantage of the computational benefits of the direct indexing and that facilitates the implementation and manipulation of three-dimensional finite elements.

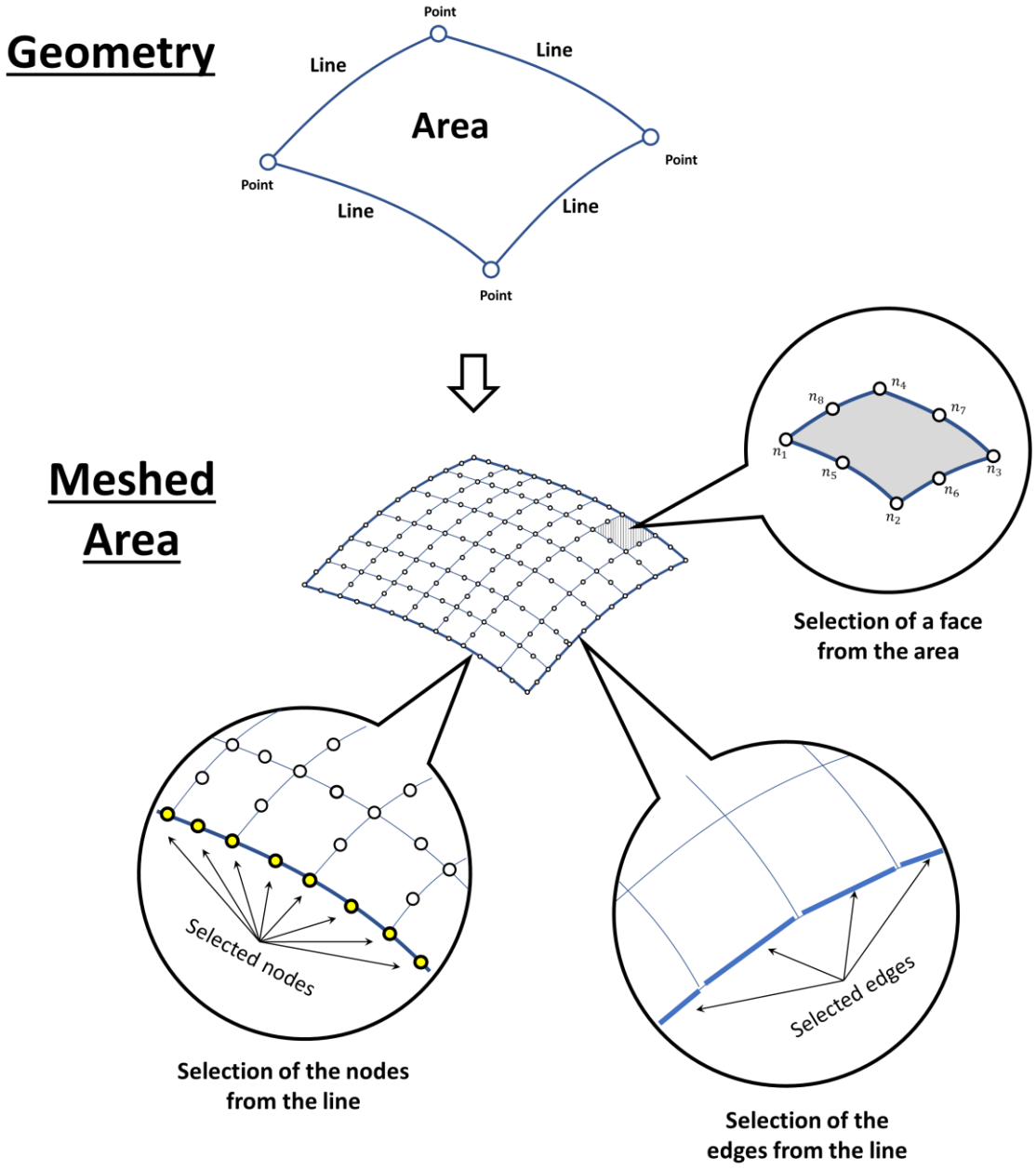

Fig. 6.23 – Indexed data-structure enables efficient entity selections.

7 Layer and Pipe

Due to its generic nature, modeling a flexible pipe only with the methods and resources provided by the geometry and mesh data structure from the previous chapter would be as laborious as modeling the pipe in a multi-purpose finite element package, such as ANSYS® or ABAQUS®. In order to circumvent this problem, by exploring the encapsulations levels provided by the C++ language, a specific methodology for the hierarchical representation and modelling of a flexible pipe was developed and implemented in PipeFEM. Geometry, mesh, loads and contact generations are encapsulated and automatized, so that, given a set of user-defined parameters, only simple instructions are necessary to construct a model of flexible pipe and simulate it.

In this methodology, two new hierarchical levels, *Layer* and *Pipe*, are introduced to the already existing ones from Geometry and Mesh, as illustrated in Fig. 7.1.

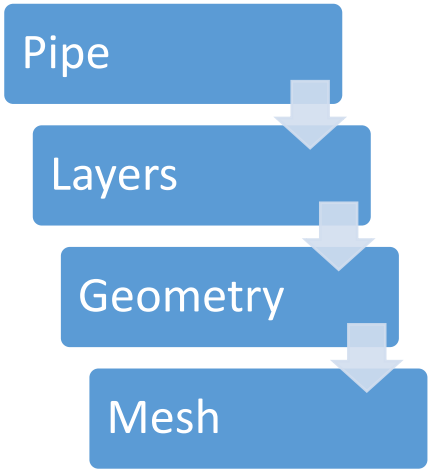

Fig. 7.1 – Global hierarchical level. Source: own authorship.

The pipe is composed of several layers. Each of these layers have their own geometry, as well as element meshes attached to them. This is exemplified in Fig. 7.2 for a tensile armor layer. In this case, the geometry consists of a user-define number of lines, that are meshed with the helical beam elements.

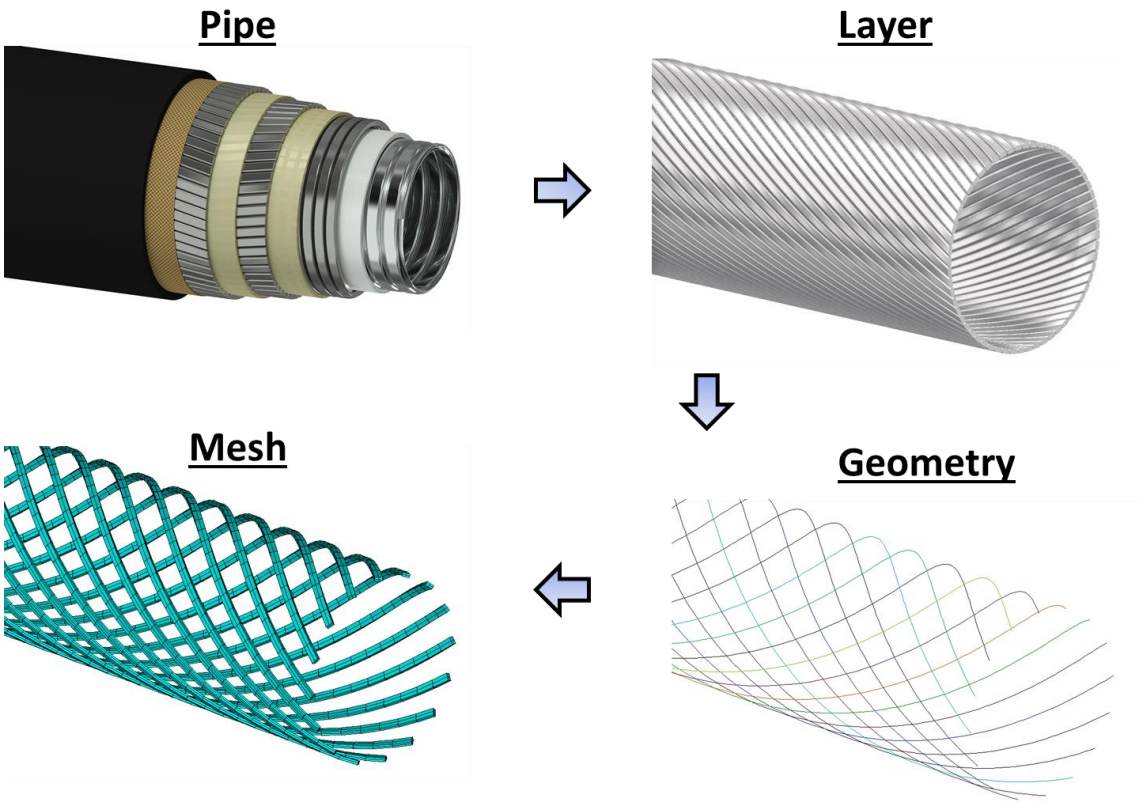

Fig. 7.2 – Example of the hierarchical levels application for a layer of tensile armors. Source: own authorship.

7.1 Layer

As already mentioned, *Layer* is the second in global hierarchy. In PipeFEM, it were implemented pre-defined types of layers, such as the cylindrical and tensile armor layers. For these cases, all geometry and mesh generation is encapsulated and automated, so that it can be done with simple instructions. As shown in Fig. 7.3, each layer has a material, defined by the user during its creation. The nature of the geometry and mesh depend on how the layer is modeled and which elements it utilizes. Layer self-contacts were included in this hierarchy, since it could occur for the interlocked layers. In this case, the layer object must be able to verify and treat correctly its occurrence. However, since there are still no macroelements for modelling these layers, self-contacts are not exploited in this version of the program.

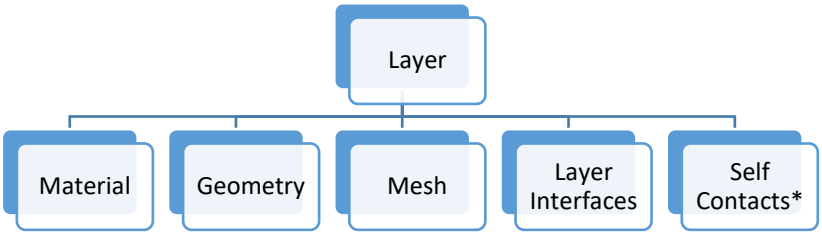

Fig. 7.3 – Layer hierarchy. Source: own authorship. *Not explored at the current version.

The layer interfaces are illustrated in Fig. 7.4. These interfaces are linked to the geometry and mesh of the layers, thus allowing a direct access to load applications and contact definitions. The bottom and top layer interfaces are responsible for the traction, compression, torsion and bending loads. The outer and internal interfaces are employed mostly for contacts between layers, except when they coincide with the inner and outer side of the tube, when they are used for pressure application.

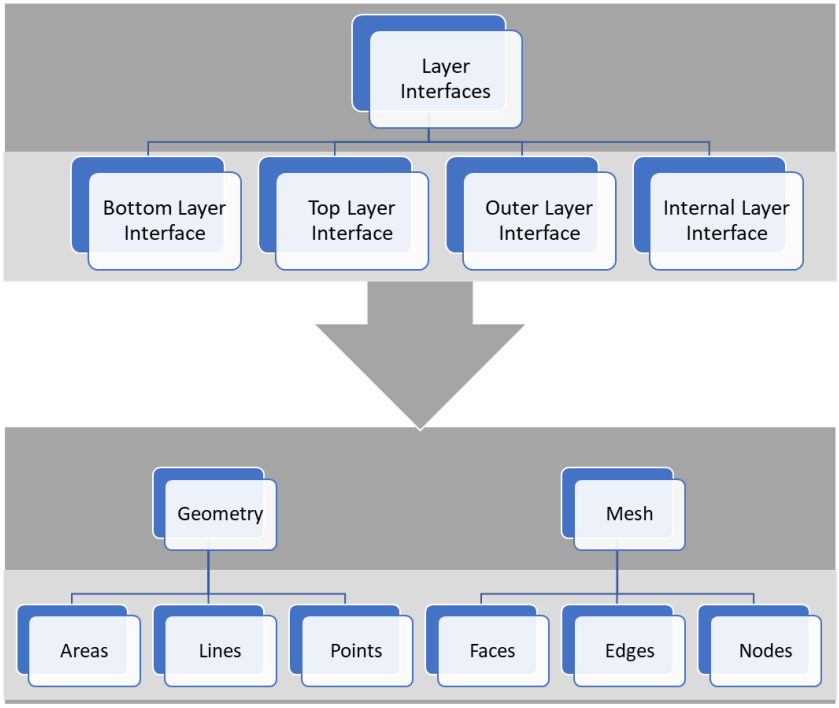

Fig. 7.4 – Layer interfaces. Source: own authorship

The Layer object, Fig. 7.5, consists of:

- **ID:** an integer that stores its identification;
- **Layer Type:** an enumerator that specifies the type of the layer;
- **Geometry:** pointers and indexing to the attached geometry;
- **Mesh:** pointer and indexing to the attached mesh;

- **Layer Interfaces:** the interfaces described in Chapter 4;
- **Selection Status:** used for global selection of entities.

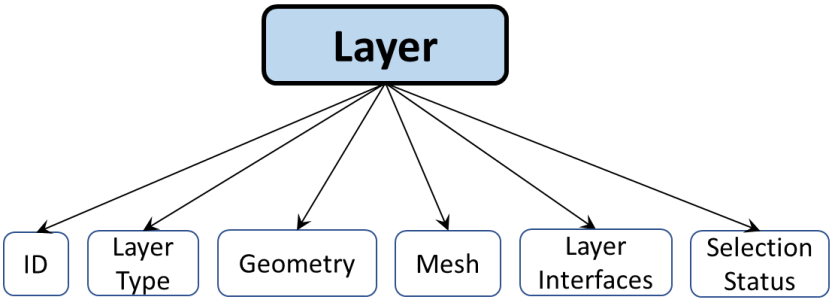

Fig. 7.5 – Layer object. Source: own authorship.

Polymorphism was employed to derive an abstract layer class into derived ones, corresponding to the layer types which can be modeled so far, as illustrated in Fig. 7.6. Each of the derived classes stores specific parameters, such as layer length, number of tendons, lay angle, layer thickness, etc.

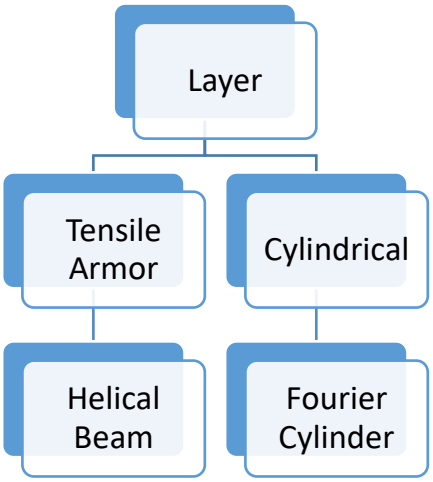

Fig. 7.6 – Layer polymorphism. Source: own authorship.

7.2 Pipe

Pipe represents the highest level of hierarchy and is the main object when creating a model of flexible pipe. As shown in Fig. 7.7, it is responsible for the definition and storage of the layers. Through the polymorphism from C++ language, the program is able to manipulate different pre-defined layer types. The pipe object is also responsible of the

contact definitions related to the layers. Through a simple instruction, the user just need to tell the program that layer A is in contact with layer B, and internally all necessary contact pairs are created.

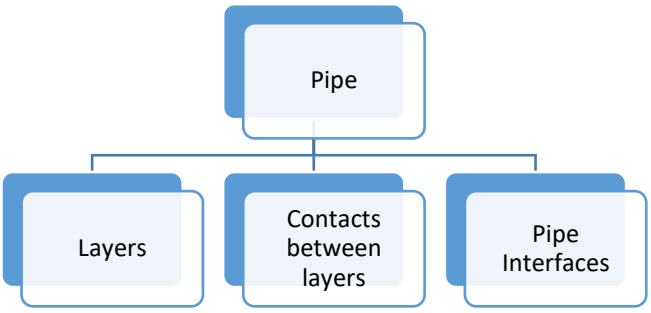

Fig. 7.7 – Pipe object. Source: own authorship.

The pipe interfaces, shown in Fig. 7.8, were created to facilitate load applications. The two end extremities of the pipe are represented by the *Bottom* and *Top* interfaces, which, in turn, are linked to the respective *Bottom* and *Top* interfaces from all layers. When a compression load is applied at the *Top* interface, for instance, it is automatically transmitted to all respective interfaces of layer level. The *Outer* and *Internal* interfaces are used for external and internal pressure loads, respectively. The definitions of the innermost and the outermost layers are not yet automated, they need to be done by the user. By comparing the layer diameters, these interfaces could be automatically determined, a possible upgrade for PipeFEM in the future.

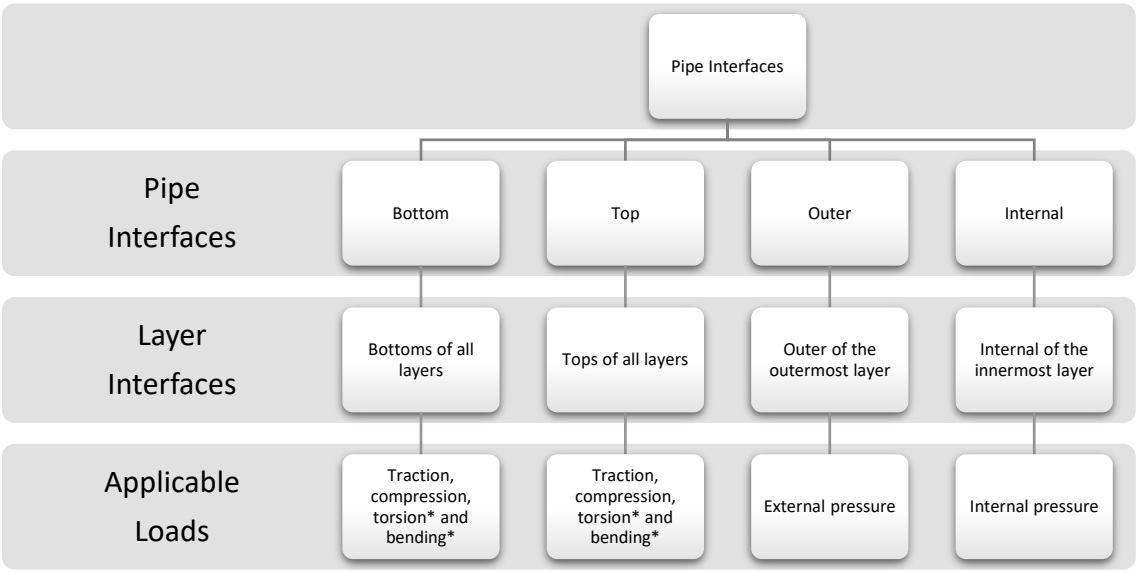

Fig. 7.8 – Pipe interfaces hierarchy. *Not yet implemented. Source: own authorship.

7.3 Contact Between Layers

With the finite macroelements developed so far, two situations of contact between layers are possible, as shown in Fig. 7.9.

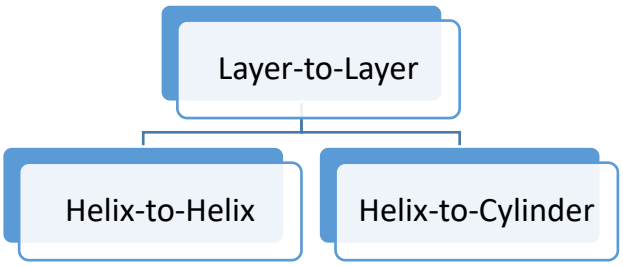

Fig. 7.9 – Possibilities of contact between layers. Source: own authorship.

In order to create the contact between two layers, the user only needs to specify the contact behavior (bonded or frictional), the layers and their respective interfaces that will be in contact (internal or external), so that PipeFEM automatically detects the contact case (helix-to-helix or helix-to-cylinder) and creates in parallel all the contact elements between the two specified layers.

A specific logics of contact detection for helix-to-helix contacts was developed by (TONI, F.G., 2014). This algorithm was implemented, optimized and parallelized in PipeFEM, reducing the total number of operations and the processing time.

8 Solver

Solver is responsible for the numerical solution of the finite element model. It receives everything that has been defined up to the moment prior to its use (such as, element meshes and boundary conditions, among others), and organizes it in a structured way to mount and compute the linear system of equations. In general, the solver demands most of the processing time of the simulation due to the high number of mathematical operations that must be performed by it. Therefore, the overall performance of the program is strongly related to the efficiency of the solver.

In PipeFEM, the model data is transmitted to the solver fully encapsulated in the "*database*" object from Fig. 8.1. Basically, it consists of a structured container of pointers to all entities that comprise the model. It also facilitates data manipulation, by allowing the selection of specific items or the iteration along all items of a desired type. In addition, the *database* object provides all necessary statistics of the model, such as the total number of nodes or elements.

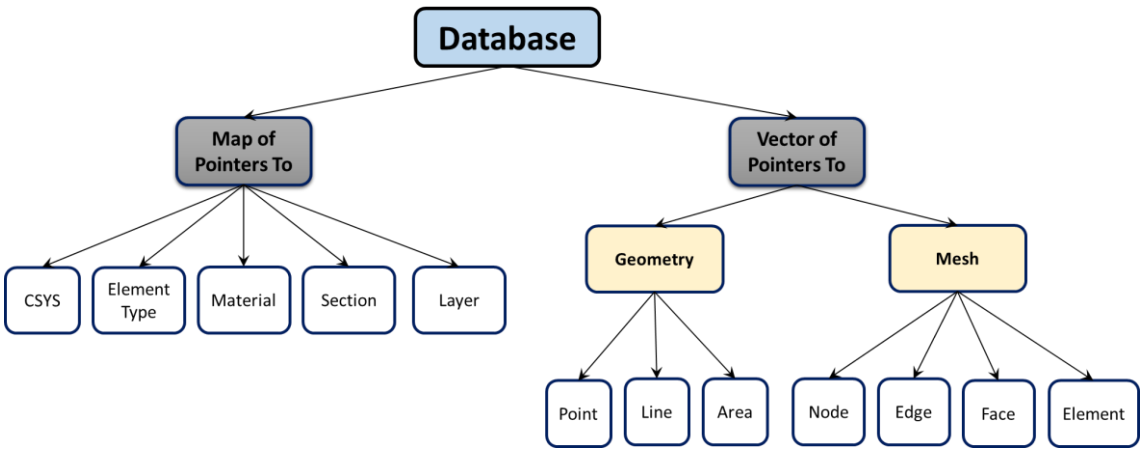

Fig. 8.1 – The *Database* object Source: own authorship.

With all model data at hand, the solver then follows the flowchart illustrated in Fig. 8.2. Aiming computational performance and efficiency, the parallelism of the solution was explored whenever possible in all these steps.

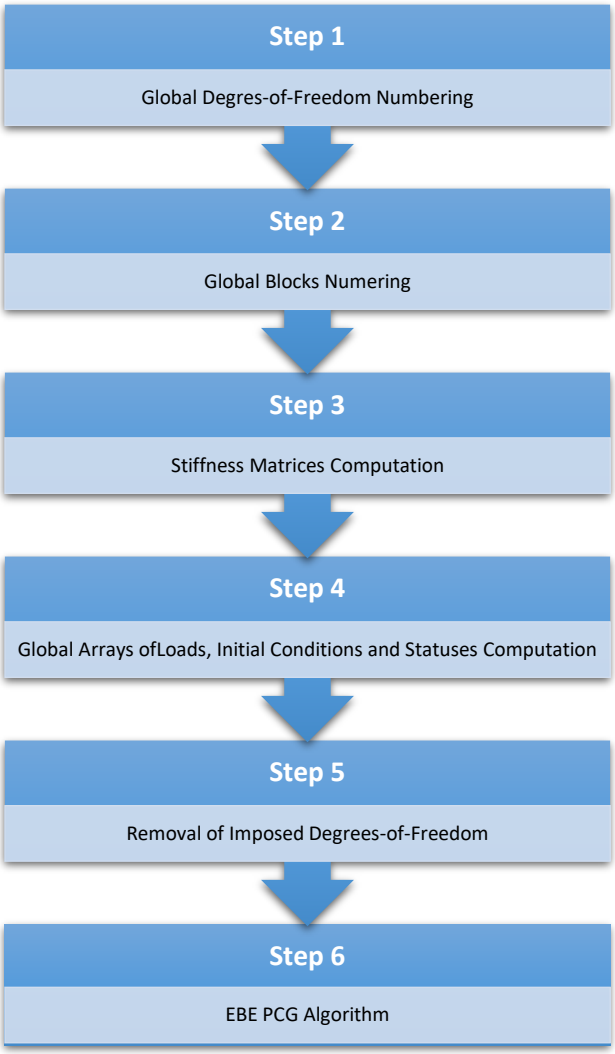

Fig. 8.2 – Solver flowchart. Source: own authorship.

The first step consists of the numbering of the degrees-of-freedom, which, in turn, are given by the nodes of the model. By choosing to perform this numbering operation during the solution, instead of in the pre-processing stage, when the nodes are created, it is obtained more flexibility for their creation or removal in parallel.

Table 8.1 shows the first developed logics to accomplish this numbering task. It begins with the allocation of the temporary vector "*initial_dof_indexes*" of size equal to the total number of nodes. The purpose of this vector is to store the initial values of degree-of-freedom (to be defined) for each node of the model. Then, a sequential iteration is performed across all nodes (unfortunately it cannot be parallelized) to update the values of "*initial_dof_indexes*" using the method "*NrDOFs()*", that returns the total number of d.o.f.s. from the selected node, including all higher-order d.o.f.s if the node is Fourier expanded, and is given by the equation:

$$NrDOFs = \begin{cases} nr_{dofs} & if\ Standard \\ nr_{dofs}\,(2\,.\,order + 1) & if\ Fourier \end{cases} \qquad \text{Eq. 8.1}$$

where nr_{dofs} is the number of degrees-of-freedom intrinsic to the node (defined by the type of element that allocated it) and $order$ is the Fourier expansion order. Once determined the initial d.o.f. values from all nodes, the remaining values share no dependency and can be filled in parallel. Lastly, the total number of degrees of freedom is computed.

Table 8.1 – First implementation of the d.o.f.s numbering algorithm. Source: own authorship.

```
   // Vector that stores the first d.o.f. value from each node of the model
1. Vector<int> initial_dof_indexes(database->nodes.size());

2. initial_dof_indexes[0] = 0; // The first d.o.f of the first node is Zero

   // Iteration over all nodes to define the initial_dof_indexes
3. for (int i = 0; i < database->nodes.size() - 1; i++)
   {
      // NrDOFs() returns the total number of d.o.f.s from the node
      int inc = database->nodes[i]->NrDOFs();
      initial_dof_indexes[i + 1] = initial_dof_indexes[i] + inc;
   }

   // Parallel iteration to fulfill the remaining d.o.f.s
4. #pragma omp parallel for num_threads(nrThreads)
   for (int i = 0; i < database->nodes.size(); i++)
   {
      database->nodes[i]->AssignGlobalDOFValues(initial_dof_indexes[i]);
   }

   // Total number of degrees-of-freedom
5. int  n = initial_dof_indexes.last() + database->nodes.last()->NrDOFs();
```

Although the global matrix is eliminated in the EBE method, its sparsity pattern remains valid and may influence the convergence rate of the iterative methods of linear system solution. The algorithm from Table 8.1 is very efficient in computational terms. However, it has a weak point, that lies the fact that the numbering pattern is exclusively determined by the sequence with which the nodes are added to the database. Since the mesh is performed in parallel, this nodal addition to the database is random and follows

no predefined logical sequence, what generates a sparsity pattern of higher bandwidth and more dispersed than the ideal. Besides that, all d.o.f. values of the same node are defined in a single pass (step 4 from Table 8.1), which is not the best procedure for the Fourier nodes. For them, ideally, the numbering process should be incremented by Fourier order value, i.e., it should define all d.o.f.s relative to the zero order, then all of them relative to the first order, and so on.

These questions about the first implementation motivated the development of a second version of the numbering logics, shown in Table 8.2. It takes advantage of the geometric entities to perform the numbering in a more structured and deterministic way. The sparsity pattern obtained with this new logic is the same from MacroFEM.

Table 8.2 – Second implementation of the d.o.f.s numbering algorithm. FO_{max}: maximum Fourier order.

1. For each area of the model:
   1.1. Iterates over all nodes of the current area and gets the FO_{max}
   1.2. For $Oder = 0, 1, 2, \dots, FO_{max}$
      1.2.1. For each node of the current order and area:
         1.2.1.1. Verify whether the node is already numbered or not
         1.2.1.2. If not, assign d.o.f. values to the specified order
         1.2.1.3. Increments the total number of d.o.f.s of the model
   1.3. Set all nodes of the current area as already numbered

2. For each line of the model:
   2.1. Iterates over all nodes of the current line and gets the FO_{max}
   2.2. For $Oder = 0, 1, 2, \dots, FO_{max}$
      2.2.1. For each node of the current order and line:
         2.2.1.1. Verify whether the node is already numbered or not
         2.2.1.2. If not, assign d.o.f. values to the specified order
         2.2.1.3. Increments the total number of d.o.f.s of the model
   2.3. Set all nodes of the current line as already numbered

3. For each node of the model:
   3.1. Iterates over all nodes and gets the FO_{max}
   3.2. For $Oder = 0, 1, 2, \dots, FO_{max}$
      3.2.1. For each node of the model:
         3.2.1.1. Verify whether the node is already numbered or not
         3.2.1.2. If not, assign d.o.f. values to the specified order
         3.2.1.3. Increments the total number of d.o.f.s of the model
   3.3. Set all nodes as already numbered

Source: own authorship.

This second implementation increases the number of operations, since it requires additional iteration passes. However, as these iterations are very fast in modern processors, the impact on the simulation time was negligible. It is also important to note

that, despite the better sparsity pattern, the numbering sequence of this second implementation is still not optimum. The optimality is achieved with bandwidth optimization algorithms, a feature that can be included in future versions of this analysis tool if necessary.

The second step of the flowchart from Fig. 8.1 consists in the numbering of the blocks of the model, which, in this case, is the same as the numbering of the element stiffness matrices. The term *block* was defined in the EBE Matrix (item 5.4) and, in this work, it comprises not only the element stiffness matrix, but also the local-global indexing array.

The main reason for numbering the blocks is explained by the fact that the Fourier elements (such as Solids of Revolution) possess more than one stiffness matrix, one for each expanded order value, more exactly. Therefore, it is important to number these blocks so that the EBE Matrix can be defined and fulfilled in parallel. The procedure used in the numbering of the blocks is presented in Table 8.3, and is analogous to the one from Table 8.1.

Another important feature of this algorithm, is the definition of the vector parameter called "*BlockDimensions*". It consists of a vector of integers, that stores the dimension of each of blocks that comprises the model. The *BlockDimensions* is directly used to create the EBE Matrix object, that employs the values specified in this vector to allocate the proper amount of memory to store all blocks.

Table 8.3 – Blocks numbering algorithm.

```
// Vector that stores the first block value from each element of the model
1. Vector<int> element_initial_block(database->nodes.size());

// The first block of the first element is Zero
2. element_initial_block[0] = 0;

// Iteration over all element to define the element_initial_block
3. for (int I = 0; I < database->elements.size() – 1; i++)
   {
     // NumberOfBlockMatrices returns the total number of blocks
     int inc = database->elements[i]->NumberOfBlockMatrices();
     element_initial_block[I + 1] = element_initial_block[i] + inc;
   }

// Parallel iteration to fulfill the remaining blocks
4. #pragma omp parallel for num_threads(nrThreads)
   for (int I = 0; I < database->elements.size(); i++)
   {
      int val = element_initial_block[i];
      database->elements[i]SetInitialBlockNumber(val);
   }

// nb: total number of blocks
5. int inc = database->elements.last()->NumberOfBlockMatrices();
   int nb = element_initial_block.last() + inc;

// Block Dimensions – Necessary for Allocation of the EBE Matrix
6. Vector<int> BlockDimensions(nb);
   #pragma omp parallel for num_threads(nrThreads)
   for (int I = 0; I < database->elements.size(); i++)
   {
     for (int j = 0; j < database->elements[i]->NumberOfBlockMatrices(); j++)
     {
        int id = element_initial_block_id[i] + j; // ID of the block
        BlockDimensions[id] = database->elements[i]->BlockSize(j);
     }
   }
```

Source: own authorship.

The third step of the flowchart from Fig. 8.1 consists in the allocation *EBE Matrix* and the parallel computation of all element stiffness matrices. As the block numbers have already been fully mapped and the *BlockDimensions* vector was defined in the previous step, all necessary memory is pre-allocated through the *EBE Matrix* object and the parallelization is trivial. The implemented algorithm is shown in Table 8.4.

Table 8.4 – EBE Matrix allocation and parallel computation of the element stiffness matrices.

```
// EBE Matrix Allocation
LAP::Containers::Parallel::EBE::EbeMatrix* K = new
      LAP::Containers::Parallel::EBE::EbeMatrix(n, nrThreads, BlockDimensions);


// Parallel Element Stiffness Matrices Computation
#pragma omp parallel for num_threads(nrThreads)
for (int i = 0; i < database->elements.size(); i++)
{
  for (int j = 0; j < database->elements[i]->NumberOfBlockMatrices(); j++)
  {
    // Block ID
    int id = element_initial_block_id[i] + j;

    // Vector of integers, the indexation between local and global basis
    Vector<int> indexes = database->elements[i]->Indexes(j);

    // The Element Stiffness Matrix
    SymMatrix<double> stiffness = database->elements[i]->StiffnessMatrix(j);

    // Block Definition
    K->DefineBlock(id, indexes, stiffness);
  }
}
```

Source: own authorship.

The fourth step of flowchart from Fig. 8.1 consists in the computations of the global arrays of loads, initial conditions and d.o.f. statuses. Despite eliminating the global stiffness matrix, the EBE method still requires the aforementioned global arrays.

As the name suggests, the global array of loads is a dynamically allocated array that stores the final external loads applied to all degrees-of-freedom. The term final means that it can be the result of a single load application, a combination of various applications or no loading at all (in this case, equals to zero). The array of initial conditions and the array of d.o.f.s statuses are intrinsically related to each other. When a degree-of-freedom is fixed or imposed, an initial value must be specified, zero if it is constrained or another value if an imposed displacement is applied, for example.

Until this portion of the implementation, these global data are contained exclusively within the nodes. They can be accessed, but only indirectly, through the pointers to the nodes given by the database. By organizing these data in the form of global arrays, the manipulation becomes direct and much more efficient. Table 8.5 shows the logics of the implemented algorithm.

Table 8.5 – Logics of computation of the global arrays of loads, initial conditions and d.o.f. statuses.

1. Dynamically allocation of the arrays
2. In parallel, for each node of the database:
   2.1. Allocates a local temporary vector of statuses with the values gathered from the nodes (it remains the same, regardless of the Fourier expansion order)
   2.2. For $Oder = 0, 1, 2, \dots, FO_{max}$ (maximum Fourier order)
      2.2.1. For each degree-of-freedom of the specific order and node:
         2.2.1.1. Sets the loads
         2.2.1.2. Sets the initial conditions
         2.2.1.3. Sets the statuses

Source: own authorship.

In the finite element method, the imposed degrees-of-freedom need to be removed from the global stiffness matrix before the solution of the linear system, otherwise it would result in null determinant, and it is still necessary in the EBE method. The fifth step of flowchart from Fig. 8.1 consists, then, of the elimination of the imposed degrees-of-freedom, which, in turn, are specified by the global array of statuses defined in the previous step.

Before the removal, a renumbering of the degrees-of-freedom is necessary, shifting the imposed ones to the end of the queue, as illustrated in Fig. 8.3. This example shows the case of a global stiffness matrix, for which it is easier to understand the removal procedure, but it is analogous in the EBE method. The only difference is that, instead of one single large matrix, this data rearrangement is performed to the many several blocks that comprises the model.

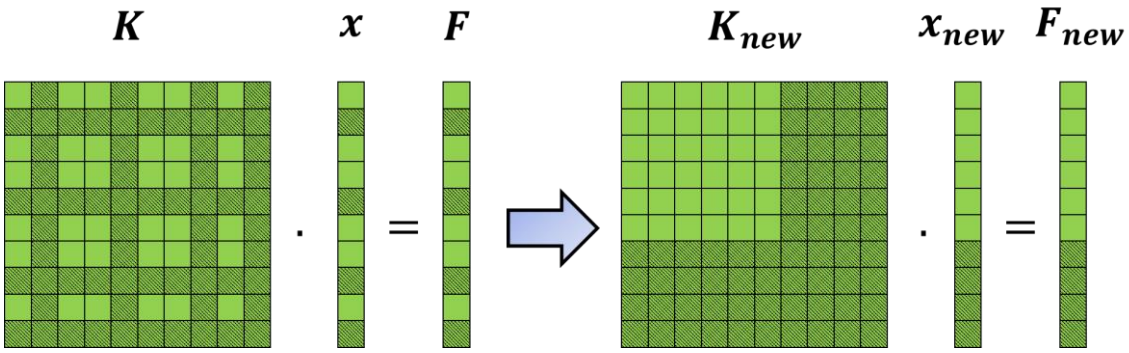

Fig. 8.3 – Global degrees-of-freedom renumbering, shifting the imposed ones to the end. Source: own authorship.

After the renumbering procedure and shifting the imposed d.o.f.s to the end of que queue, the linear system of equations can be divided into four sub-regions as shown in Fig. 8.4, in which:

- The free degrees-of-freedom are denoted by the subscript "*B*"
- The imposed degrees-of-freedom are denoted by the subscript "*C*"
- $\boldsymbol{x}_b$ denotes the free or unknown degrees-of-freedom;
- $\boldsymbol{x}_c$ are the imposed or known degrees-of-freedom;
- $\boldsymbol{F}_b$ represent the external applied loads;
- $\boldsymbol{F}_c$ denotes the unknown boundary reactions;
- $\boldsymbol{K}_{BB}$ are the stiffness terms exclusively from the free degrees-of-freedom;
- $\boldsymbol{K}_{CC}$ are the stiffness terms related to the imposed degrees-of-freedom;
- $\boldsymbol{K}_{BC}$ and $\boldsymbol{K}_{CB}$ are the crossed stiffness terms.

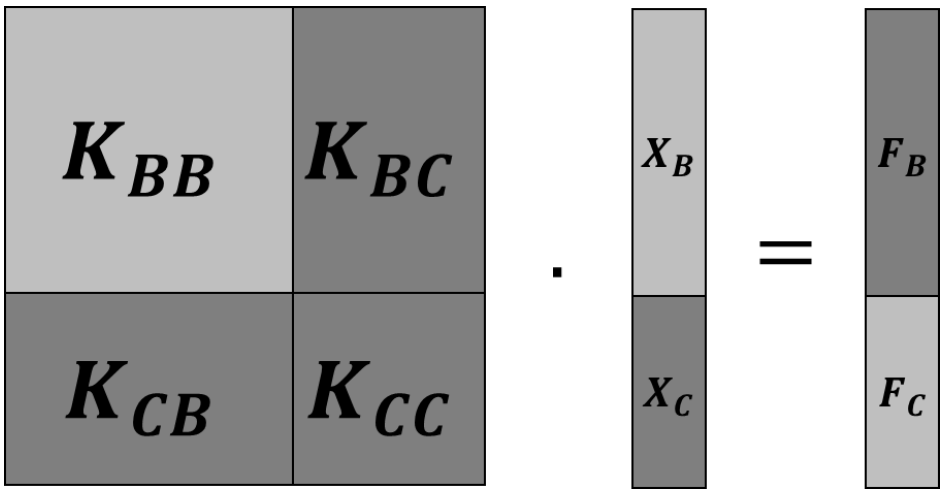

Fig. 8.4 – Linear system sub-regions. Source: own authorship.

The following equation is valid:

$$\boldsymbol{K}_{BB}\,\boldsymbol{x}_b + \boldsymbol{K}_{BC}\,\boldsymbol{x}_c = \boldsymbol{F}_b \qquad \text{Eq. 8.2}$$

In this equation, the only unknown term is $\boldsymbol{x}_b$. Then, the product $\boldsymbol{K}_{BC}\,\boldsymbol{x}_c$ is a vector of known values and it can be moved to the right side of the equation:

$$\boldsymbol{K}_{BB}\,\boldsymbol{x}_b = \boldsymbol{F}_b - \boldsymbol{K}_{BC}\,\boldsymbol{x}_c \qquad \text{Eq. 8.3}$$

Obtaining, thus, the final linear system of equations:

$$\boldsymbol{K}_{BB}\,\boldsymbol{x}_b = \boldsymbol{F}_{final} \quad \text{with} \quad \boldsymbol{F}_{final} = \boldsymbol{F}_b - \boldsymbol{K}_{BC}\,\boldsymbol{x}_c \qquad \text{Eq. 8.4}$$

As already mentioned in item 5.4, the *EBE Matrix* has two methods, that automatically rearranges its internal data in order to remove the imposed degrees-of-freedom, and that computes the terms $\boldsymbol{K}_{BB}$ and $\boldsymbol{F}_{final}$. It is interesting to note that, the product $\boldsymbol{K}_{BC}\,\boldsymbol{x}_c$ is performed in a local element basis on the EBE method.

The sixth and final step of flowchart from Fig. 8.1 consists of the solution of linear system of equations with the implemented EBE-PCG algorithm. Since this algorithm is the core of the work, it is presented individually in the next chapter.

9 Element-by-Element Preconditioned Conjugate Gradient Method

In this chapter, it is presented the complete implementation of the element-by-element version of the preconditioned conjugate gradient method. This algorithm was developed for structural mechanics applications, more specifically to solve large-scale problems of flexible pipes modeled with the finite macroelements developed by PROVASI & MARTINS (Chapter 2).

In PipeFEM, this EBE-PCG algorithm is employed by the *Solver* in the solution of the linear system $\boldsymbol{K}_{BB}\,\boldsymbol{x}_b = \boldsymbol{F}_{final}$ from Eq. 8.4. In order to make the notation more concise and facilitate the reading, these subscripts are removed in this chapter, so that this same linear system, with the imposed degrees-of-freedom already removed, is represented by $\boldsymbol{K}\boldsymbol{x} = \boldsymbol{f}$.

Aiming computational performance, the implementation was performed in the C++ language and parallelized with OpenMP, enabling, thus, the complete utilization of the processing capacity of modern multi-core processors. But, before the discussion of the details of the parallelization, it is important to carefully analyze the PCG algorithm, shown in Table 9.1.

The first four operations are introductory. The first one consists of specifying the initial guess, $\boldsymbol{x}_0$. In most cases, the null vector $\boldsymbol{x}_0 = \boldsymbol{0}$ is adopted, without any loss of generality. In some cases, it is possible to determine a better initial guess, with the advantage that, the closer it is to the exact solution, in less iterations the algorithm will converge. In the second operation, the initial vector of linear residuals is computed. If the initial guess is the null vector, there is no need to perform the costly matrix-vector product operation given by $\boldsymbol{K}\,\boldsymbol{x}_0$. The third one corresponds to the preconditioning application, which it is a trivial operation in the case of the diagonal version, as well as the fourth operation, which consists of a simple copy of array.

Table 9.1 – PCG Algorithm, solution of the linear system $\boldsymbol{K}\boldsymbol{x} = \boldsymbol{f}$.

1. $\boldsymbol{x}_0 = \boldsymbol{x}_{Guess}$
2. $\boldsymbol{r}_0 = \boldsymbol{f} - \boldsymbol{K}\,\boldsymbol{x}_0$
3. $\boldsymbol{z}_0 = \boldsymbol{M}^{-1}\,\boldsymbol{r}_0$
4. $\boldsymbol{p}_0 = \boldsymbol{z}_0$
5. $for\ k = 0, 1, \ldots, max_{iter}$
   - 5.1. $\alpha_k = \frac{\boldsymbol{r}_k^T\,\boldsymbol{z}_k}{\boldsymbol{p}_k^T\,\boldsymbol{K}\,\boldsymbol{p}_k}$
   - 5.2. $\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \alpha_k\,\boldsymbol{p}_k$
   - 5.3. $\boldsymbol{r}_{k+1} = \boldsymbol{r}_k - \alpha_k\,\boldsymbol{K}\,\boldsymbol{p}_k$
   - 5.4. if $\|\boldsymbol{r}_{k+1}\| < tolerance$, then solution converged, exit loop
   - 5.5. $\boldsymbol{z}_{k+1} = \boldsymbol{M}^{-1}\,\boldsymbol{r}_{k+1}$
   - 5.6. $\beta_k = \frac{\boldsymbol{z}_{k+1}^T\,\boldsymbol{r}_{k+1}}{\boldsymbol{z}_k^T\,\boldsymbol{r}_k}$
   - 5.7. $\boldsymbol{p}_{k+1} = \boldsymbol{z}_{k+1} + \beta_k\,\boldsymbol{p}_k$

   $end\ for$
6. If the solution converged, the result is $\boldsymbol{x}_{k+1}$

Source: (SAAD, 2003).

where:

- k – is the iteration count;
- $\boldsymbol{K}$ – is the stiffness matrix;
- $\boldsymbol{r}$ – is the linear residual;
- $\boldsymbol{x_0}$ – is the initial guess or a prediction;
- $\boldsymbol{x}$ – is the trial displacement vector;
- $\boldsymbol{M}$ – is the preconditioning transformation;
- $\boldsymbol{p}$ – denotes the step direction;
- α – is the step length;
- β – defines the correction factor.

It can be noted that, from the second to the fourth operation, there is a dependence on the values of the immediately preceding one. It means that the execution sequence must be respected, i.e., they cannot be concomitantly performed. Nevertheless, each of these four operations is easily parallelizable with the OpenMP directives, with only the exception of the matrix-vector product $\boldsymbol{K}\,\boldsymbol{x}_0$, which requires synchronization in the EBE version. This matrix-vector product is also performed in each iteration of the algorithm and, given its direct importance and influence on the performance of the algorithm, it will be addressed individually and in greater detail further on in this chapter.

The fifth step is the main core of the algorithm and it is responsible for the iterative procedure until the achievement of the convergence or the maximum predefined number of iterations. Again, each of the operations from this iterative scheme has a value dependency with the immediately preceding one. With the exception of the matrix-vector product, all other operations are easily parallelizable with the directives from OpenMP. The sixth and last operation consists of return the final solution if the convergence was achieved.

9.1 Numerical Implementation

The numerical implementation of the method is not exactly a direct transcription of the algorithm from Table 9.1. Aiming the computational performance, in some points, small modifications were necessary in the way that the variables are calculated or manipulated, most of them justified mainly by working in parallel. The complete implementation of the aforementioned algorithm is found in Table 9.2.

Table 9.2 – Implemented EBE-PCG algorithm.

```
double* Solve(LAP::Containers::Parallel::EBE::EbeMatrix* K, double* F)
{

   /* CONTROL VARIABLES – THEY CAN BE SPECIFIED BY THE USER */
   int nrThreads; // Number of Threads for Parallelization
   int maxit;     // Maximum Number of Iterations
   double tol;    // Numerical Tolerance or Admissible Error

   /* INTERNAL VARIABLES */
   bool converged = false;   // Converged?
   double res = 0.0;         // Residual
   int n = K->size();        // Linear Syst. Size or Dimension
   double num = 0, den = 0;  // Auxiliary Numerator and Denominator
```

```
/* ALLOCATION OF THE VARIABLES OF THE PCG ALGORITHM */
int k = 0;              // Iteration Counter
double alfa, beta;      // Step Length and Correction Factor

double* x = new double[n];    // Solution Array
double* Kpk = new double[n];  // Array of the Mat-Vect Prod Result

double* rk = new double[n];   // r_k
double* zk = new double[n];   // z_k
double* pk = new double[n];   // p_k

double* rkp1 = new double[n]; // r_{k+1}
double* zkp1 = new double[n]; // z_{k+1}
double* pkp1 = new double[n]; // p_{k+1}

double* DiagPrec = new double[n]; // Diagonal Preconditioner

/* BEGINNING OF THE METHOD */

K->DiagonalPreconditioner(DiagPrec); // Comp. Diagonal Prec.

#pragma omp parallel for // Initial Guess x0 = 0
for (int i = 0; i < n; i++)
    x[i] = 0.0;

K->MatrixVectorProduct(x, Kpk); // Comp. Mat-Vect Prod.: K x_0

#pragma omp parallel for // Initial Linear Residual
for (int i = 0; i < n; i++)
    rk[i] = F[i] - Kpk[i];

#pragma omp parallel for reduction(+: res) // Initial Residual
for (int i = 0; i < n; i++)
{
    double value = rk[i]; // Square
    res += value * value; // Sum of the square
}

res = sqrt(res); // residual value

if(res <= tol)
    converged = true; // Initial Guess, x0, is already the solution!
else
{
    res = 0.0; // Resets the Residual Value

    // Resets the Residual Value
    #pragma omp parallel for
    for (int i = 0; i < n; i++)
    {
          zk[i] = DiagPrec[i] * rk[i];
          pk[i] = zk[i];
    }

    // ITERATIVE SCHEME
    while (k < maxit)
    {
        K->MatrixVectorProduct(pk, Kpk); // EBE-Matrix Vector Product
```

```
        // Computation of Alfa
        #pragma omp parallel for reduction(+: num, den)
        for (int i = 0; i < n; i++)
        {
            num += rk[i] * zk[i];
            den += pk[i] * Kpk[i];
        }
        alfa = num / den;
        num = 0, den = 0;

        // Updates the solution array "x" and the new residuals
        #pragma omp parallel for
        for (int i = 0; i < n; i++)
        {
            x[i] += alfa * pk[i];
            rkp1[i] = rk[i] - alfa * Kpk[i];
        }

        // Computation of the Norm of the Residual
        res = 0.0;
        #pragma omp parallel for reduction(+: res)
        for (int i = 0; i < n; i++)
        {
            double value = rkp1[i];
            res += value * value;
        }
        res = sqrt(res);

        if (res <= tol) { converged = true; break;} // Convergence check

        // Computation of the Pseudo Residual
        #pragma omp parallel for
        for (int i = 0; i < n; i++)
        {
            zkp1[i] = DiagPrec[i] * rkp1[i];
        }

        // Computation of Beta
        #pragma omp parallel for reduction(+: num, den)
        for (int i = 0; i < n; i++)
        {
            num += zkp1[i] * rkp1[i];
            den += zk[i] * rk[i];
        }
        beta = num / den;
        num = 0.0, den = 0.0;

        // Computation of the new Step Direction
        #pragma omp parallel for
        for (int i = 0; i < n; i++)
            pkp1[i] = zkp1[i] + beta * pk[i];

        k++;

        std::swap(rk, rkp1);
        std::swap(zk, zkp1);
        std::swap(pk, pkp1);

    } // End while / Iterative scheme
} // End if
```

```
    /* DEALLOCATION */
    delete[] DiagPrec; DiagPrec = nullptr;
    delete[] Kpk; Kpk = nullptr;
    delete[] zk; zk = nullptr;
    delete[] rk; rk = nullptr;
    delete[] pk; pk = nullptr;
    delete[] rkp1; rkp1 = nullptr;
    delete[] zkp1; zkp1 = nullptr;
    delete[] pkp1; pkp1 = nullptr;

    return x;

} // End Solve Method
```

Source: own authorship.

The implementation still remains very simple, an acknowledged characteristic of the PCG algorithm, and that helps to explain its success in the literature. It has three control parameters with default values, but that can be changed by the user, if necessary: the numerical tolerance, the maximum number of iterations and the number of threads (for parallelism). It also possesses internal auxiliary variables. In addition to these, it also counts with variables directly associated with the PCG algorithm, mostly of them dynamically allocated arrays of doubles. After defining and allocating the variables, the method is then ready to begin.

Before detailing all the operations, it is important to say that two of them will be explained separately in the next items: the computation of the diagonal preconditioner and the element-by-element matrix-vector product. This is justified by the fact that, due to the EBE method, these two operations are carried out in a very particular way. Besides that, they have a great impact on the over-all performance of the implementation, deserving, therefore, a more detailed description.

After computing the diagonal preconditioner, it proceeds to the definition of the initial guess array (x_0). The null vector option was adopted, and this operation is easily parallelizable with OpenMP.

In sequence, the matrix-vector product $\boldsymbol{K}\,\boldsymbol{x}_0$ is calculated, so that it is possible to determine the initial linear residuals. At this point, it was necessary to include a verification of the norm of the initial residuals (r_0). That is because, if the array on the right side of the linear system is null (in FEM it may happens when no external load is applied to the model and no displacements are prescribed), the null initial guess is already the exact solution of the problem. In this case, the iterative scheme cannot be started,

since there would occur a division by zero right in the first calculation of the step length, α_0.

If the residual is higher than the specified tolerance, the program computes the pseudo-residuals (z_0), the array of step directions (p_0) and then proceeds to the iterative scheme. The first operation of this iterative scheme is the determination of the step length, α_k, given by the division of $\boldsymbol{r}_k^T\ \boldsymbol{z}_k$ with $\boldsymbol{p}_k^T\ \boldsymbol{K}\ \boldsymbol{p}_k$. Both these numerator and denominator are calculated by a parallel sum of n values in to the auxiliary variables num and den. A "*critical section*" or a "lock" could be employed to avoid race condition, but in practice this would serialize the computation and jeopardize the scalability. For situations like these, the OpenMP has a "*reduction*" clause, which is responsible for automatically making copies of the specified reduction variable for each thread, which in turn will act and update only its local copy. At the end of the loop, the local variables are combined to form the final result, a strategy that ensures a good scalability of the solution.

With the step length at hand, it is possible to update the solution array (x_{k+1}) and to compute the new residuals (r_{k+1}). For advantages in cache memory and speed, these two operations were unified in the same loop. Then, the new residual is computed and, if the norm of the array of residuals is smaller than the tolerance, *converged* is set as true and the while loop is broken. Otherwise, the iterative scheme proceeds to the computation of the new pseudo residuals (z_k), an operation that is easily done and parallelized in the case of the diagonal preconditioner. The parallelization strategy in the computation of the correction factor, β_k, is the same as the one from α_k. Lastly the new step directions (p_{k+1}) are determined.

After incrementing the iteration counter and swapping the pointers, a new iteration is ready to start. This swapping operation consists of switching with each other the memory addresses to which the two pointers are pointing. Instead of copying the contents of one array to the other, the exactly same effect is obtained with only one swapping operation. Besides that, the swap eliminates new memory allocations and reallocations, being, therefore, an extremely efficient way to set the arrays for the next iteration.

This iterative scheme is executed until the convergence or the maximum number of iterations is achieved. At last, the dynamically allocated variables must be deleted, so there is no memory leak.

9.2 Diagonal Preconditioner Computation

As the name suggests, the diagonal preconditioner consists of using the values of the main diagonal of the global stiffness matrix to increase the convergence rate of the solution. When the global matrix is stored in a dense or sparse format, its diagonal is readily available. In the EBE method, however, it is obtained indirectly, through a procedure that consists of summing into a global array the values of the diagonals of each of the blocks of the model. Despite the simplicity of this procedure, when performed in parallel, it will occur situations in which two or more blocks share the same position in the global array, as illustrated in Fig. 9.1. In this case, multiple threads will try to concomitantly update a single memory location, generating the need for synchronization mechanisms, otherwise the calculated values would be wrong.

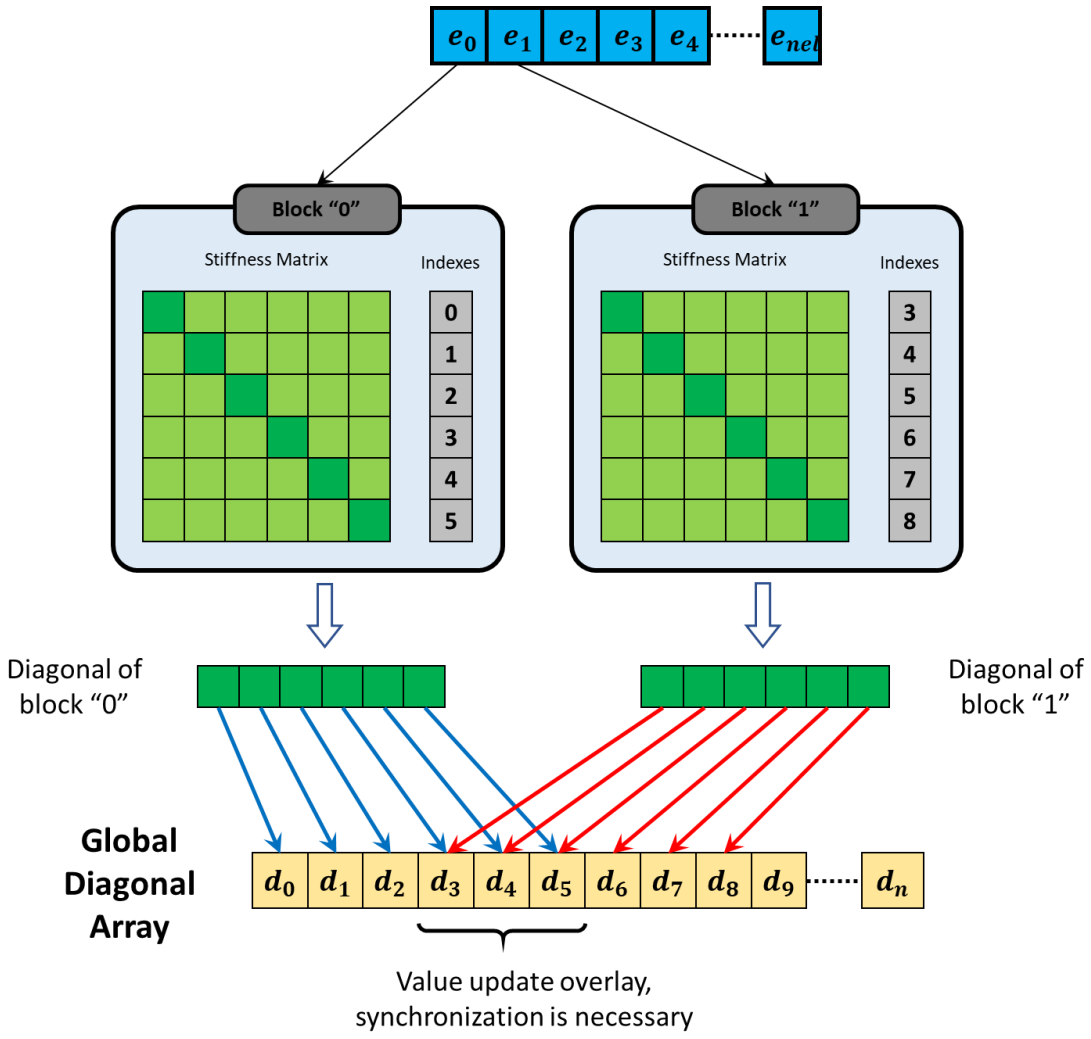

Fig. 9.1 – Example of update overlapping during the parallel evaluation of the global diagonal. Source: own authorship.

In order to synchronize the operation, the method based on local copies was adopted. Illustrated in Fig. 9.2, it consists of allocating a local global array to each thread (initially with all null values), so that each thread can operate on a set of blocks and updates these local copies independently. At the end of the iterations, the local copies are summed in parallel in the final global array.

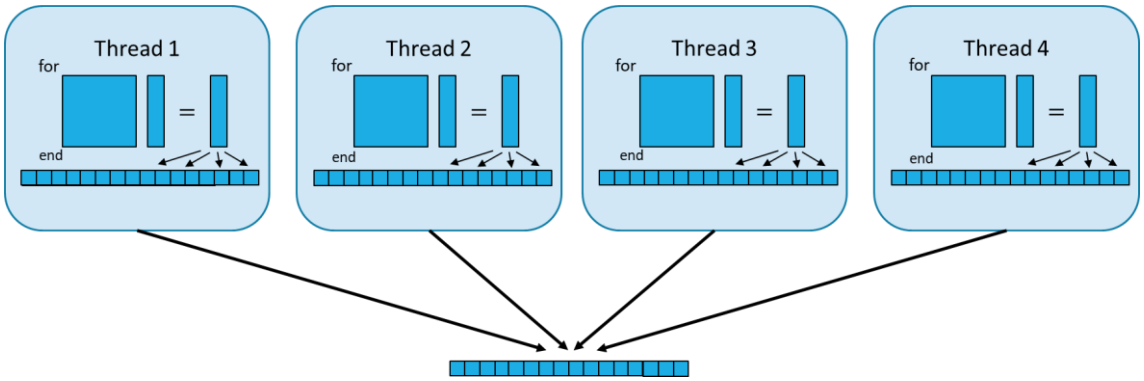

Fig. 9.2 – Synchronization method based on local copies.

As it will be seen in the next item, this synchronization technique is not the most efficient. However, it has been maintained in the implementation for the diagonal preconditioner for two reasons: it is executed only once (thus not compromising the overall performance of the algorithm); and second because it is completely generic and works regardless of the blocks indexes.

Lastly, the diagonal values are inverted, since the multiplication operation is less costly than the division one, thus saving time during the iterative process. If any value of the diagonal is null, the implementation generates a warning and the inversion is replaced by one.

9.3 EBE Matrix-Vector Product

Both the preconditioner and the matrix-vector product are the most critical operations of the PCG algorithm. While the relevance of the first is justified by convergence improvement and consequent reduction in the number of iterations, the importance of matrix-vector product is due to the high number of mathematical operations that must be executed at every iteration. In this way, the overall performance of the algorithm is directly affected to the efficiency of the matrix-vector operation.

The objective of this operation is to perform the product between the global stiffness matrix and the global array of step directions. In the EBE method, however, this product is performed in a local basis, employing all blocks that comprise the model. The detailed explanation of this procedure begins with Fig. 9.3, in which it can be seen that each block contains an element stiffness matrix and an array of integer indexes that relate the local degrees-of-freedom with the global ones. A *gathering* operation is then performed as illustrated in Fig. 9.4. It consists of selecting the corresponding values of the array of global step directions and mounting its own local version. Following the gathering operation, a local matrix-vector product is performed between the element stiffness matrix and the array of element step directions, as shown in Fig. 9.5. The last step is the scattering operation, Fig. 9.6, which consists of spreading the results of the local product into the global product array.

These procedures from Fig. 9.3 to Fig. 9.6 are illustrative. As already mentioned in item 5.4, the element stiffness matrices are stored in a single array of doubles, as well as the indexes in a single array of integers, in order to ensure contiguous memory allocation. All the necessary manipulations to implement this procedure are encapsulated in the EBE Matrix container.

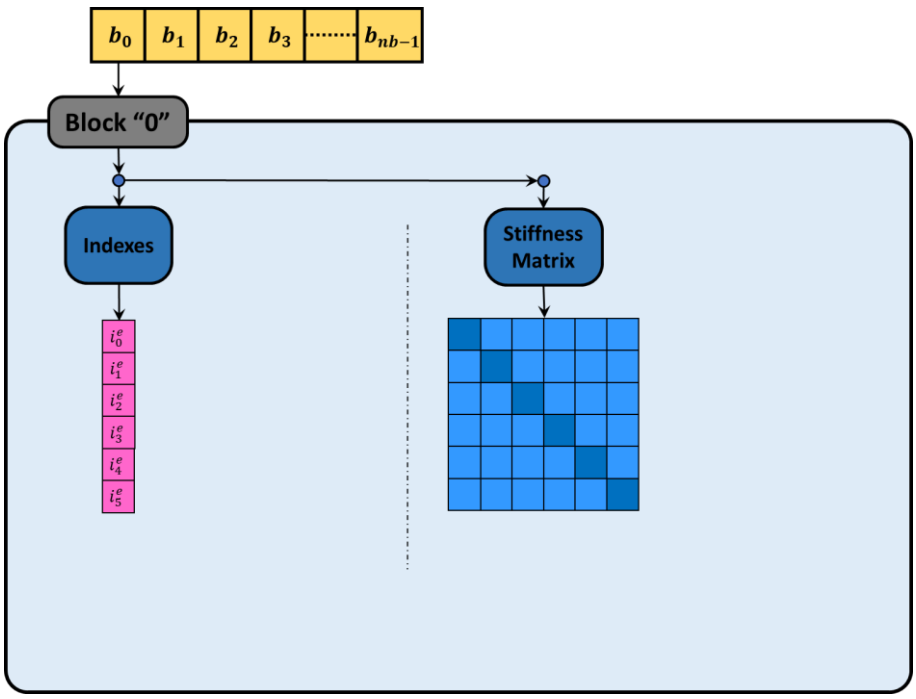

Fig. 9.3 – Each block has an array of indexes and a stiffness matrix. Source: own authorship.

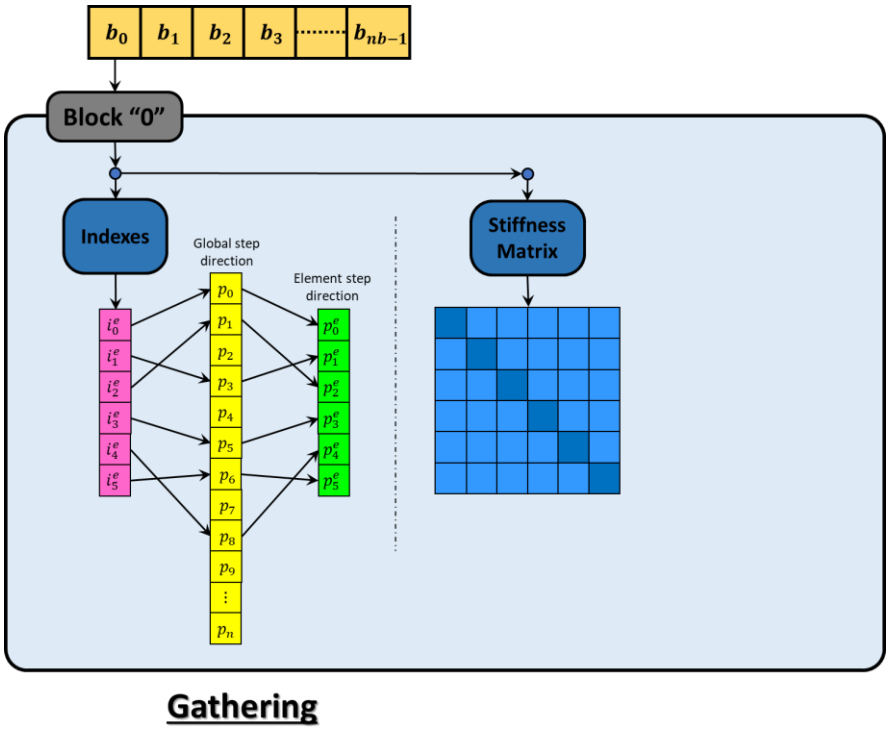

Fig. 9.4 – Gathering operation: the indexes are used to gather the local values of step directions. Source: own authorship.

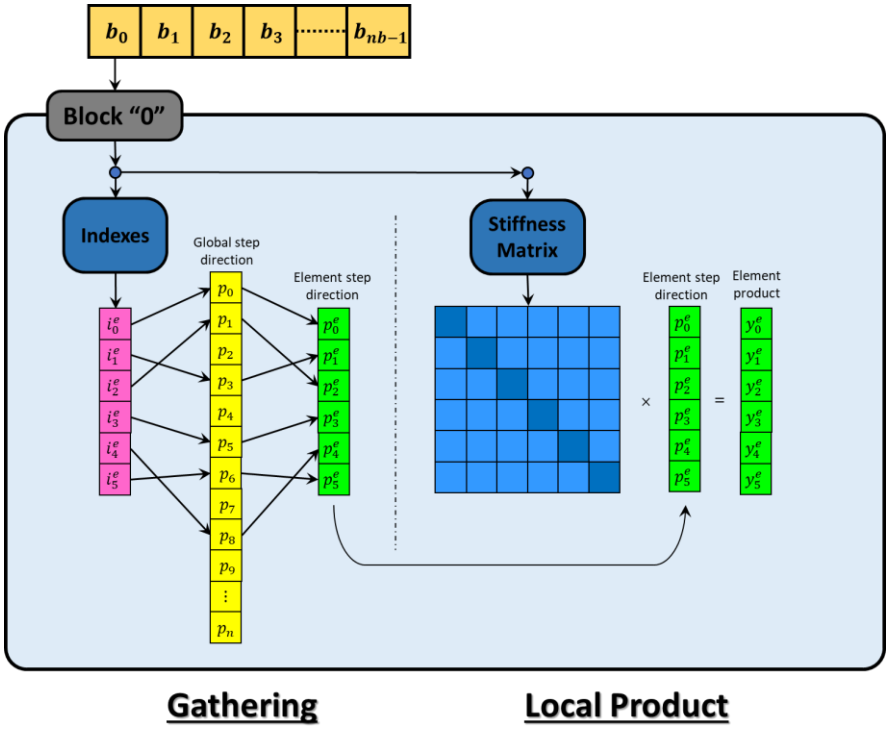

Fig. 9.5 – Local product operation. Source: own authorship.

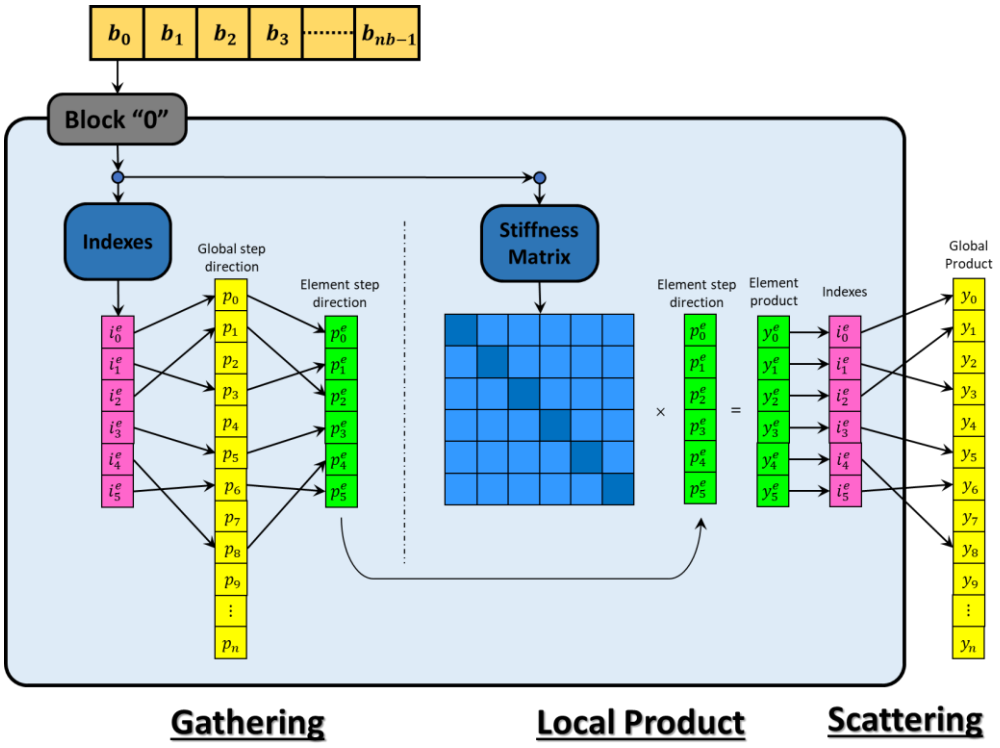

Fig. 9.6 – Scattering operation. Source: own authorship.

The gathering and local product operations are completely independent between the blocks and, therefore, they are easily parallelized. It is important to note that, in the gathering operation, the several threads will act concomitantly in the global array of step directions, but only with *reading* operations, that can be executed in parallel without the need for synchronization. The local matrix-vector product is strictly local and independent. On the other hand, the *scattering* consists of several *writing* operations into the same global result array. When it is performed in parallel, update overlays as the ones illustrated in Fig. 9.1 will occur, demanding synchronization techniques. Given the importance of the matrix-vector product operation, four different synchronization techniques were developed and explored in this work and are explained in detail in the next items.

9.3.1 Synchronization I: Global Array of Locks

The first adopted strategy to synchronize the scattering operation consists of a global array of locks. When applied to a block of code, the lock acts as a semaphore, allowing only one thread at a time to execute it. The first thread that reaches the lock obtains permission to execute the code and triggers it, while the others remain in hold, waiting

for the release of the lock. This method ensures no concomitant execution of the piece of code surrounded by the lock.

The global array of locks consists of creating a lock for each degree-of-freedom of the model. Table 9.3 shows the procedure for defining and allocating this array. It is important to note that, for each element of this array, the lock must be initialized (and destroyed in the deallocation at the end of the execution).

Table 9.3 – Definition and allocation of the array of locks.

```
/* Parameters*/
int nrThreads;   // Number of threads, user-specified value
int n;           // Final global dimension of the linear system
int nmax;        // Maximum block size

// Dynamically Allocation of the Global Array of OpenMP Locks
omp_lock_t* lock = new omp_lock_t[n];

// Each Lock of the Array Must Be Initialized
for(int i = 0; i < n; i++)
   omp_init_lock(&(lock[i]));

////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////

// Deallocation at the end of the EBE-PCG algorithm
for(int i = 0; i < n; i++)
   omp_destroy_lock(&(lock[i]));

delete[] lock;
```

Source: own authorship.

The algorithm for the matrix-vector product is then shown in Table 9.4. It starts with the creation of the parallel region. The code that is within this region is run in parallel by the specified number of threads. The first step consists of resetting the values of the $Output$ array, since it is reused from one iteration to another, avoiding, thus, unnecessary reallocations. The second step is the iteration over all blocks of the model. A dynamic scheduling for this for loop was chosen, because the blocks have varied sizes. The schedule clause specifies how the loop is distributed into the threads. When it is set as *dynamic*, this distribution is defined during the execution of the program, each thread receives a new chunk-sized block of loops when the previously received one has already been executed, with extra overhead associated, but more appropriate when the processes differ considerably in execution time. For each block, auxiliary variables are gathered,

such as block sizes and position in memory, and then it proceeds to the local product. The scattering is performed in the three last commands, when the lock is set, the global array is updated with the increment value and lastly the lock is unset.

Table 9.4 – Matrix-vector product using locks.

```
void MatrixVectorProduct(double*& Input, double*& Output)
{
   // Definition of the Parallel Region
   #pragma omp parallel num_threads(nrThreads)
   {
       // Cleaning the Output values
       #pragma omp for
       for (int i = 0; i < n; i++)
          Output[i] = 0.0;

       /* Gather, Product and Scatter */
       #pragma omp for schedule(dynamic, 1)
       for (int bk = 0; bk < nb; bk++)
       {
          int dimc = _bksiz_cond[bk]; // Final bk dimension
          int dimo = _bksiz_orig[bk]; // Orig. bk dimension
          int i1D  = _bk_ini_1D[bk];  // 1D initial position in memory
          int i2D  = _bk_ini_2D[bk];  // 2D initial position in memory

          /* Local Product */
          for (int i = 0; i < dimc; i++)
          {
             int id = _index[i1D + i]; // Global index to be updated

             double inc = 0.0; // Increment

             // Product
             for (int j = 0; j < dimc; j++)
                inc += p[i2D + i * dimo + j] * Input[_index[i1D + j]];

             omp_set_lock(&(lock[id]));     // Set lock for position "id"

             Output[id] += inc;             // Scatter

             omp_unset_lock(&(lock[id]));   // Unset lock for position "id"

          }
       }
   }
}
```

Source: own authorship.

This algorithm with locks is very simple and straightforward. However, for each increment in the global result array, two additional and costly operations are required (the lock set and unset operations).

9.3.2 Synchronization II: Local Copies

The second synchronization strategy consists of using local copies of the global result array. It has the advantage of completely eliminating the locks, since each thread works with its own array and the increments can be made without the risk of overlapping values. After the product operations, these local copies are summed in parallel into the global output array.

Table 9.5 shows the definitions and allocation of the local arrays, named as $localResult$. To avoid unnecessary reallocations, they are defined only once and reused at each iteration. Additionally, local copies of the input arrays were also created (that form the element local step directions), avoiding reallocations, for a better computational performance of the algorithm.

Table 9.5 – Definition and allocation of the local copy arrays.

```
/* Parameters*/
int nrThreads;   // Number of threads, user-specified value
int n;           // Final global dimension of the linear system
int nmax;        // Maximum block size

/* Definition and allocation of the local input arrays */
double** localInput = new double*[nrThreads]; // Local Input Array
for (int i = 0; i < _nrThreads; i++)
   localInput[i] = new double[nmax];

/* Definition and allocation of the local copy arrays */
double** localResult = new double*[nrThreads]; // Local Copy Array
for (int i = 0; i < _nrThreads; i++)
   localResult[i] = new double[n];
```

Source: own authorship.

The matrix-vector algorithm that uses the local copies as synchronization strategy is shown in Table 9.6. Right after the creation of the parallel region, thread local variables are defined and consists of: the current thread identification and pointers to the thread-respective input and result arrays. Setting these pointers right at the beginning optimizes variables indexing, with benefits in performance. The first procedure is then to reset the values of the local copy array. After that, it proceeds to the iteration over the blocks of the model. The gathering is performed updating the local input array, which is employed in the local product. All these operations, including the local scatter, are completely

thread-independent. Lastly, the local result arrays are summed together in the output array.

Table 9.6 – Matrix-vector product using the local copy arrays as synchronization methodology.

```
void MatrixVectorProduct(double*& Input, double*& Output)
{
   #pragma omp parallel num_threads(nrThreads)
   {
      int td = omp_get_thread_num();   // Thread number
      double* linp = localInput[td];   // Local Input Array
      double* lres = localResult[td];  // Local Result Array

      /* Cleans the Local Result Array */
      for (int j = 0; j < n; j++)
        lres[j] = 0.0;

      /* Gather, Product and Local-Scatter */
      #pragma omp for schedule(dynamic, 1)
      for (int bk = 0; bk < nb; bk++)
      {
         int dimc = _bksiz_cond[bk]; // Final bk dimension
         int dimo = _bksiz_orig[bk]; // Orig. bk dimension
         int i1D  = _bk_ini_1D[bk];  // 1D initial position in memory
         int i2D  = _bk_ini_2D[bk];  // 2D initial position in memory

         /* Gather */
         for (int i = 0; i < dimc; i++)
            linp[i] = Input[_index[i1D + i]];

         /* Block Product  */
         for (int i = 0; i < dimc; i++)
         {
            double inc = 0.0; // Increment variable

            for (int j = 0; j < dimc; j++) // Product
               inc += p[i2D + i * dimo + j] * linp[j];

            /* Local Scatter */
            lres[_index[i1D + i]] += inc;
         }
      }

      /* Parallel Summation of the Local Result Arrays */
      #pragma omp for
      for (int i = 0; i < n; i++)
      {
         Output[i] = 0.0;
         for (int j = 0; j < nrThreads; j++)
            Output[i] += localResult[j][i];
      }
   }
}
```

Source: own authorship.

This technique presents, however, scalability limitations. The gather, product and local-scatter are very scalable operations. The initial local array resetting and the final

arrays summation operations are, nevertheless, not accelerated with the increase of the number of threads. This is because the number of actions that they must perform increases linearly with the number of employed threads.

9.3.3 Synchronization III: Mapped Local Copies

The aforementioned limitations of the previous synchronization strategy motivated the development of an optimized algorithm. This new algorithm is based on the fact that, if the blocks are assembled into sets, each of which assigned to a different thread, it is possible to map all the degrees-of-freedom that each thread will modify, thereby reducing significantly the amount of operations performed in the initial resetting and in the final summation of the local copy arrays.

The first step consists of distributing the blocks into balanced sets. The simplest way to perform this is to divide the total number of blocks by the total number of threads, creating sets with the same number of blocks. However, this is not the best alternative, since the blocks have varying sizes, what would result in unbalanced sets and in consequent reduction of the overall available processing capacity. In addition to this, it is important to notice that the number of operations of the matrix-vector product is a quadratic function with respect to the dimensions of the blocks. The implemented method of division considers these question by employing the cumulative distribution function of the square of the block dimensions, as exemplified in Fig. 9.7 for a situation with 20 blocks and 4 threads, allowing the achievement of highly balanced sets.

Number of Threads	4

Block Nr	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
Block Dimensions	10	10	10	10	10	10	10	6	6	6	6	6	6	4	4	4	4	4	4	4
Squares of Block Dimensions	100	100	100	100	100	100	100	36	36	36	36	36	36	16	16	16	16	16	16	16
Cumulative distribution function - F(x)	10	20	30	40	50	60	70	76	82	88	94	100	106	110	114	118	122	126	130	134
Cumulative distribution function - $F(x^2)$	100	200	300	400	500	600	700	736	772	808	844	880	916	932	948	964	980	996	1012	1028

Total:	134
	1028
Pivot:	33.5
	257

Block Distribution based on F(x)	Set of Blocks 0	Set of Blocks 1	Set of Blocks 2	Set of Blocks 3
Block Distribution based on $F(x^2)$	Set of Blocks 0	Set of Blocks 1	Set of Blocks 2	Set of Blocks 3

Fig. 9.7 – The distribution of blocks into sets considers the squares of their dimensions. Source: own authorship.

With the sets of blocks, the algorithm proceeds then to mapping operation. A matrix of booleans is defined and initialized only with *false* values. As shown in Fig. 9.8, the dimensions of this matrix are given by the total number of degrees-of-freedom and the number of threads.

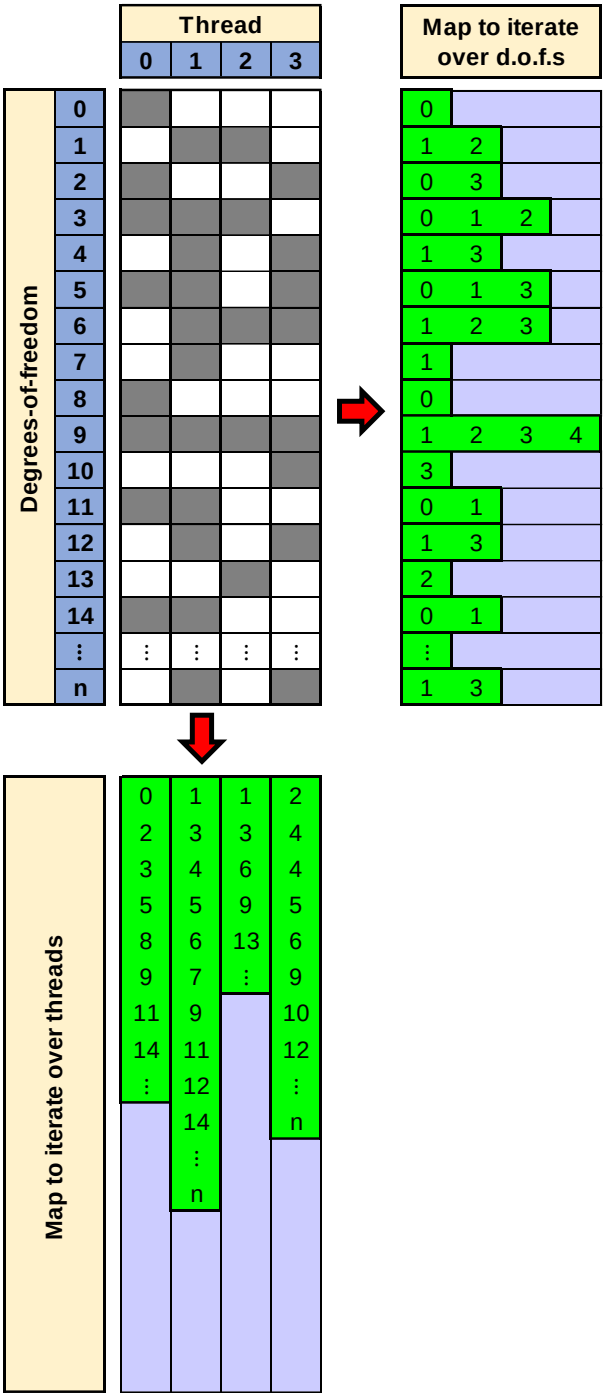

Fig. 9.8 – Table of booleans specifies the degrees-of-freedom that each thread modifies, with which it is possible to generate the maps. Source: own authorship.

In each column, the degrees-of-freedom in dark-gray are the ones modified by the respective thread. They are defined through the iteration along the set of blocks that belongs to the thread, and the degrees-of-freedom specified by the arrays of indexes have their values in the table switched from *false* to *true*. This table is employed to generate two different mappings. The first one specifies, for each available thread, all the degrees-of-freedom that are modified by it. The second mapping specifies, for each degree-of-freedom of the model, the threads that modify it.

The first mapping is then employed to improve the initial resetting of the local copy arrays. For each thread, only the degrees-of-freedom specified by the mapping have their values set to zero. The values of the remaining ones have no importance to the algorithm.

The second mapping is employed in the final summation of the local copy arrays, as shown in Table 9.7. For each of degree-of-freedom, only the threads specified by the mapping have their value summed into the output array.

Table 9.7 – Mapped-optimized parallel summation of the local copy arrays.

```
/* Parallel Summation of the Local Result Arrays */
/* The mapping-by-dof is used to eliminate unnecessary summations */
#pragma omp parallel for num_threads(nrThreads)
for (int i = 0; i < n; i++)
{
   int* loc_mapping_by_dof = mapping_by_dof[i];

   double val = 0.0;

   for (int j = 0; j < mbd_siz[i]; j++)
   {
      int td = loc_mapping_by_dof[j];
      val += localResult[td][i];
   }

   Output[i] += val;
}
```

Source: own authorship.

Small additional improvements were made in this algorithm, such as to utilize the output array itself as the local copy for the first thread, what already eliminates one global array to be summed in the final operation.

By reducing the unnecessary operations from the resetting and summation operations, some level of scalability is obtained in these steps from this new algorithm. However, the main advantage of this synchronization technique consists in the fact that it

is completely generic. It works independently of the indexing system, of the dimensions of the blocks and it will still be valid if new types of finite macroelements are included in the future.

9.3.4 Synchronization IV: Geometry- and Mesh-Based Mapped Solution

If the blocks are distributed in sets totally independent between each other, or if a large number of independent blocks is grouped into a set, it is possible to perform the aforementioned global scatter operations for these sets without worrying about synchronization, as the initial conditions ensure that there will be no simultaneous writing operations in the global result array.

However, the development of completely generic mappings procedures for the distribution of the blocks into independent sets is a very complex task. The contact elements increase significantly the matrix bandwidth and leave the indexing very dispersed, since one node can be in contact with several others (this occurs in the case of the bridge contact, for example, in which, for each wire of the armor, the same Fourier node is connected to another standard node). Nonetheless, by taking into account the characteristics of the geometry and mesh, it is possible to generate improved mappings of block distribution for certain predicted situations.

The explanation of this methodology starts then with the tensile armors, Fig. 9.9, which are composed by a predefined number of helical tendons, each of them modeled with the helical beam elements. It is interesting to note that the beam elements from a wire do not cause or suffer interference with the elements situated on the other wires (the contacts are handled separately). It means that, between the tendons, the elements are already distributed into independent sets. During the computation, each thread receives the elements from a specified wire and the global result array can be updated in parallel without any problem.

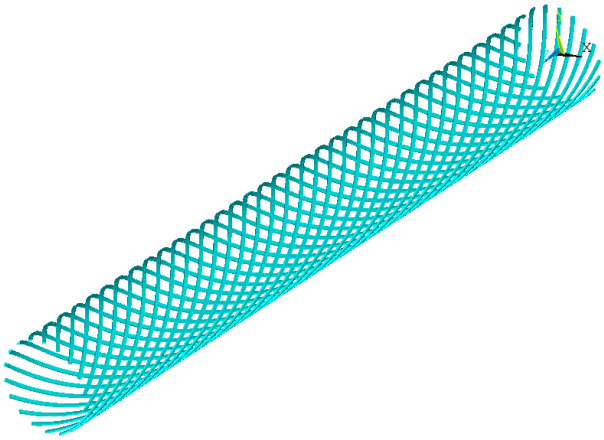

Fig. 9.9 – Tensile armor: the elements that belong to a wire are independent in relation to the remaining wires. The contacts are handled separately. Source: own authorship.

There is another possibility of element distribution for continuously connected beam, which is the case of the tensile armors. As illustrated in Fig. 9.10, the elements are grouped into two independent sets in this case. Then a two-step procedure is adopted, meaning that, firstly, the iteration must occur only on all elements of set 1. In the sequence, the iteration occurs for the elements from set 2.

Fig. 9.10 – Continuously connect beam elements can be grouped into two single independent sets.

The first approach has cache advantages. When a thread iterates through the elements of a wire, there is greater continuity in the numbering of blocks and degrees-of-freedom. However, this parallelization is limited by the total number of wires of the pipe. If a computer with some hundreds of cores is available, a portion of the processing capacity would be wasted, which does not occur in the second approach, that is completely generic in this sense.

With respect to the polymeric sheath, modeled with the Fourier expanded solids of revolution, the geometry is a rectangle and the mesh is perfectly mapped, as illustrated by the gray region in Fig. 9.11.

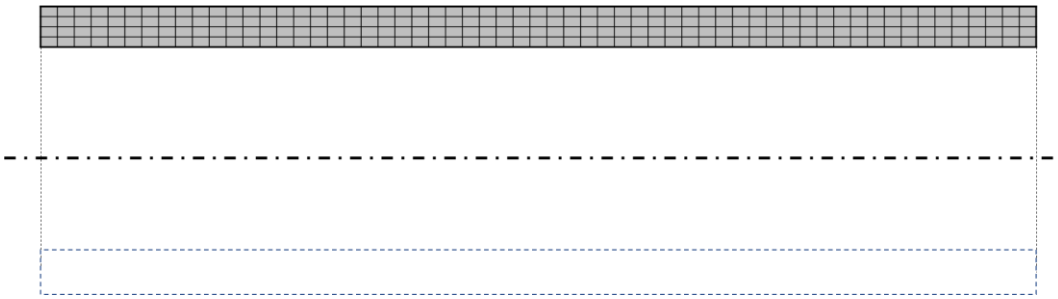

Fig. 9.11 – Geometry and mesh of the polymeric sheath. Source: own authorship.

In this case, the two-step procedure from Fig. 9.12 was developed and implemented. It is based on the fact that, in the first step, if considered only the columns designated by 1, there is no degree-of-freedom sharing among these columns. If each of them is assigned to a different thread, the global scatter can be simultaneously performed without any problem or need for synchronization. The second step is analogous the first by considering only the columns designated by the number 2.

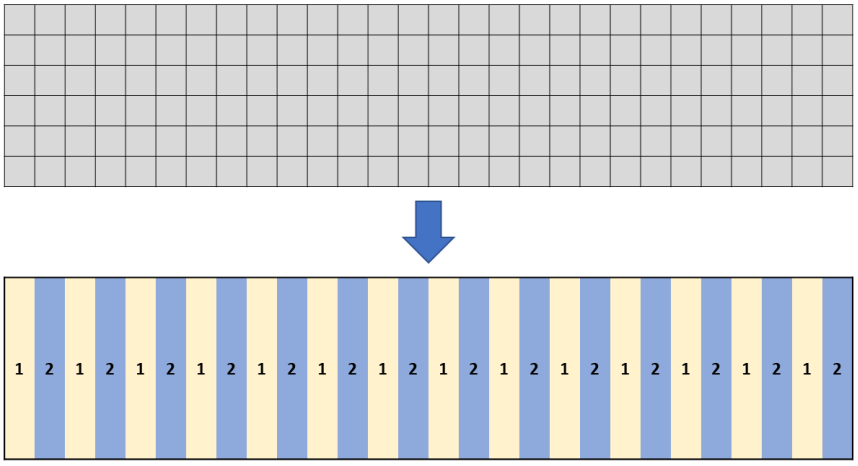

Fig. 9.12 – Two-step procedure: in the first step, only the columns designated by 1 are considered; in the second, the ones designated by 2. Source: own authorship.

A four-step procedure was also developed, as shown in Fig. 9.13. If the blocks numbering is well-behaved, this technique has great potential for parallelization methods that explore vectorization.

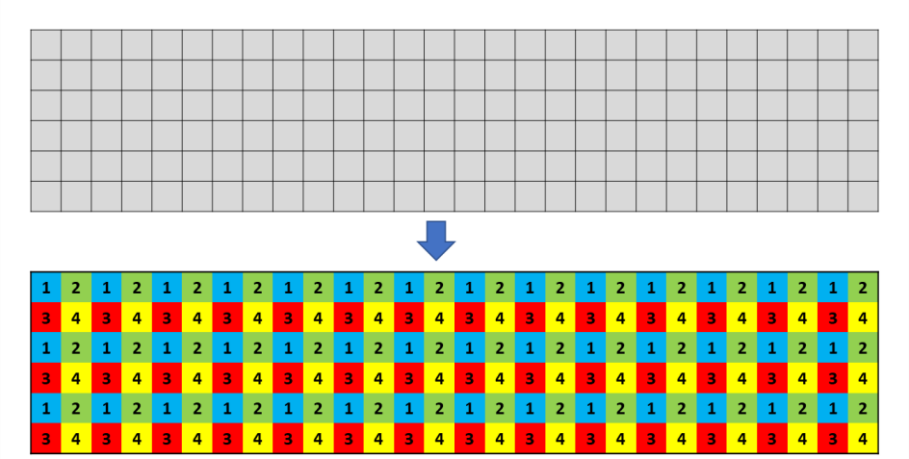

Fig. 9.13 – Four-step procedure. Source: own authorship.

Lastly, it is considered the distribution of the blocks relative the contact elements. As illustrated in Fig. 9.14, the contact pairs are dispersed through the model, making it difficult to determine logical patterns. The solution found consists of employing exclusively geometric properties, more specifically the fact that the pipe axial dimension is much higher in comparison to the others.

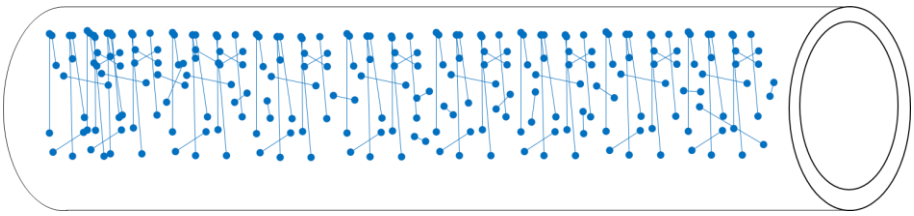

Fig. 9.14 – Illustrative representation of the contact pairs for a pipe model with two tensile armors and an external polymeric sheath. Source: own authorship.

This strategy enables the creation of subdomains with minimized overlapping frontiers, as shown in Fig. 9.15. In this case the pipe is axially subdivided into four equidistant domains.

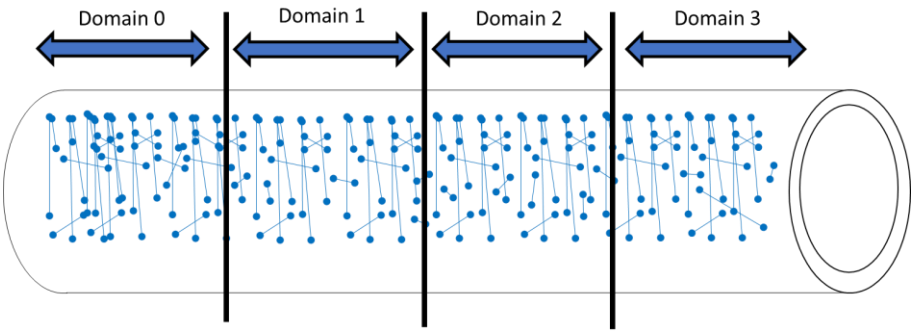

Fig. 9.15 – Domain subdivision. Source: own authorship.

For each contact pair, the nodal coordinates are verified and classified into a domain. The ideal (and most frequent) case is when both nodes are located in the same domain. If this does not occur, i.e., if each node is situated in different domains, the block corresponding to this element is identified and receives special treatment with synchronization. It is interesting to note that, if all layers have the same number of axial divisions, this problematic situation does not occur.

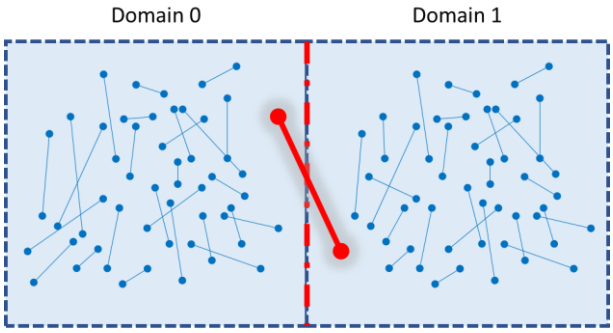

Fig. 9.16 – Problematic situation: contact pair located between two different domains. Source: own authorship.

It is important to note that this parallelization strategy is not generic, since the iteration maps are based on the geometry and mesh of the model. If new finite macroelements or new rearrangements between layers are developed in the future, it is very likely that this algorithm will have to be complemented in order to contemplate the new scenarios. Despite this, it is also important to note that, only by rearranging and creating an intelligent sequence of execution of the local matrix-vector products, it was possible to completely eliminate the synchronization mechanisms, making this algorithm very efficient and fast in computational terms.

In the next chapter, numerical results are presented, in conjunction with a detailed analysis of the execution times and scalability, allowing thereby the comparison of the parallelization strategies presented in this chapter.

10 Results

In this chapter, the results of the implemented EBE-PCG algorithm are presented, with emphasis on simulation time, scalability and memory consumption. In addition to the PCG algorithm itself, the four synchronization strategies for the matrix-vector product discussed in detail in the previous chapter have also been evaluated and compared.

Before the results, however, a simplified model of flexible pipe is introduced. It consists of a flexible pipe modeled with the finite macroelements presented in Chapter 2 with the objective of testing and validating the implementation of the EBE-PCG solver.

10.1 Finite Macroelement Model

The flexible pipe illustrated in Fig. 10.1 was modeled with the finite macroelements from Chapter 2 with the objective of testing the implementation of the EBE-PCG algorithm. It contains three layers, from the inner to the outermost: an inner and an outer tensile armor layers; and an external polymeric sheath. This pipe is simplified with respect to the total number of layers (only three), but all of them are consistent with those of a possible real pipe with 4 inches of internal diameter. All the characteristics and properties of these layers are presented in greater detail in the next items.

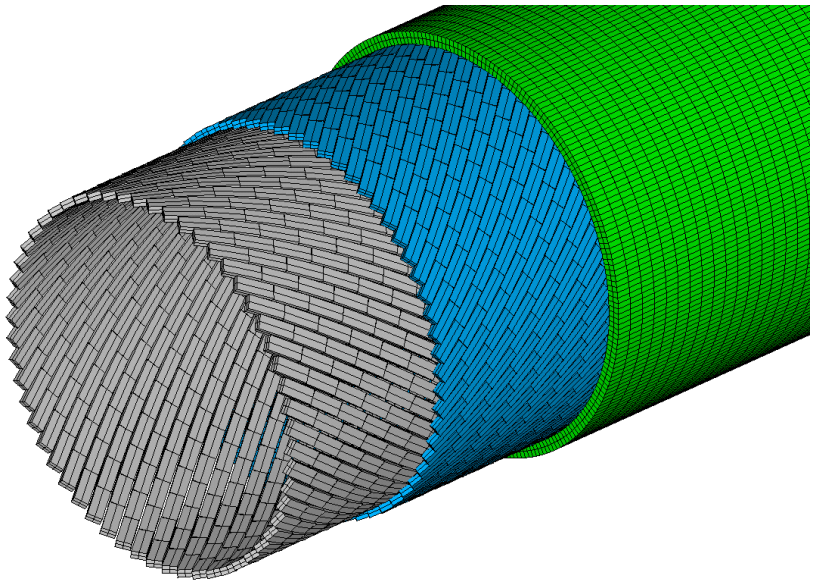

Fig. 10.1 – Simplified model of flexible pipe. Image generated in ANSYS®. Source: own authorship.

10.1.1 Inner Tensile Armor Layer

The inner tensile armor layer, Fig. 10.2, consists of 56 helically extruded metallic tendons of rectangular shaped cross-sections. The tendons are modeled with the helical beam element from item 2.2, so that the only control parameter of the element mesh is the number of axial divisions. All geometric and material properties are found in Table 10.1.

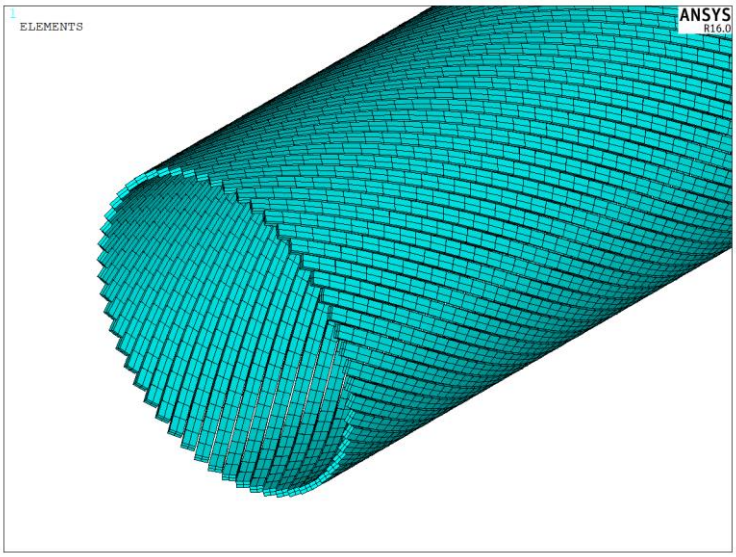

Fig. 10.2 – Inner tensile armor layer. Image generated with ANSYS®. Source: own authorship.

Table 10.1 – Parameters of the inner layer of tensile armor. Source: own authorship.

Parameter	***Value***
Length (mm)	1,692.00
Mean Radius (mm)	101.25
Lay Angle (deg.)	36.00
Cross-section W x H (mm)	8.00 x 4.00
Number of Tendons	56
Material Young Modulus (MPa)	207,000.00
Material Poisson Ratio	0.30

10.1.2 Outer Tensile Armor Layer

The outer tensile armor layer, Fig. 10.3, consists of 63 helically extruded metallic tendons of rectangular shaped cross-sections. The tendons are modeled with the helical beam element from item 2.2, so that the only control parameter of the element mesh is the number of axial divisions. All the geometric properties of this layer are found in Table 10.2, as well as the material properties.

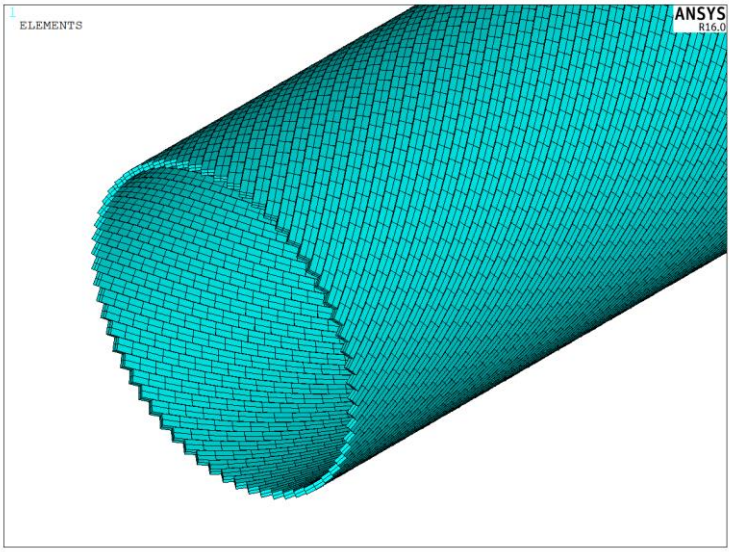

Fig. 10.3 – Outer tensile armor layer. Image generated with ANSYS®. Source: own authorship.

Table 10.2 – Parameters of the outer layer of tensile armor.

Parameter	***Value***
Length (mm)	1,692.00
Mean radius (mm)	105.25
Lay angle (deg.)	-38.00
Cross-section W x H (mm)	8.00 x 4.00
Number of tendons	63
Material Young Modulus (MPa)	207,000.00
Material Poisson Ratio	0.30

Source: own authorship.

10.1.3 External Polymeric Sheath

The outermost layer is a cylindrical sheath, as illustrated in Fig. 10.4, made of polymeric material.

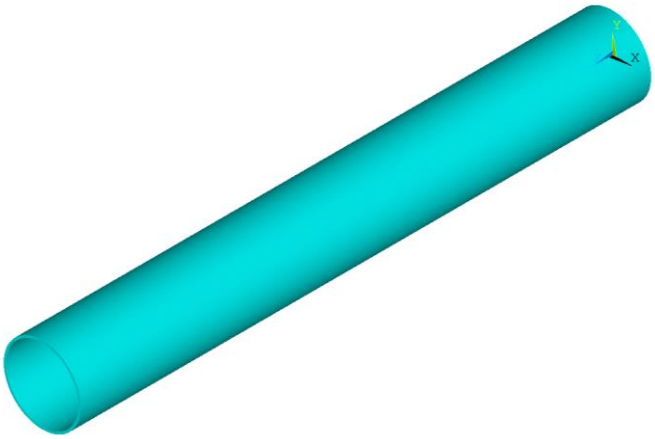

Fig. 10.4 – Polymeric sheath. Image generated with ANSYS®. Source: own authorship.

It is important to note that, for being modeled in PipeFEM with the Fourier expanded cylindrical element (2.1), the element mesh is a simplified surface, as shown in the gray region of Fig. 10.5. It has three control parameters, the number of axial and radial divisions and the maximum expansion order of the Fourier series, which must be varied to verify the numerical convergence of the model.

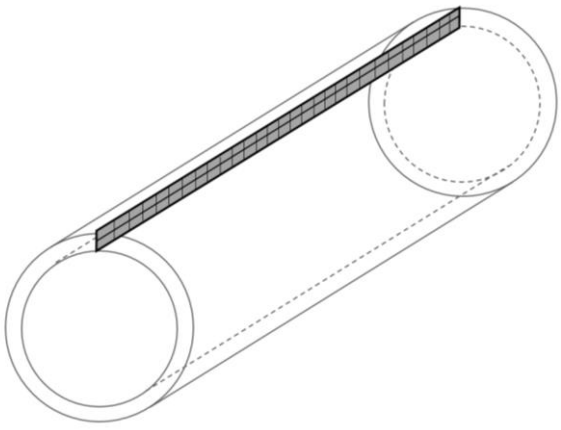

Fig. 10.5 – The element mesh is illustrated in dark grey. Source: own authorship. Source: own authorship.

All material and geometric properties are found in Table 10.3. Although the employed material (HDPE) has non-linear characteristics, it was employed a linear elastic material model, since material non-linearities are not explored by the program.

Table 10.3 – Parameters of the polymeric sheath layer.

Parameter	***Value***
Length (mm)	1,692.00
Mean radius (mm)	110.75
Thickness (mm)	7.00
Material Young Modulus (MPa)	570.88
Material Poisson Ratio	0.45

Source: own authorship.

10.1.4 Contacts Between Layers

Before discussing the contacts, a brief summary of the layers is available in Table 10.4, including the finite macroelements used to mesh them.

Table 10.4 – Summary of the layers.

ID	***Layer***	***Finite Macroelement***
1	*Inner Tensile Armor (10.1.1)*	*Curved Helical Beam (2.2)*
2	*Outer Tensile Armor (10.1.2)*	*Curved Helical Beam (2.2)*
3	*Polymeric Sheath (10.1.3)*	*Fourier Cylinder (2.1)*

Source: own authorship.

All the interactions between these layers were modeled as completely rigid, that is, without relative displacements between the nodes that compose the contact pairs. The contact between the inner and the outer layers of tensile armor was done with conventional node-to-node bonded contact elements, since both layers have the same standard nodal type. As for the contact between the outer tensile armor layer and the polymeric sheath, due to the nodal different natures, it was employed the bridge contact elements from item 2.3. All these contacts are summarized in Table 10.5.

Table 10.5 – Contact between layers.

Contact ID	Between layers	Finite Macroelement
1	1 and 2	Bonded Contact
2	2 and 3	Bridge Contact (2.3)

Source: own authorship.

10.1.5 Meshes

In order to evaluate the computational performance (mainly the scalability) of the implementation for large-scale models, two different levels of mesh refinement were defined, named as "*Mesh A*" and "*Mesh B*". All parameters and statistics of these two meshes are shown in Table 10.6 and Table 10.7, respectively. For Mesh B, a non-round value was chosen as the number of axial divisions, with the objective of avoiding any possible influence on scalability due the multiplicity between this number of divisions and the number of employed threads. In addition to this, a complete convergence analysis is performed in item 10.7, in which several other combinations of mesh parameters are tested. As it will be seen, Mesh A and Mesh B can be considered very refined, but it is worth remembering once again that they were created with the purpose of evaluating the behavior of the implementation for large-scale situations.

Table 10.6 – Mesh A.

Parameters	***Value***
Axial divisions in the tensile armors	200
Axial divisions in the cylinder	400
Radial divisions in the cylinder	2
Fourier order	4
Number of nodes	48,922
Number of elements	63,744
Number of d.o.f.s	318,795
Number of blocks	66,944

Source: own authorship.

Table 10.7 – Mesh B.

Parameters	***Value***
Axial divisions in the tensile armors	973
Axial divisions in the cylinder	1946
Radial divisions in the cylinder	2
Fourier order	6
Number of nodes	237,534
Number of elements	256,221
Number of d.o.f.s	1,617,957
Number of blocks	279,573

Source: own authorship.

10.2 Hardware

All results from this chapter were generated with the workstation available in LMO (Laboratory of Offshore Mechanics) from the Polytechnic School of the University of Sao Paulo, for which the complete specifications are listed in Table 10.8. This workstation has an Intel® based motherboard that has the *Turbo Boost Technology*.

Table 10.8 – Workstation specifications: 16 real cores available for scalability tests.

Workstation Super Micro: Super Server SYS-7048R-TR
Motherboard X10DRi (Intel® C612 chipset; 16x DIMM slots)
(2x) Intel Xeon E5-2630v4 (8cores, 25M Cache, 2.20 GHz)
256GB (8x32GB) RAM Memory DDR4-2400 ECC LRDIMM
12TB (2x6TB raid 0) Seagate 3.5" 7.200 RPM 128MB cache 6GB/s
SSD Intel P3500 1.2TB, NVMe PCIe 3.0 x4, MLC HHHL AIC 20nm 0.3DWPD
NVIDIA PNY Quadro K1200 4GB DDR5 PCIe 2.0
Windows 10 Pro English

Source: own authorship.

According to (INTEL, 2018), *"Intel® Turbo Boost Technology accelerates processor and graphics performance for peak loads, automatically allowing*

processor cores to run faster than the rated operating frequency if they're operating below power, current, and temperature specification limits. Whether the processor enters into Intel® Turbo Boost Technology 2.0 and the amount of time the processor spends in that state depends on the workload and operating environment".

This technology automatically increases the clock of the processor when it is at low load, which ends up accelerating the solution when few threads (or processors) are being used in the parallelization. Although very useful in practical situations, the Turbo Boost feature misrepresents the scalability results and creates the false impression that the results are not good enough. In this way, for a fair performance comparison, the Turbo Boost was disabled directly in the machine's BIOS.

10.3 Definition of Speedup

In a perfectly parallel code, the simulation time is reduced by half by doubling the number of processors. However, this is not always achievable. System overheads, synchronization points, barriers, sequential passages, among others, are examples of situations that decrease the efficiency of the parallelization. In this way, in order to measure the success of the implementation, the parallel speedup is defined and given by the formula:

$$s = {T_1}/{T_p} \qquad \text{Eq. 10.1}$$

where:

- T_1 – is the sequential execution time or on 1 processor;
- T_p – denotes the execution time on P processors.

The speedup is used to evaluate the quality of the implementation and to compare different parallelization strategies.

10.4 Results of the Computation of the Element Stiffness Matrices

The computation of the element stiffness matrices is characterized by the high number of mathematical operations. However, since the elements are independent between each other, this operation is easily and also highly scalable. The simulation time of the computation of the element stiffness matrices in function of the number of threads for Mesh A (Table 10.6) is shown in Fig. 10.6, while the speedup is represented in Fig. 10.7. Before analyzing the results, it is important to discuss a feature from OpenMP, called *schedule*, which defines the loop distribution to the threads. In the implementation, two scheduling options were tested: the *static*, in which the loop is divided into equal-sized chunks (or as close to) with little system overhead; and *dynamic*, in which each thread receives a new chunk-sized block of loops when the previously received one has already been executed, with extra overhead associated, but more appropriate when the processes differ considerably in execution time. With scheduling options already clarified, it proceeds then to the analysis of the results. Both graphs show that the static scheduling has limited scalability and is considerably slower than the dynamic one. Two reasons justify the low computational performance of the static scheduling: several types of finite elements, with different formulations and distinct stiffness matrices computation times; and the iteration through the elements of the model, as shown in Table 10.9, since an element may have only one stiffness matrix, which is the case of the helical beam, or may have one stiffness matrix for each Fourier expansion order, which is the case of the solids of revolution. These two reasons generate an imbalance in the static scheduling, making the dynamic one the most appropriate solution, despite the extra overhead, achieving results very close to the perfect scalability and attesting the quality of the implementation. By using the EBE Matrix object, the necessary memory is allocated only once at the beginning of the solver, since the dimensions of the blocks are known beforehand, which avoids unnecessary reallocations and contributes to performance.

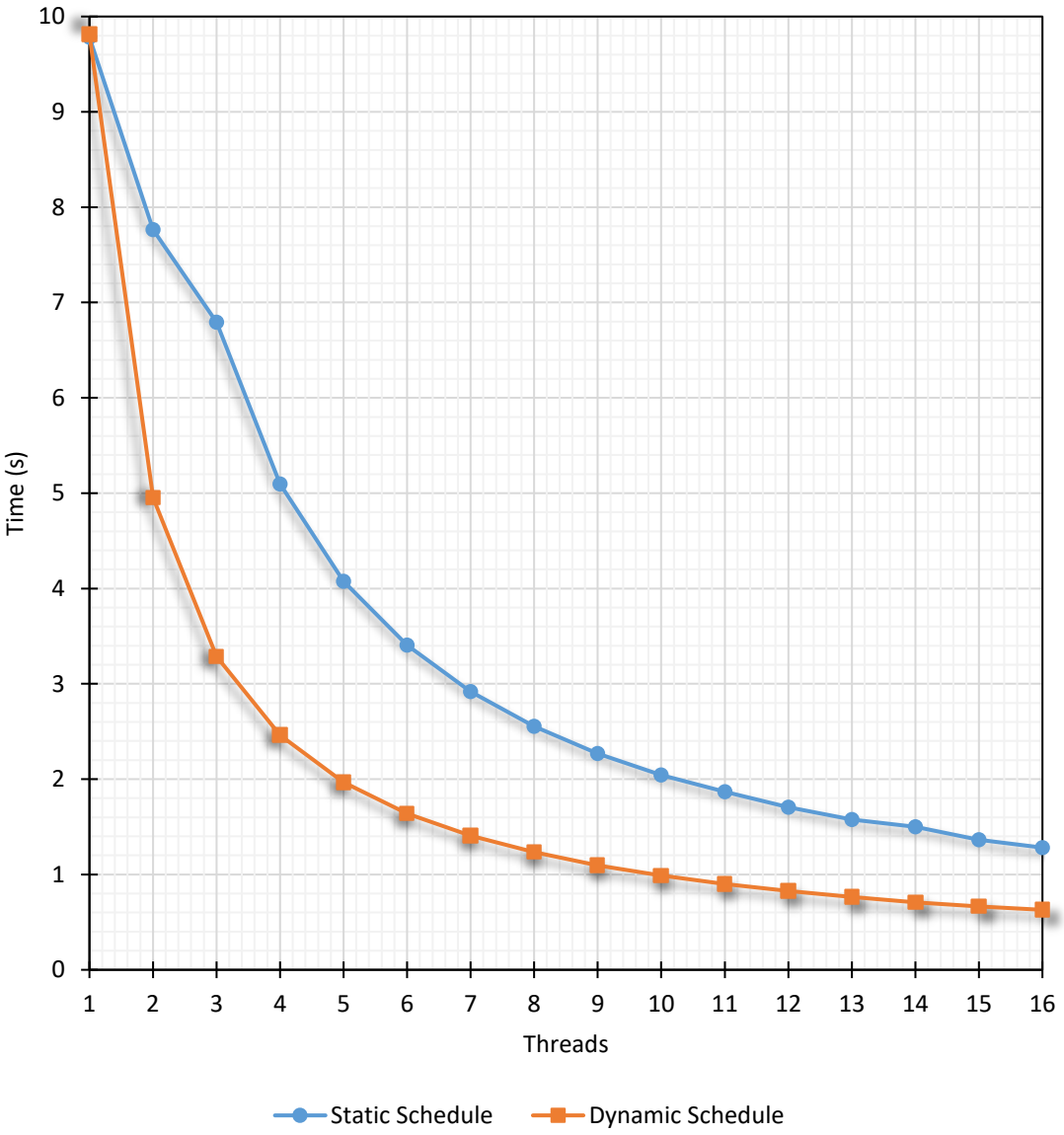

Fig. 10.6 – Simulation time of the computation of the element stiffness matrices for Mesh A (Table 10.6).

Source: own authorship.

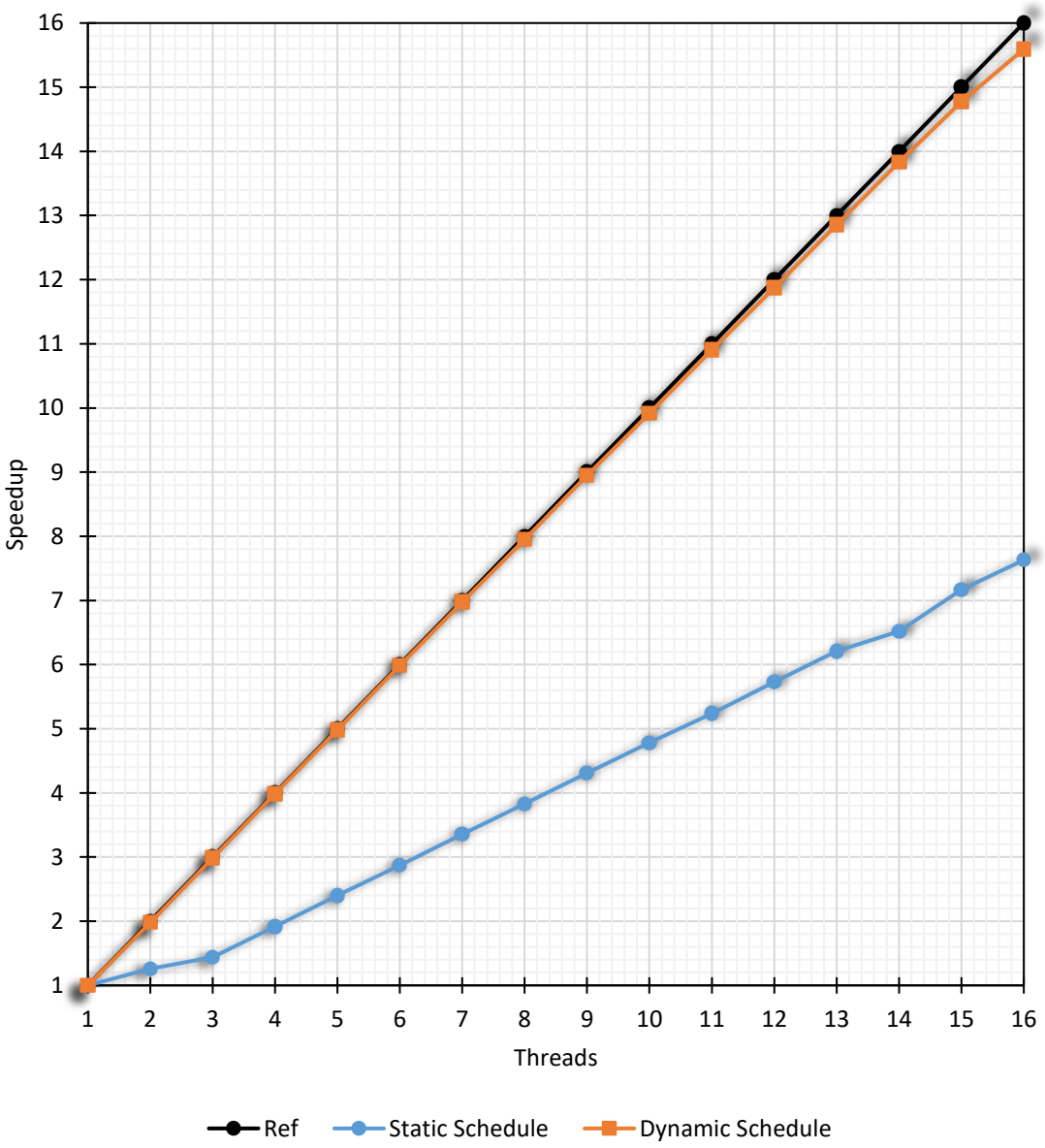

Fig. 10.7 – Speedup of the computation of the element stiffness matrices for Mesh A (Table 10.6).

Source: own authorship.

Table 10.9 – Iteration procedure to compute the element stiffness matrices.

1. Iteration over all elements

    1.1. Iteration over all orders/blocks of the element

        1.1.1. Computation of the stiffness matrix for the specified element and order

Source: own authorship.

Analogous results were also obtained for Mesh B and are available in Fig. 10.8 and Fig. 10.9. All analyses and conclusions made for the previous case remain valid for this new mesh configuration.

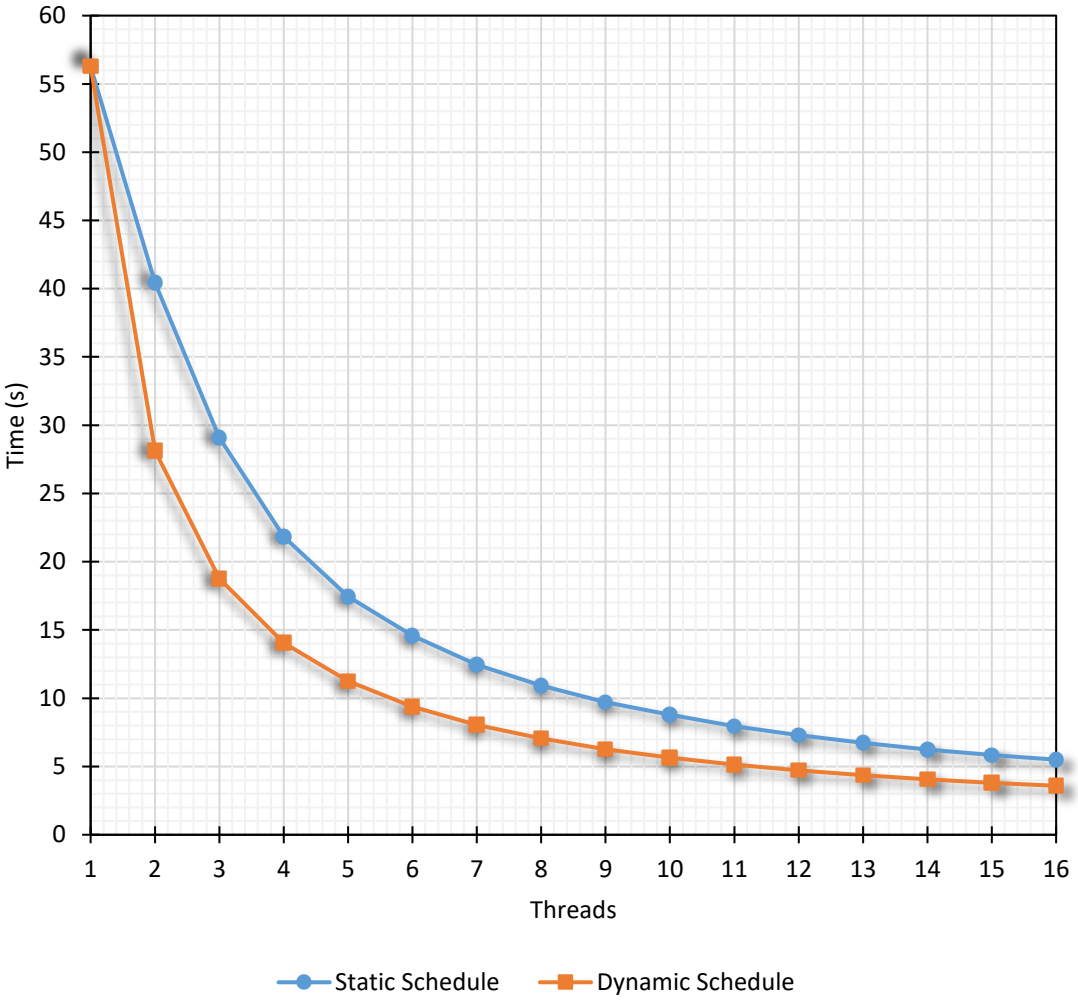

Fig. 10.8 – Simulation time of the computation of the element stiffness matrices for Mesh B (Table 10.7). Source: own authorship.

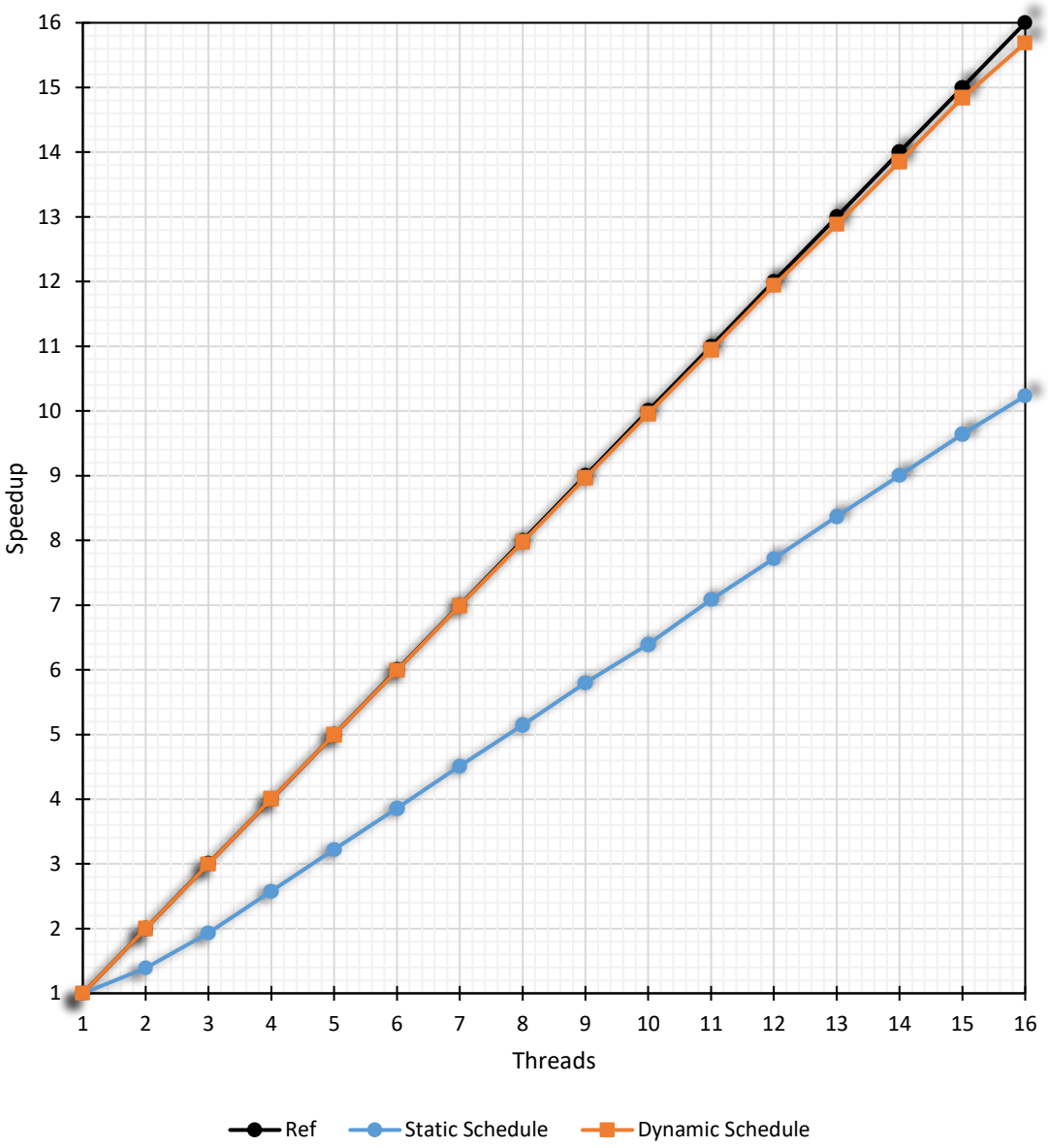

Fig. 10.9 – Speedup of the computation of the element stiffness matrices for Mesh B (Table 10.7). Source: own authorship.

In the EBE method, this operation is already completed after allocating and computing the element stiffness matrices. In the conventional finite element method, however, the sparse global stiffness matrix must be assembled, an operation that, if not properly implemented, may compromise seriously the scalability.

10.5 Results of the EBE Matrix-Vector Product

For being executed at each iteration, the scalability of matrix-vector product affects directly the efficiency of the EBE-PCG algorithm. In this way, due to its importance, it

was decided to individually analyze the efficiency of this operation for the different types of developed synchronization mechanisms (discussed in Chapter 9), whose nomenclature used in the graph legends is in Table 10.10.

Table 10.10 – Synchronization methods.

Symbol	***Synchronization Method***
SYNC 1	*Locks*
SYNC 2	*Local Copies*
SYNC 3	*Mapped Local Copies*
SYNC 4	*Geometric- and Mesh-Mapped*

Source: own authorship.

The execution time, in milliseconds, of the matrix-vector product in function of the number of threads for Mesh A (Table 10.6) is illustrated in Fig. 10.10, while the speedup curves are shown in Fig. 10.11. These results show that the synchronization method based on geometric and mesh-mappings is the fastest alternative to perform the EBE matrix-vector product. The first two methods show limited scalability, mainly the method based on local copies, because, as aforementioned, the number of operations to clean and to synchronize the local copies grows linearly with the number of threads. This problem is mitigated in the third synchronization strategy, since the mapping of the local copies allows a reduction in the number of operations and thus obtain greater scalability. In this third curve, there is an unexpected increase in the speedup for 10 threads, what is suspected to be due the fact that the number of threads is a round and multiple number of the total axial divisions.

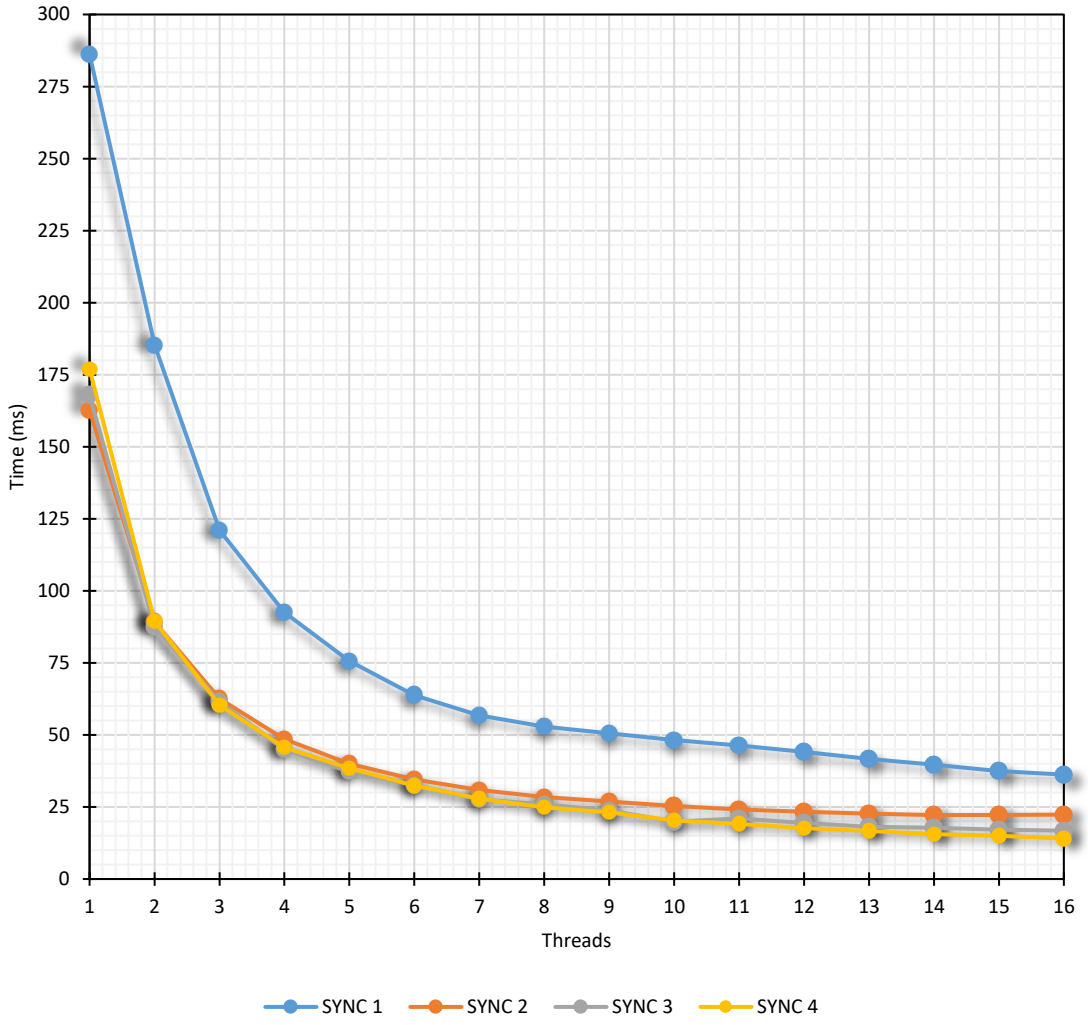

Fig. 10.10 – Simulation time, in milliseconds, of the matrix-vector product for Mesh A (Table 10.6). Source: own authorship.

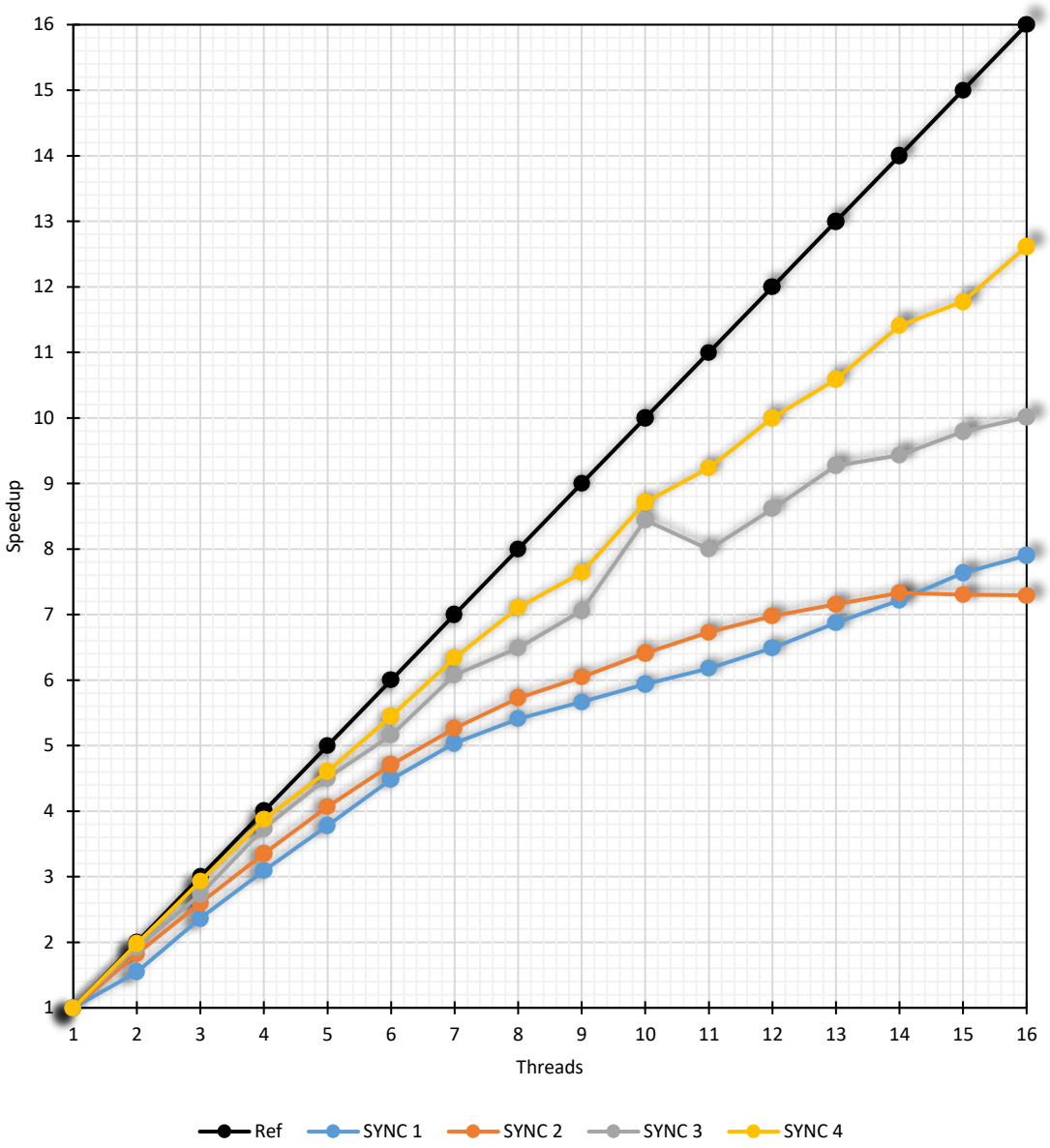

Fig. 10.11 – Speedup of the matrix-vector product operation for Mesh A (Table 10.6). Source: own authorship.

Aiming the understanding of the effect of model size on the efficiency of the implemented EBE matrix-vector product, this operation was also tested for Mesh B (Table 10.7). The execution times in function of the number of threads for the different synchronization mechanisms is shown in Fig. 10.12, while the speedup curves are in Fig. 10.13.

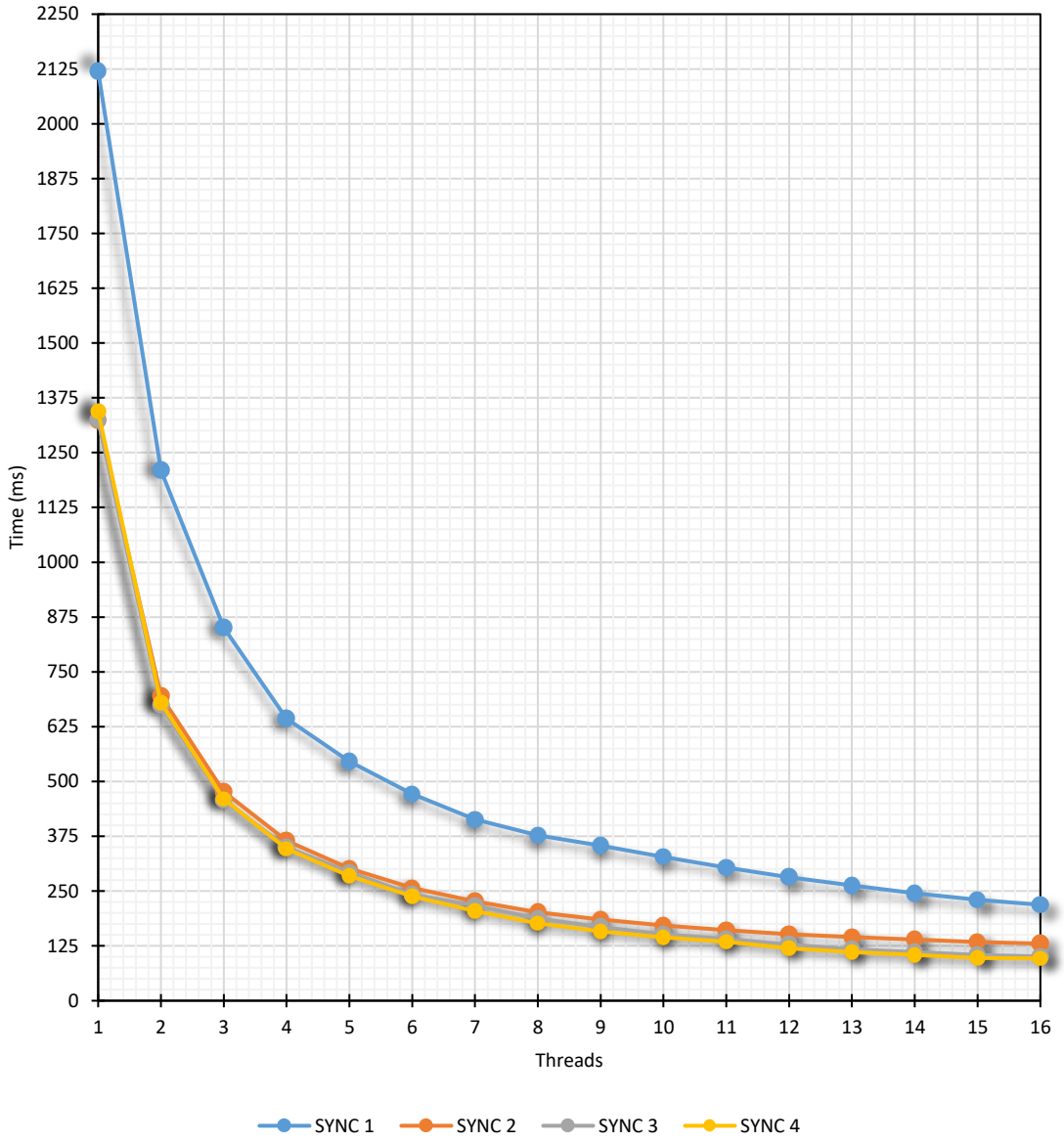

Fig. 10.12 – Simulation time, in milliseconds, of the matrix-vector product for Mesh B (Table 10.7). Source: own authorship.

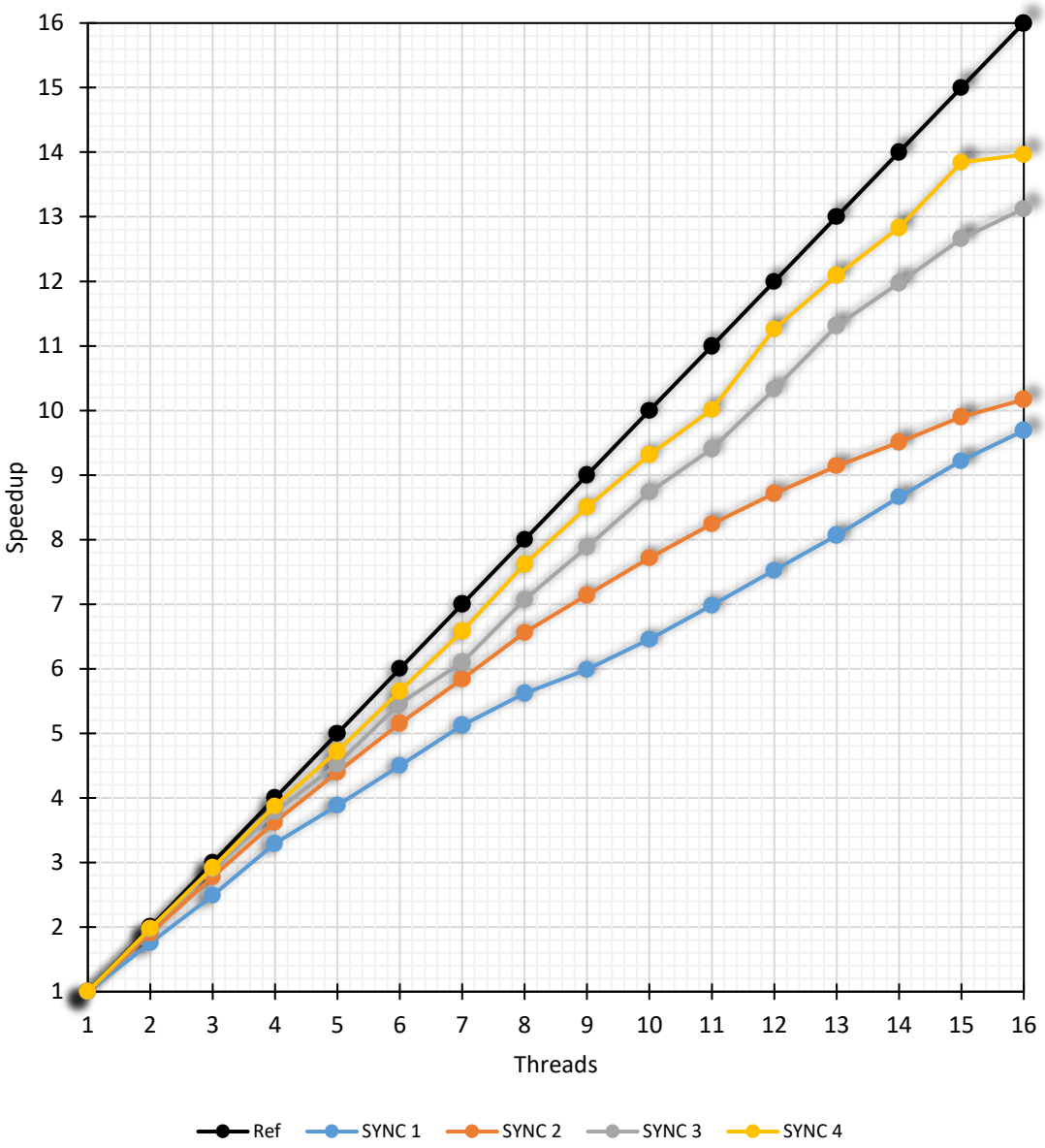

Fig. 10.13 – Speedup of the matrix-vector product operation for Mesh B (Table 10.7). Source: own authorship.

The sequence of the fastest synchronization methods has remained the same in relation to Mesh A, being the fourth method the most efficient of them. Nevertheless, it is interesting to note an increase in the speedup curves, mainly for the third and fourth synchronization methods. Additionally, there was a performance approximation from the third to the fourth method, which may justify the adoption of the mapped local copies as the synchronization strategy for large-scale models, due to its generic nature, since the fourth method requires external mappings based on geometry and mesh.

A comparison between Mesh A and Mesh B is available in Table 10.11. Despite being in the same order of magnitude, the increase of the simulation time was slightly

higher than the increase of the number of degrees-of-freedom and the number of blocks. However, when comparing both meshes, it is important to consider the fact that the increase of the Fourier expansion order generate element stiffness matrices of larger dimensions (for bridge contact element, for example), which consequently demands more mathematical operations for the execution of the product. By dividing the simulation time by the number of mathematical operations to perform the matrix vector product, although simplistic, it shows that the relative cost has been maintained, indicating that the implementation of this matrix-vector product is suitable for large-scale models.

Table 10.11 – Result comparison between Mesh A and Mesh B.

Parameter	***Mesh A***	***Mesh B***	***Variation***
Number of d.o.f.s	318,795	1,617,957	4.1x
Number of blocks	66,944	279,573	3.2x
Minimum matrix-vector product simulation time	14.03 s	96.29 s	5.9x
Number of mathematical operations to execute the matrix-vector product	34,261,992	276,326,274	7.1x
Simulation time / Nr of Operations	4.09E-07	3.48E-07	-0.15x

Source: own authorship.

10.6 Results of the EBE-PCG Algorithm

The analysis of the results proceeds then to the implemented EBE-PCG algorithm. The graph from Fig. 10.14 consists of the simulation time, in seconds, of the PCG algorithm in function of the number of threads for Mesh A (Table 10.6), measured for the different developed synchronization mechanisms. As expected, these results show that the fourth synchronization strategy is the most efficient alternative to be applied to the PCG algorithm, taking around 315 seconds to solve a model with approximately 320,000 degrees-of-freedom, with a total consumption of 421MB of RAM memory.

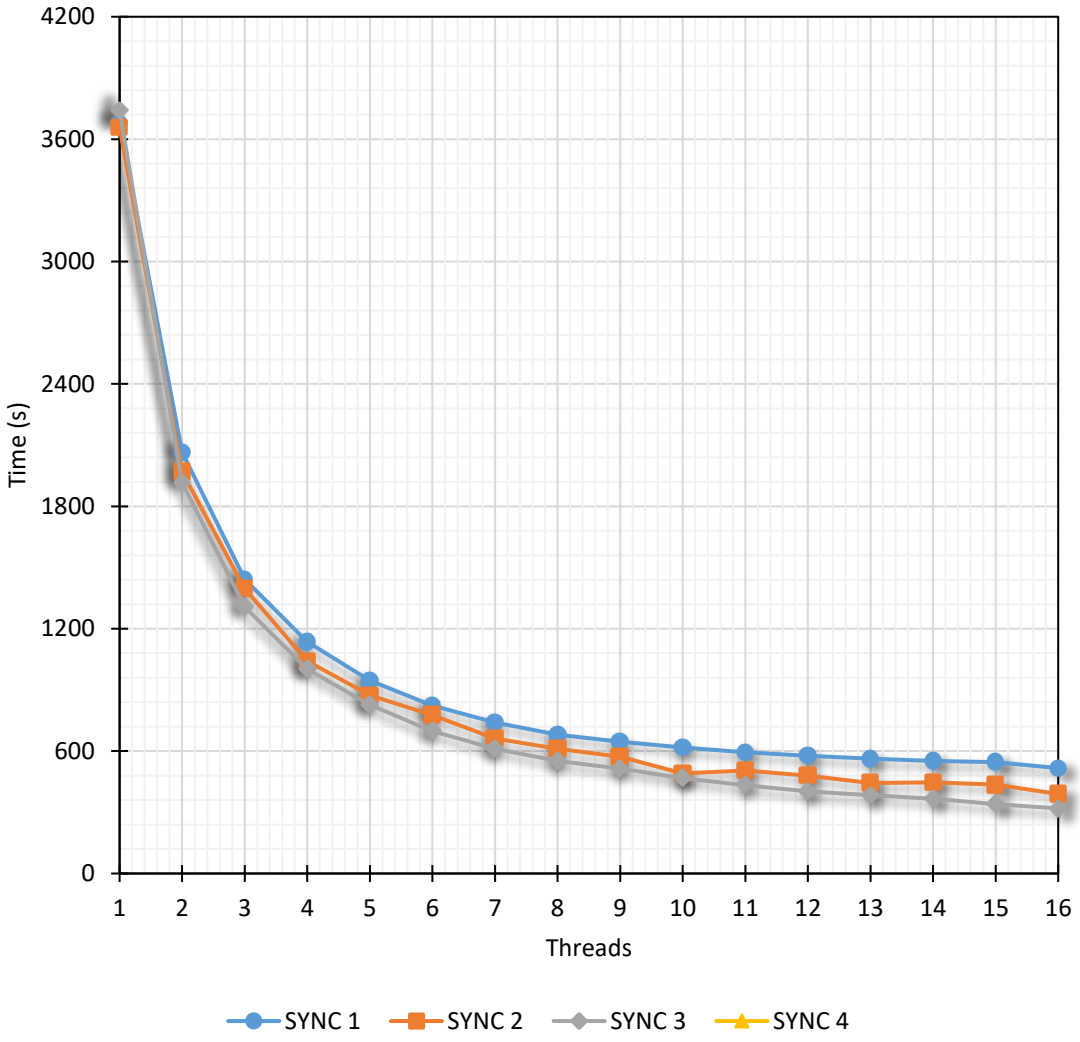

Fig. 10.14 – Simulation time, in seconds, of the PCG algorithm in function of the number of threads for Mesh A (Table 10.6). Source: own authorship.

The speedup curves from Fig. 10.14 show a good scalability of the fourth implemented synchronization method, so that the algorithm could be further accelerated if more computational resources were available. It can also be noted a slight reduction of the total scalability in comparison to the matrix-vector product results from item 10.5. This is because, although the scalability of the PCG algorithm is strongly influenced by the matrix-vector product, additional system overheads are introduced by the other operations of the method.

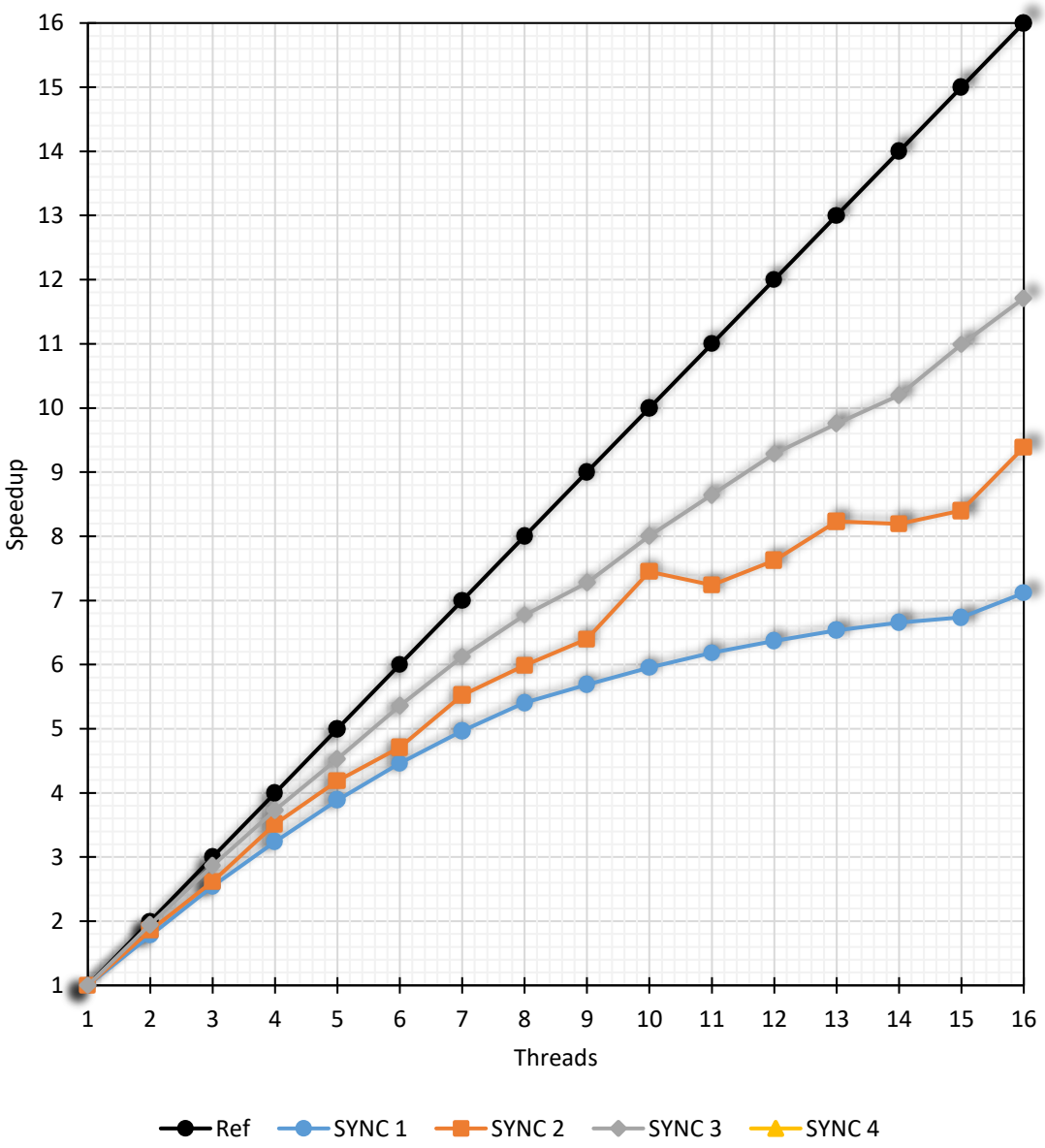

Fig. 10.15 – Speedup of the PCG algorithm in function of the number of threads for Mesh A (Table 10.6). Source: own authorship.

The PCG algorithm is composed of a series of very scalable and parallelizable operations, but that depend on the values of the previous ones. In other words, the iterative scheme is composed of a series of operations that cannot be performed concomitantly. The *reduce* operations, for example, necessary to evaluate the values of *alpha* and *beta*, represent synchronization points, with associated system overhead, but these costs are diluted as the size of the model increases. In addition to this, improved scalability of the matrix-vector product was obtained with the larger model from Mesh B. Therefore, an increase of the scalability of the implemented EBE-PCG algorithm is expected as model size increases.

Fig. 10.16 illustrates the residue curve as function of the number of iterations for Mesh A. This curve is important for the understanding of the convergence rate of the problem and its format is directly associated with the used preconditioner. To achieve a residual less than 1.0E-06, around 22,000 iterations were necessary. This considerably high number of iterations is due to the diagonal Jacobi preconditioner, which is characterized by simplicity and low computational cost, but with the disadvantage of the limited convergence rate.

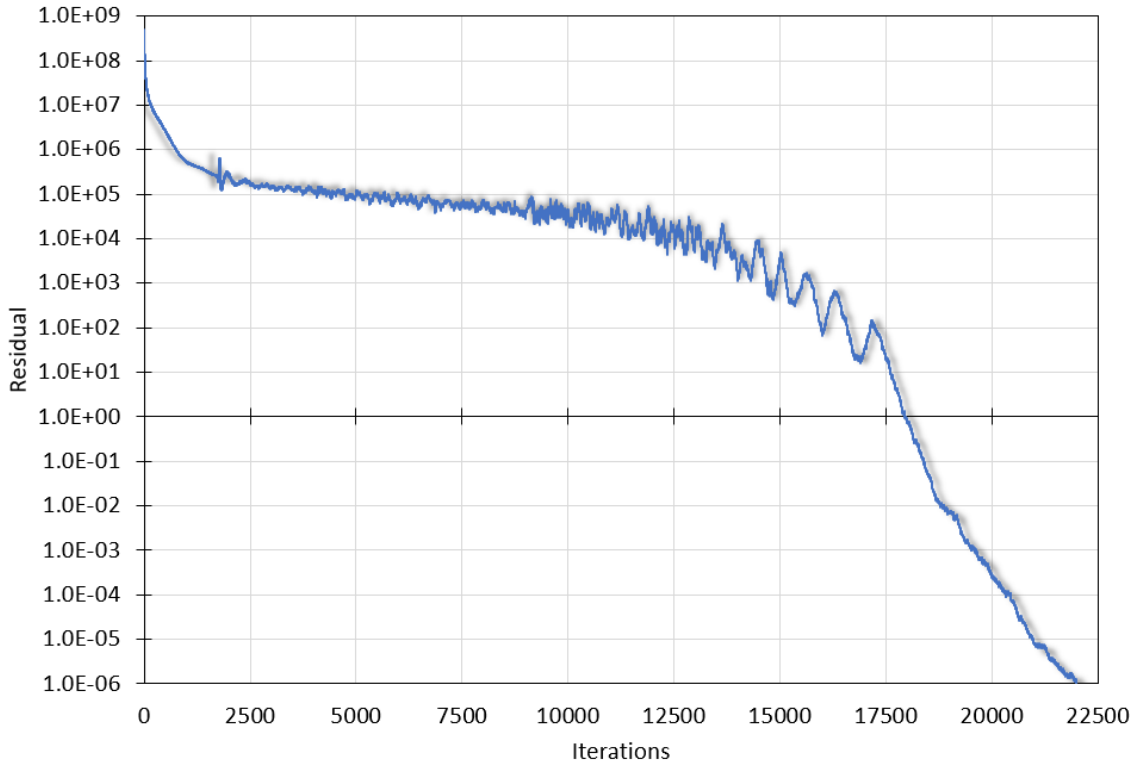

Fig. 10.16 – Residual curve for the diagonal Jacobi preconditioned algorithm for Mesh A (Table 10.6). Source: own authorship.

The low convergence rate of the Jacobi diagonal preconditioner represents a bottleneck to this implementation. That is because, by increasing the number of layers or further refining the mesh, not only does each iteration become more costly to be computed (what is expected and inevitable), but it also increases the total number of iterations to achieve the numerical convergence of the problem. The simulation of Mesh B, for instance, required over 320,000 iterations and took around 9 hours to be completed with 16 threads.

The efficiency of the preconditioner depends on the pattern of the global stiffness matrix, in this case, of the types and combination of the finite macroelements being used. To illustrate this, four different models with approximately 320,000 degrees-of-freedom were simulated and the residual curves are shown in Fig. 10.17. To obtain approximate values of this amount of degrees-of-freedom, only the number of axial divisions was varied (or increased). The first model consists of two layers of tensile armors, meshed with helical beam macroelements. The second consists of the same two layers, but with the addition of bonded contact between them. The third model is a single polymeric sheath, meshed with Fourier cylindrical macroelements. The last and fourth model is exactly the same three-layered simplified flexible pipe from Mesh A (Table 10.6). The cylindrical layer model was the one that converged with the smallest number of iterations. These results show that the efficiency of the diagonal preconditioner is strongly affected by the presence of beam elements in the model.

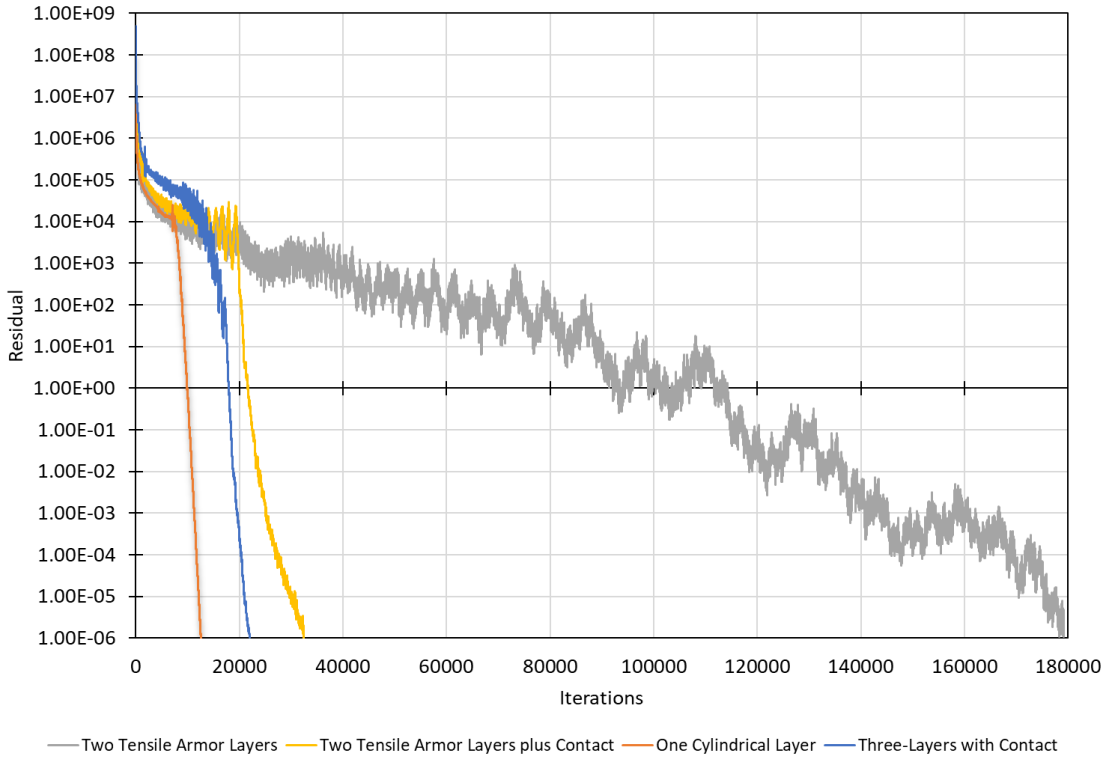

Fig. 10.17 – Comparison of residual curves for different models. Source: own authorship.

Most examples in the literature deal with structural solid elements, including (GULLERUD & DODDS JR, 2001), for instance. One of the examples presented by the authors of the aforementioned work consists of a three-dimensional structure modeled

with 96,120 second-order isoparametric solid elements and 107,436 nodes (322,308 degrees-of-freedom). Adopted a tolerance of 1.0×10^{-4}, it were necessary 1,463 iterations for convergence with the diagonal preconditioner, and 890 iterations with the Hughes-Winget version. When using the diagonal preconditioner, the difference in the number of iterations between beam and solid elements is remarkable, showing that this version of preconditioner is not adequate for models with beam elements.

10.7 Additional Results and Comparison with ANSYS®

In order to compare the efficiency of PipeFEM with a well-established finite element software, the same simplified flexible pipe from Fig. 10.1 was also modeled in ANSYS®, in which the both tensile armor layers were meshed with second-order beam elements, BEAM189. Orientation *keypoints* were necessary to correctly rotate the cross-sections of the beam elements in accordance with the helical pattern, otherwise they would be erroneously twisted. The polymeric sheath was meshed with second-order isoparametric solid elements, SOLID186. The interface between both armor layers was modeled with 3D *line-to-line* contact elements, CONTA176, in crossing condition and with contact radius stipulated as half of the cross-sections heights, which enables great results for beam-to-beam contacts. In the interface between external armor and the external sheath it were employed 3D line-to-surface contact elements, CONTA177. For both interfaces, the contact behavior was selected as "*bonded always*".

For the aforementioned comparison, the flexible pipe models were submitted to a traction case. The pipe was constrained in one of the ends and a traction-displacement of 10 mm was imposed to all the three layers. Before proceeding to the performance comparisons, however, it is important to consider that ANSYS® and PipeFEM have distinct formulations. While ANSYS® uses the conventional finite element method, PipeFEM employs the finite macroelements formulated by PROVASI & MARTINS specifically for the modelling of flexible pipe. Therefore, for a fair comparison between ANSYS® and PipeFEM, it is necessary to ensure that the element meshes are converged in both programs. In this way, convergence analyzes were carried out, always starting with a coarse mesh and refining it incrementally. The convergence is certified by monitoring the variation of the displacements (radial, circumferential and axial directions) along the axial length of a tendon from the internal tensile armor.

It was decided to start the convergence analysis with PipeFEM, in which the element mesh has three control parameters: the number of axial divisions in the tensile armors (the number of axial divisions in the polymeric sheath is always the double); the number of radial divisions in the polymeric sheath; and the Fourier expansion order.

Firstly, the influence of the Fourier expansion order was analyzed. By fixing the number of axial and radial divisions (one hundred and two divisions, respectively), the Fourier order was varied from zero to four. The displacements results, illustrated from Fig. 10.18 to Fig. 10.20, show that order zero is enough for the problem. Despite the non-symmetric geometry of each tendon, both tensile armor layers behave very closely to the axisymmetric for the traction case. This can be explained by the high number of tendons in both tensile layers (that are compatible with real project applications) and by the bonded contact interactions between all layers.

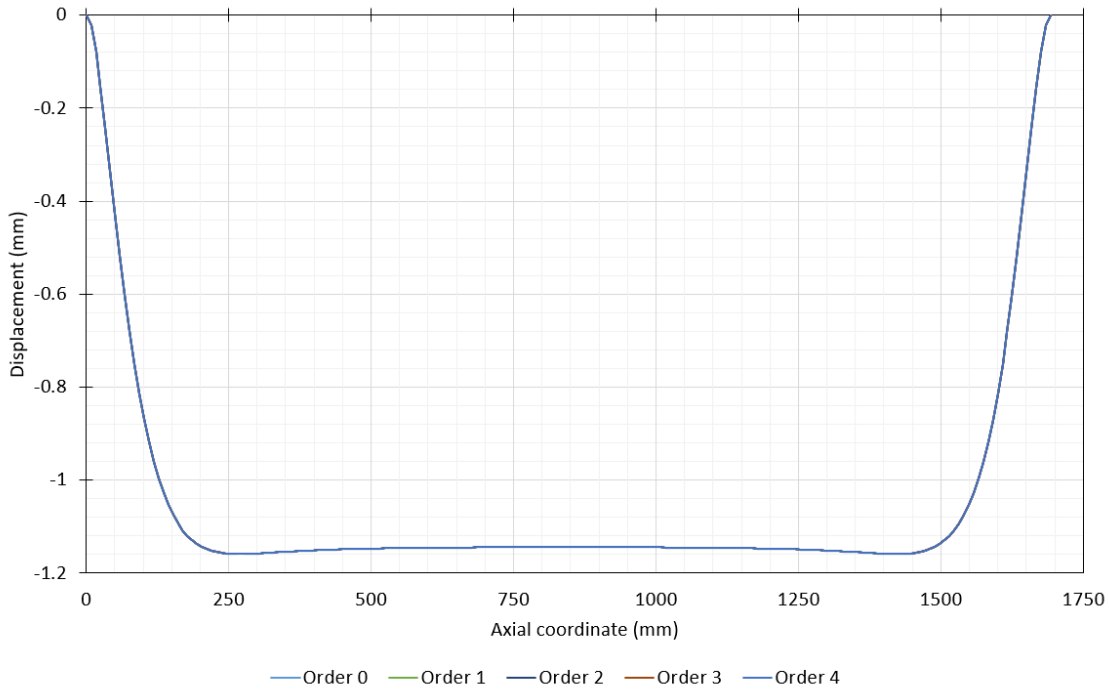

Fig. 10.18 – Convergence in PipeFEM: radial displacements, in mm, along a tendon from the internal tensile layer (Fixed: 100 axial and 2 radial divisions). Source: own authorship.

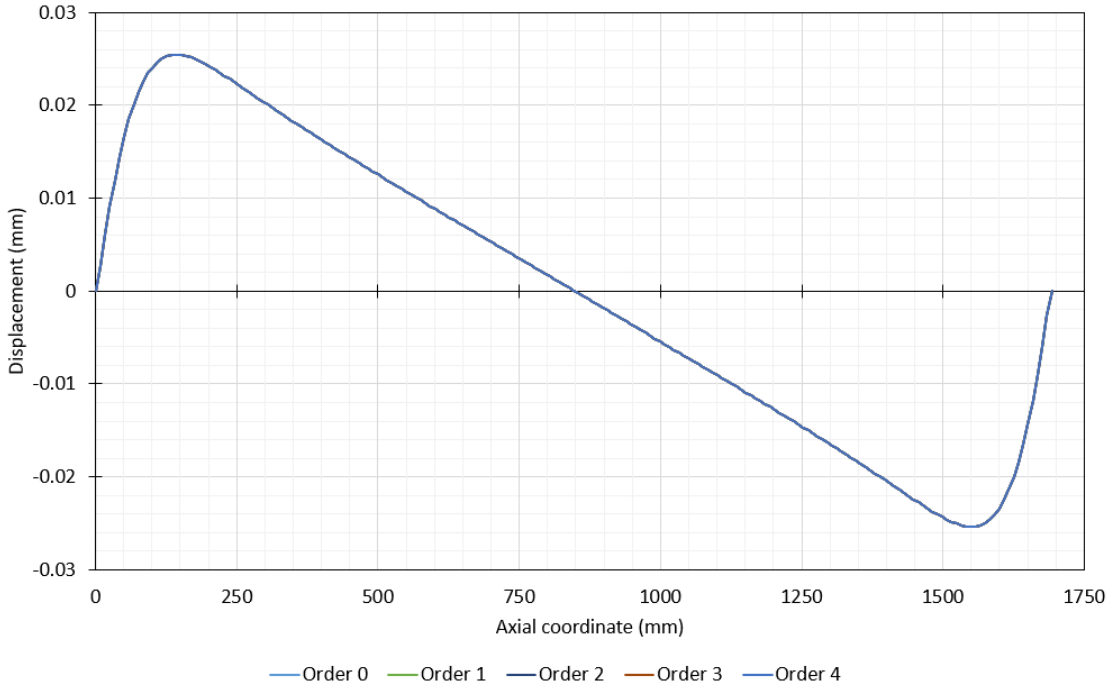

Fig. 10.19 – Convergence in PipeFEM: circumferential displacements, in mm, along a tendon from the internal tensile layer (Fixed: 100 axial and 2 radial divisions). Source: own authorship.

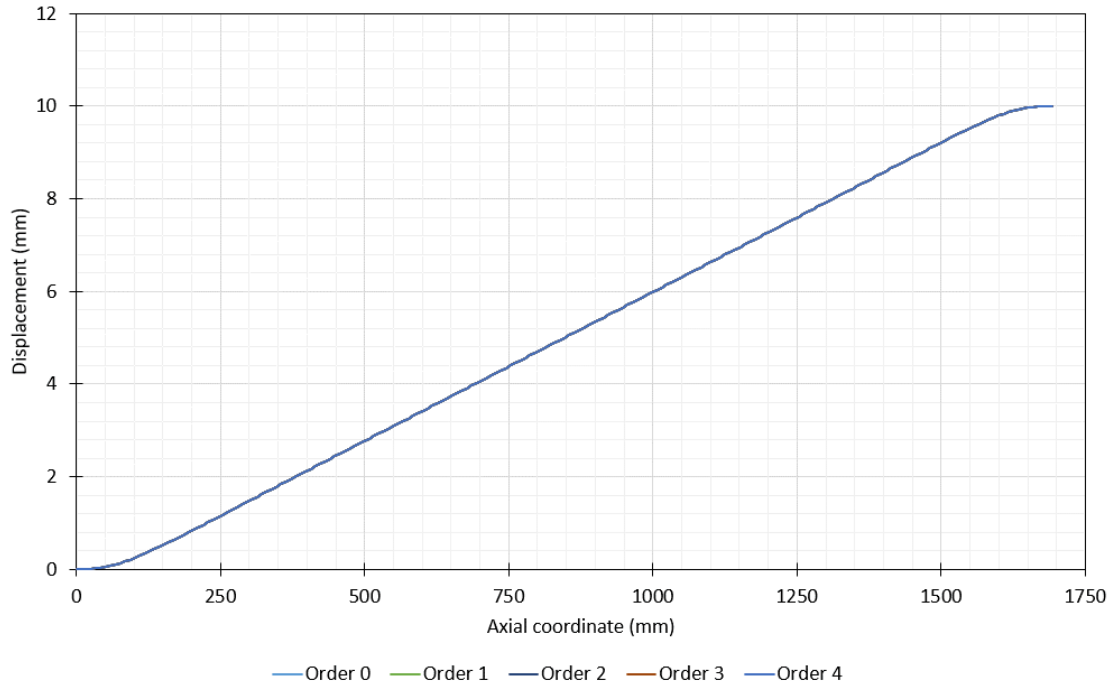

Fig. 10.20 – Convergence in PipeFEM: axial displacements, in mm, along a tendon from the internal tensile layer (Fixed: 100 axial and 2 radial divisions). Source: own authorship.

In sequence, the number of axial divisions were modified, while maintaining the number of radial divisions as 2 and the Fourier order as zero. The results are shown from

Fig. 10.21 to Fig. 10.23. It can be observed that the second case, with 50 divisions, is already converged in all directions.

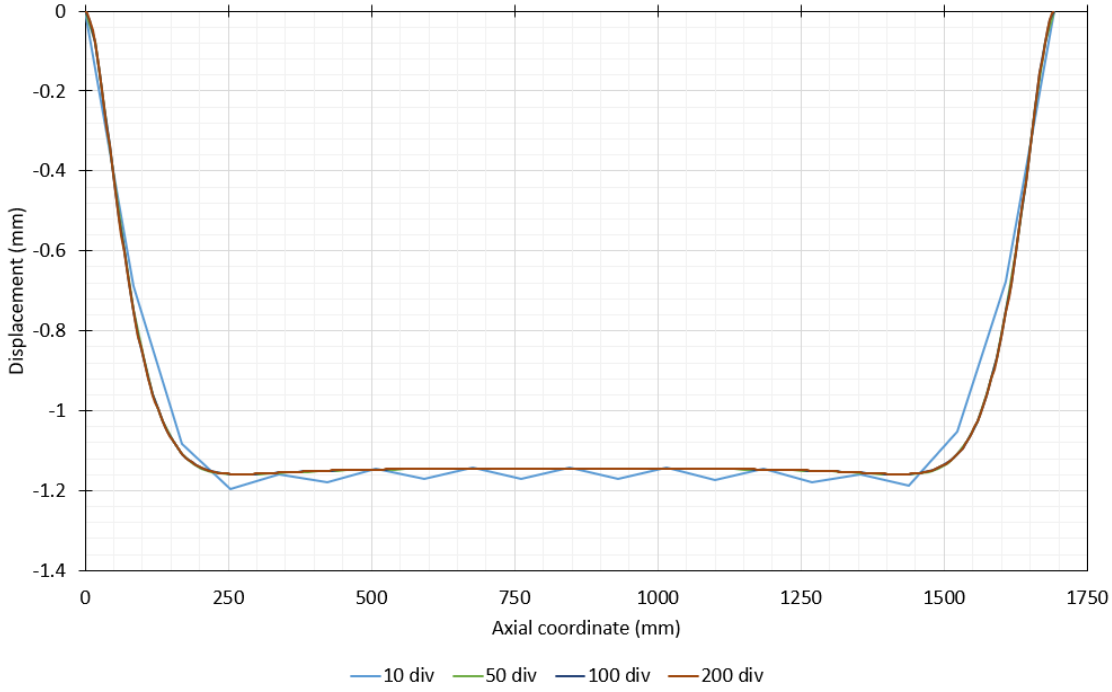

Fig. 10.21 – Convergence in PipeFEM: radial displacements, in mm, along a tendon from the internal tensile layer (Fixed: 2 radial divisions and 0 Order). Source: own authorship.

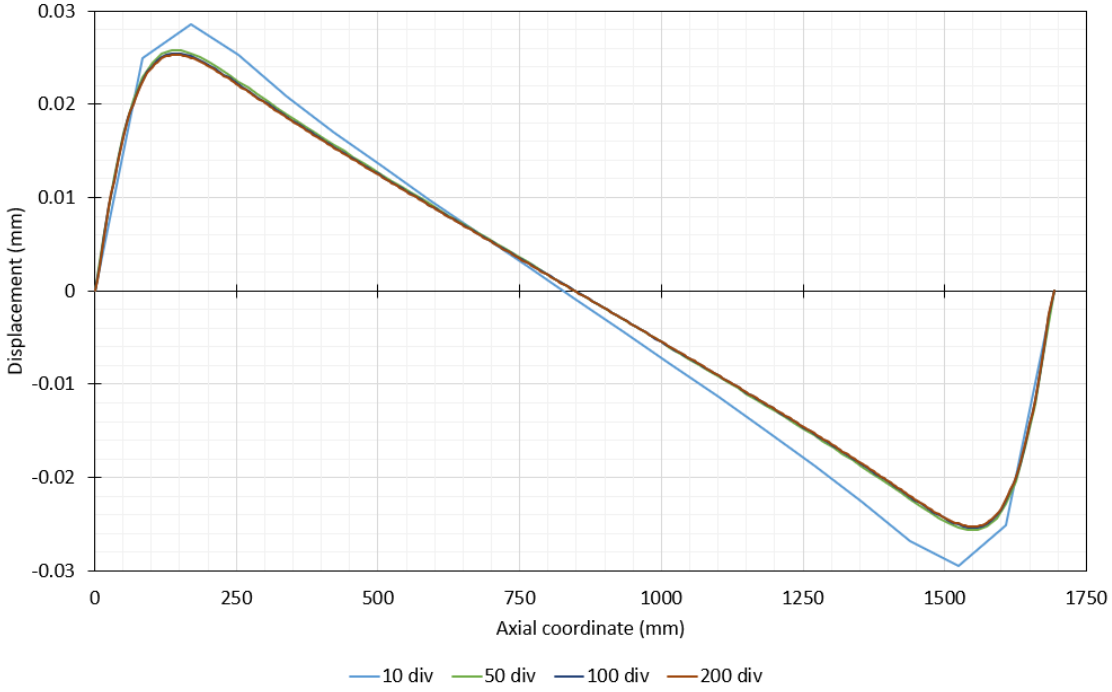

Fig. 10.22 – Convergence in PipeFEM: circumferential displacements, in mm, along a tendon from the internal tensile layer (Fixed: 2 radial divisions and 0 Order). Source: own authorship.

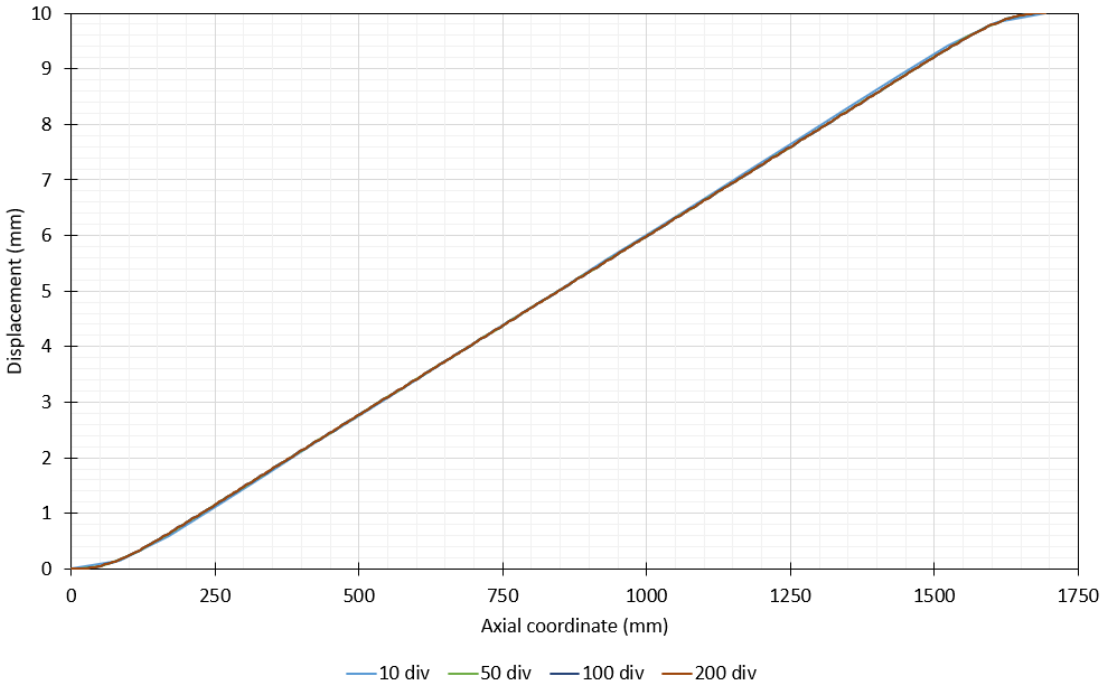

Fig. 10.23 – Convergence in PipeFEM: axial displacements, in mm, along a tendon from the internal tensile layer (Fixed: 2 radial divisions and 0 Order). Source: own authorship.

Lastly, another convergence analysis was also performed for PipeFEM. The number of radial divisions was fixed as one and the Fourier order as zero, while the number of axial divisions was changed. The graphs from Fig. 10.24 to Fig. 10.26 show that the numerical convergence of the model is achieved with 30 axial divisions in the tensile armor layers, 1 radial division in the polymeric sheath and Fourier order 0.

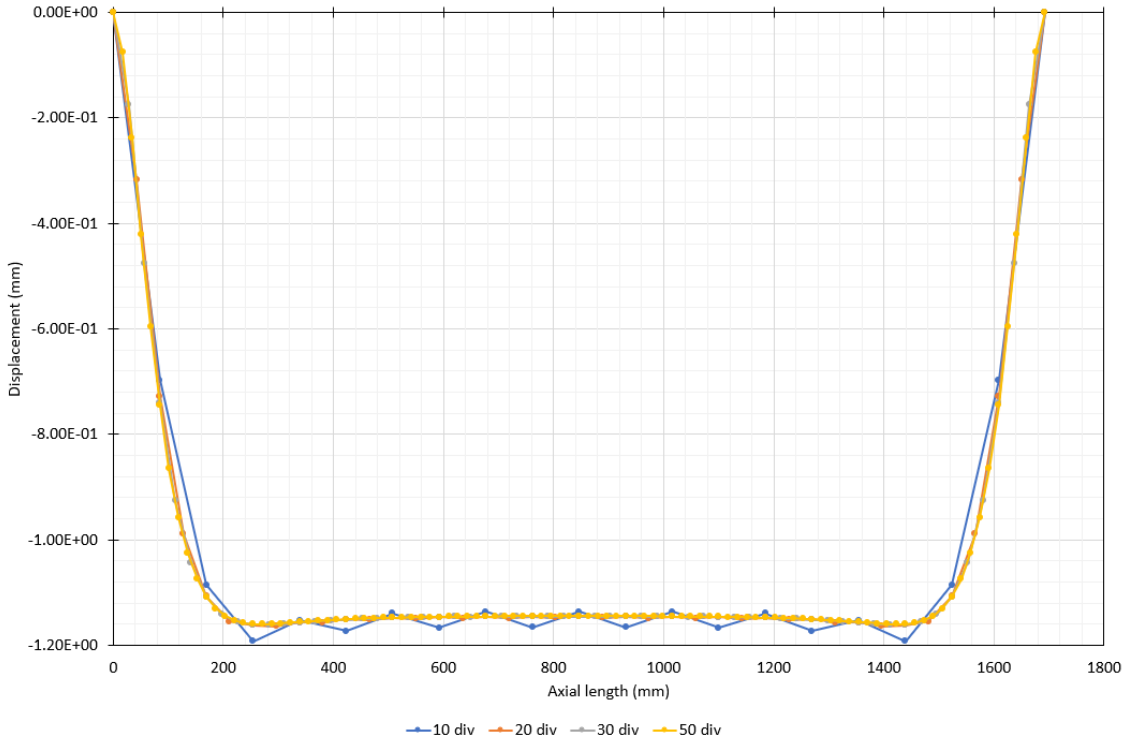

Fig. 10.24 – Convergence in PipeFEM: radial displacements, in mm, along a tendon from the internal tensile layer (Fixed: 1 radial division and 0 Order). Source: own authorship.

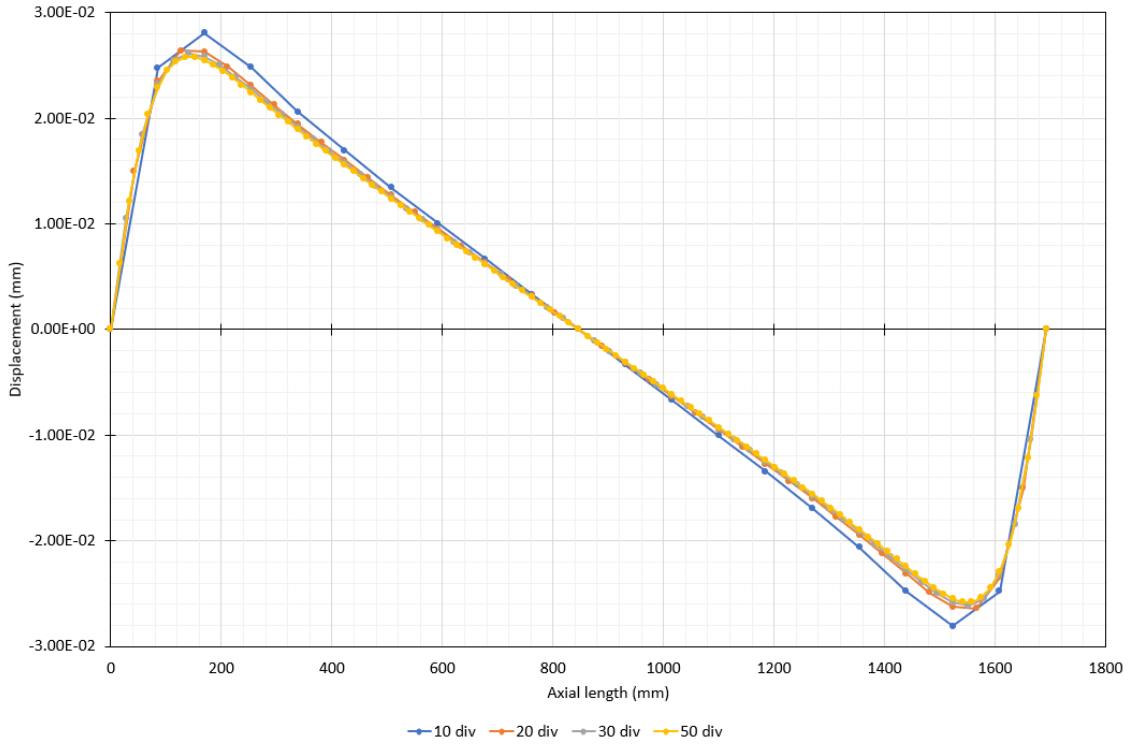

Fig. 10.25 – Convergence in PipeFEM: circumferential displacements, in mm, along a tendon from the internal tensile layer (Fixed: 1 radial division and 0 Order). Source: own authorship.

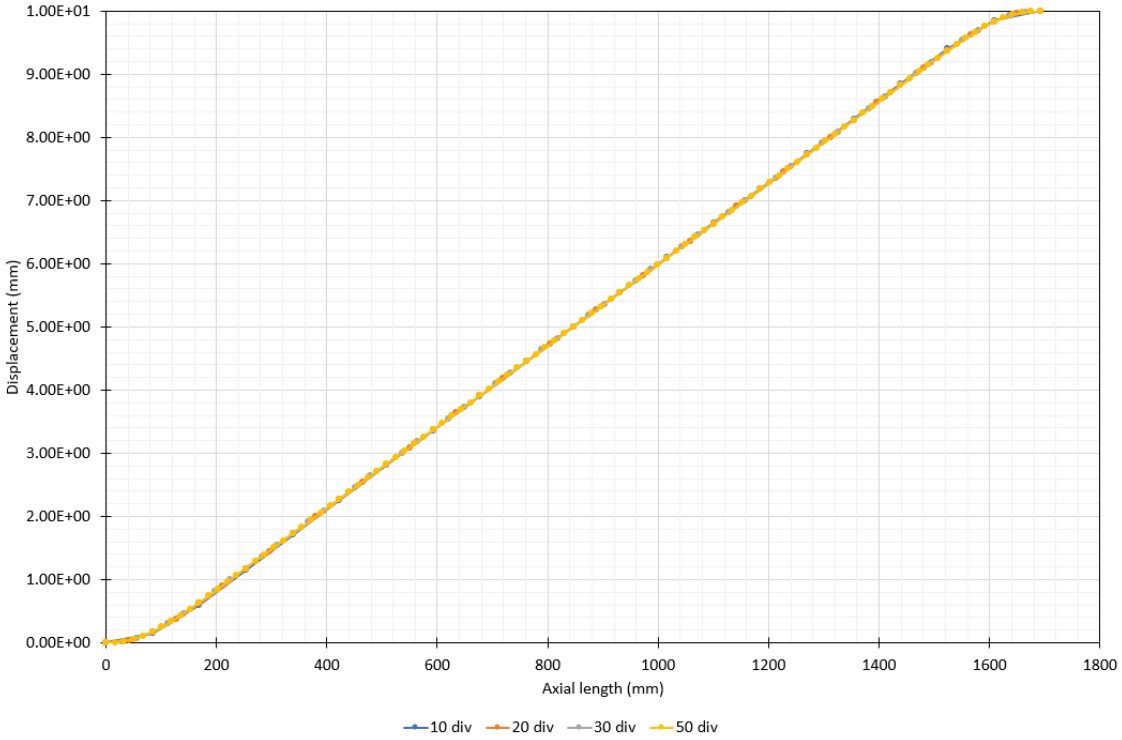

Fig. 10.26 – Convergence in PipeFEM: axial displacements, in mm, along a tendon from the internal tensile layer (Fixed: 1 radial division and 0 Order). Source: own authorship.

The statistics of execution time and memory consumption for several combinations of mesh parameters are presented in Table 10.12. The previous charts showed that convergence was achieved in PipeFEM with 50 axial divisions, 1 radial division and Order 0. With these mesh parameters, the flexible pipe model was built almost immediately (0.102 seconds) and the numerical solution took 24.27 seconds, with a very low RAM memory consumption of 61.8 MB. When considering a more conservative mesh, containing 50 axial divisions, 2 radial divisions and Order 4, the model still was built in a fraction of seconds, while the solver took approximately 1 minute and consumed 113.9 MB.

Table 10.12 – Execution time and memory consumption in PipeFEM.

Mesh	*Mesh parameters*			*DOFs*	*PCG iterations*	*Execution time*			*RAM memory*
	Axial	*Radial*	*Order*			*Model*	*PCG*	*Solver*	
M1	30	1	0	21,354	9,900	0.075s	12.98s	13.25s	37.6 MB
M2	50	1	0	72,720	12,800	0.102s	23.89s	24.27s	61.8 MB
M3	50	2	0	73,023	12,800	0.104s	24.08s	24.47s	62.4 MB
M4	50	2	4	80,295	13,800	0.107s	59.46s	59.94s	113.9 MB
M5	100	1	0	144,720	15,800	0.167s	50.93s	51.63s	114.6 MB
M6	100	2	0	145,323	15,800	0.167s	52.06s	52.75s	114.9 MB
M7	100	2	4	159,795	16,600	0.168s	129.36s	130.19s	215.9 MB
M8	200	2	0	289,923	21,600	0,290s	120.31s	121.59s	219.8 MB
M9	200	2	4	318,795	22,800	0.297s	320.06s	326.66s	420.4 MB

Fig. 10.27 illustrates a parametric analysis of the memory consumption from PipeFEM in function of the number of the degrees-of-freedom, for the orders 0, 2 and 4, maintaining constant the number of radial divisions as two and varying the number of axial divisions. These curves show that, for a fixed Fourier order, PipeFEM presents linear growth in memory consumption, which is in complete agreement with the EBE method and demonstrates the quality of the implementation in this aspect. The change in the Fourier expansion order has great impact in memory consumption, since it increases the dimensions of a large portion of the element stiffness matrices of the model.

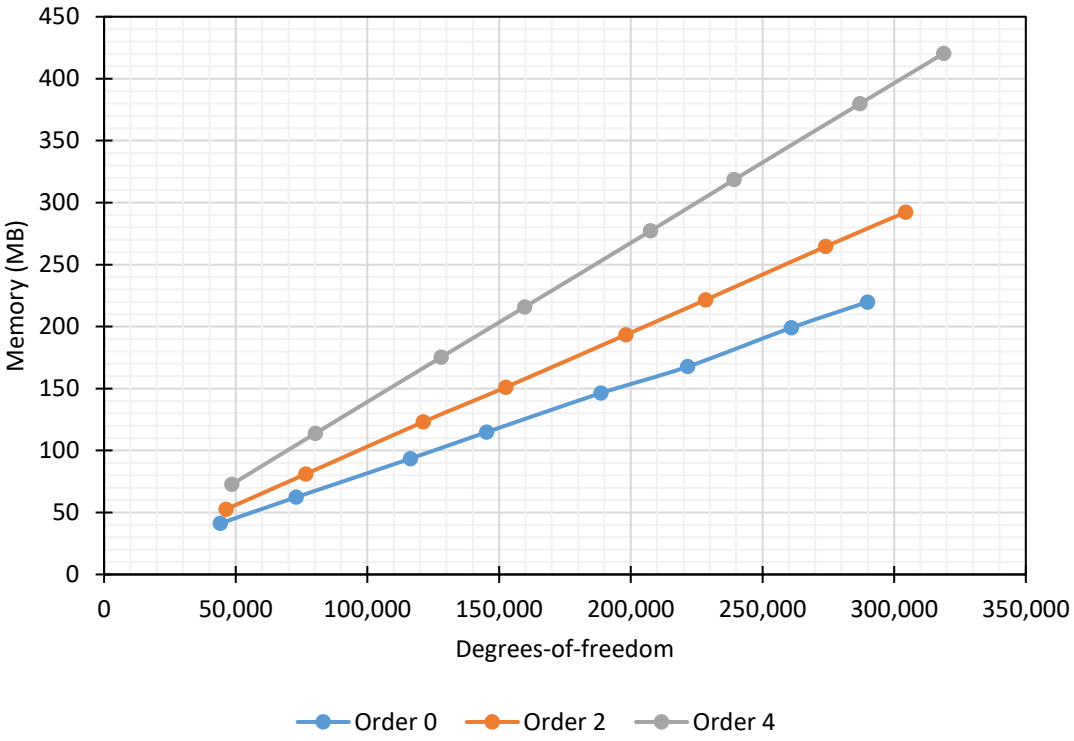

Fig. 10.27 – Memory consumption in function of the number of degrees-of-freedom in PipeFEM (Fixed: 2 radial divisions). Source: own authorship.

The parametric analysis of the solver simulation time in function of the number of degrees-of-freedom is illustrated in Fig. 10.28. The growth rate of the PCG simulation time is a little higher than linear. This is because, when increasing model size, not only the iterations become more costly, but also a higher number of them is required to the numerical convergence of the algorithm (with the diagonal preconditioner). By dividing these execution times by the respective total number of iterations, the graph from Fig. 10.29 was obtained, which consists of the timer per iteration in function of the number of degrees-of-freedom from PipeFEM. These results show that the cost per iteration increases almost linearly with the number of degrees-of-freedom, indicating the effectiveness of the implementation.

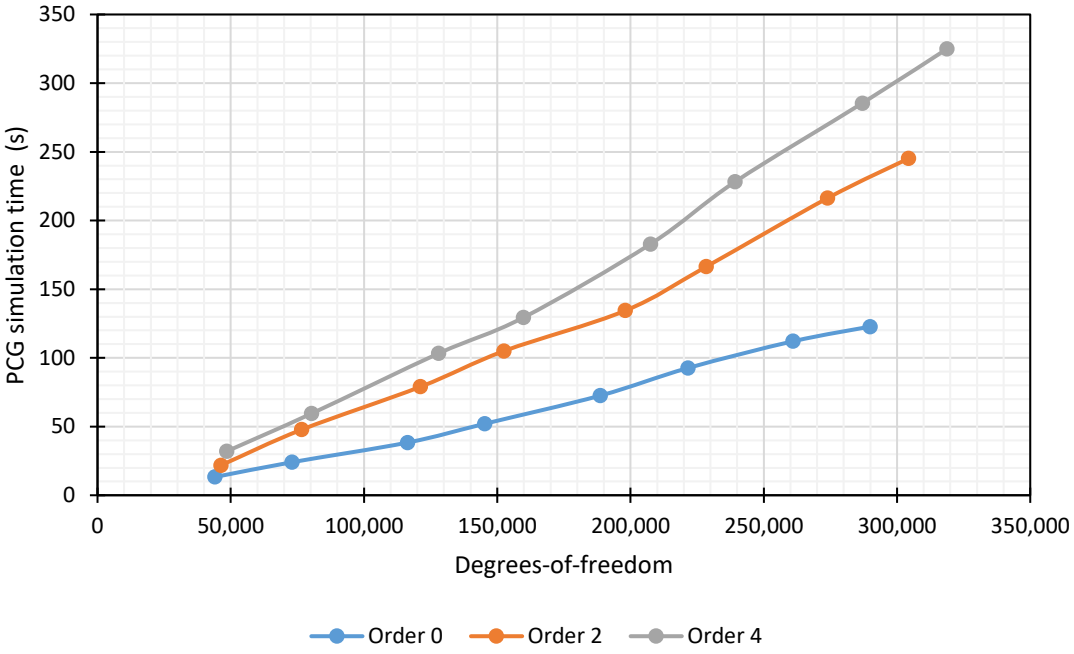

Fig. 10.28 – PCG simulation time in function of the number of degrees-of-freedom in PipeFEM. (Fixed: 2 radial divisions). Source: own authorship.

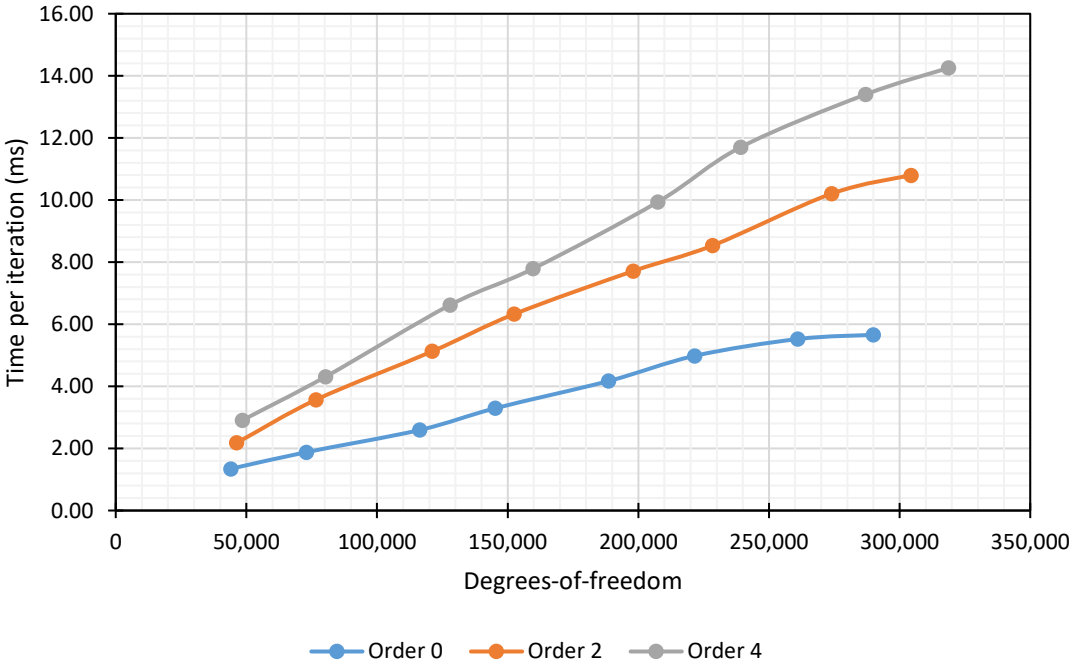

Fig. 10.29 – Time per iteration of the PCG algorithm in function of the number of degrees-of-freedom in PipeFEM (Fixed: 2 radial divisions). Source: own authorship.

The convergence analysis proceeds then to the ANSYS® software. In this mesh has four control parameters: the number of axial divisions from the tensile armors; the number

of radial, circumferential and axial divisions from the polymeric sheath. Nine combinations of mesh parameters were simulated in ANSYS® and they are listed in Table 10.13.

Table 10.13 – Element meshes tested in ANSYS® for the convergence analysis.

Mesh	***Tensile Armors***	***Polymeric Sheath***			***D.O.F.s***
	Axial	***Radial***	***Circumferential***	***Axial***	
M1	20	1	12	10	37,206
M2	20	1	12	20	47,286
M3	40	1	12	20	75,846
M4	40	1	12	40	96,006
M5	40	1	20	40	122,406
M6	40	1	40	40	188,406
M7	80	1	40	80	379,926
M8	80	2	40	80	532,086
M9	100	2	40	100	666,246

Fig. 10.30 illustrates the convergence test in ANSYS® of the radial displacements, in millimeters, along the axial length of a tendon from the inner tensile armor. Analogously, Fig. 10.31 and Fig. 10.32 illustrate the convergence tests for the circumferential and axial directions. These results show that the circumferential direction is the most sensitive one in relation to convergence, which only was completely obtained from the seventh mesh.

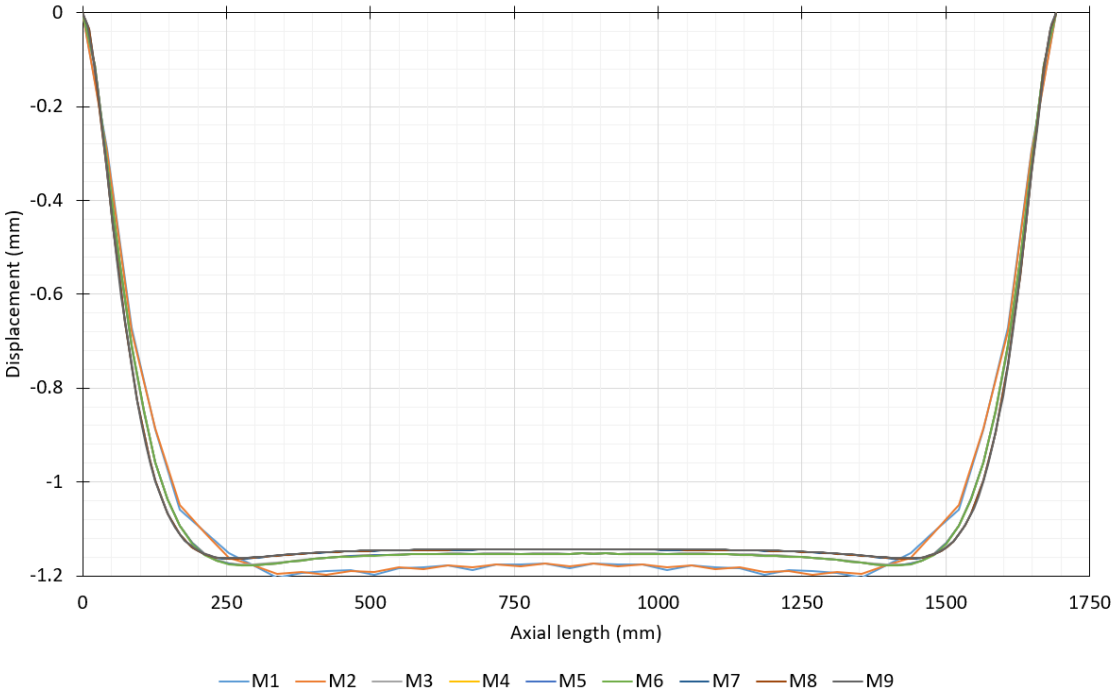

Fig. 10.30 – Convergence in ANSYS®: radial displacements, in mm, along a tendon from the internal tensile layer. Source: own authorship.

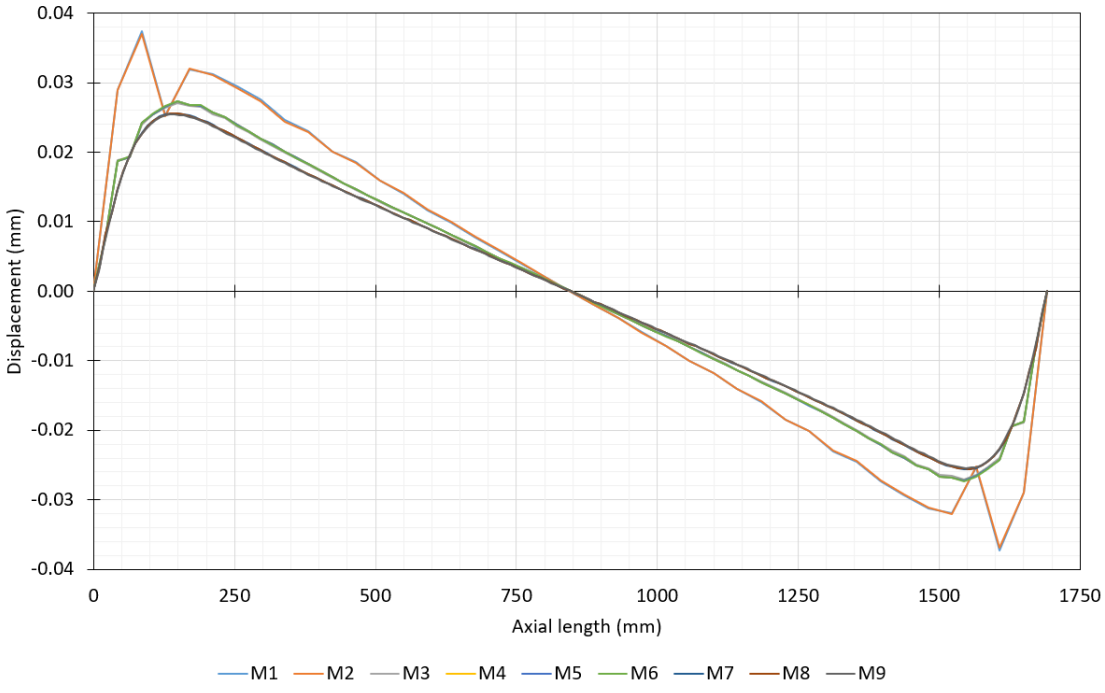

Fig. 10.31 – Convergence in ANSYS®: circumferential displacements, in mm, along a tendon from the internal tensile layer. Source: own authorship.

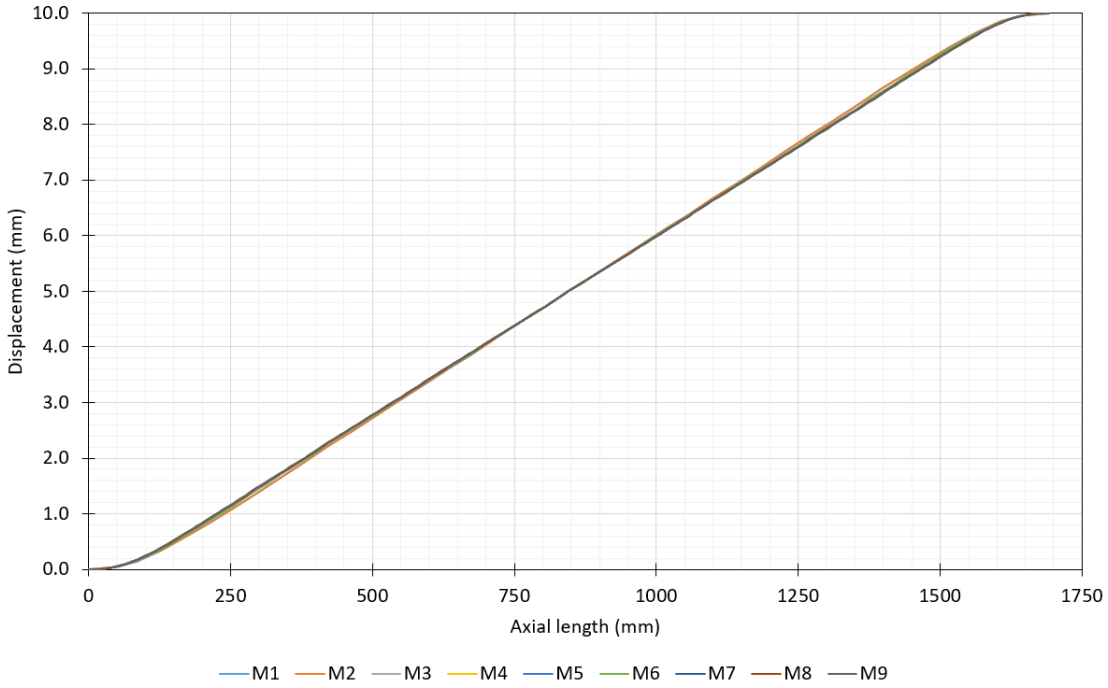

Fig. 10.32 – Convergence in ANSYS®: axial displacements, in mm, along a tendon from the internal tensile layer. Source: own authorship.

The construction of the "*M7*" mesh in ANSYS® took 12 minutes and 55 seconds. This high processing time is due the fact that the contact between the two tensile armor layers had to be made helix-by-helix, what created 3,528 different contact pairs (the numerical combination of the 56 tendons of internal tensile armor and the 63 tendons of the external). A simpler unique contact pair methodology was also tested, with 56 lines in the master and 63 lines the contact regions, but this approach has lost robustness and stopped converging for some types of external applied loads, such as external pressure. In addition to this, ANSYS® took 33 minutes and 18 seconds to numerically solve the "*M7*" model, and consumed approximately 12.5 GB of RAM memory (5.7 GB is relative to the graphic interface). Table 10.14 summarizes these numbers and also the consumptions from the "*M8*" mesh.

Table 10.14 – Execution time and memory consumption in ANSYS®.

Mesh	***Model construction***	***Numerical solution***	***Memory consumption (graphic interface)***
M7	12min 55s	33min 18s	12.5 GB (5.7 GB)
M8	25min 56s	41min 33s	18.5 GB (7.8 GB)

Lastly, the results of the converged mesh from PipeFEM ("*M2*" from Table 10.12) are compared with the results of the converged mesh from ANSYS® ("*M7*" from Table 10.13). The radial, circumferential and axial displacements of a tendon in the internal armor as illustrated in Fig. 10.33, Fig. 10.34 and Fig. 10.35, respectively. There is no notable differences in the results, attesting that the EBE-PCG algorithm was correctly implemented in PipeFEM. It is important to consider, however, the performance differences between both programs. Regarding the construction of the model, it was accomplished in 0.102 seconds in PipeFEM, against 12 minutes and 55 seconds in ANSYS®. Significant performance differences were also registered in the total simulation time: 24.27 seconds in PipeFEM, versus 33 minutes and 18 seconds in ANSYS®. In addition, memory consumption was much lower in the developed program (approximately 12.5 GB in ANSYS® versus 61.8 MB in PipeFEM).

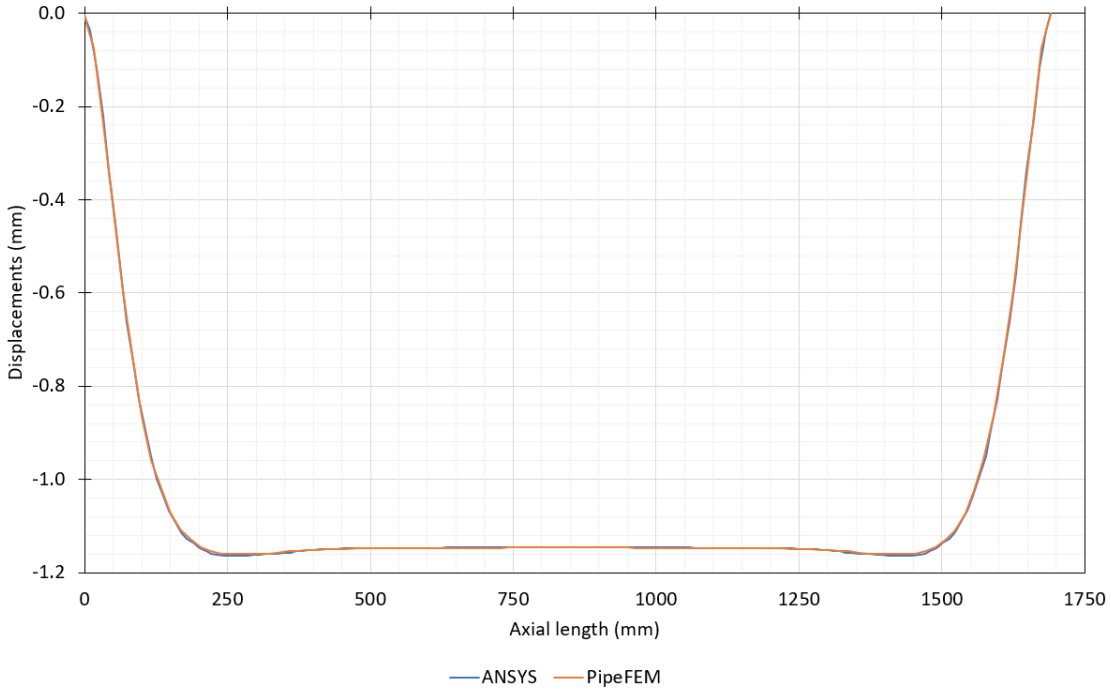

Fig. 10.33 – Radial displacement of a tendon in the internal armor, traction loading. Source: own authorship.

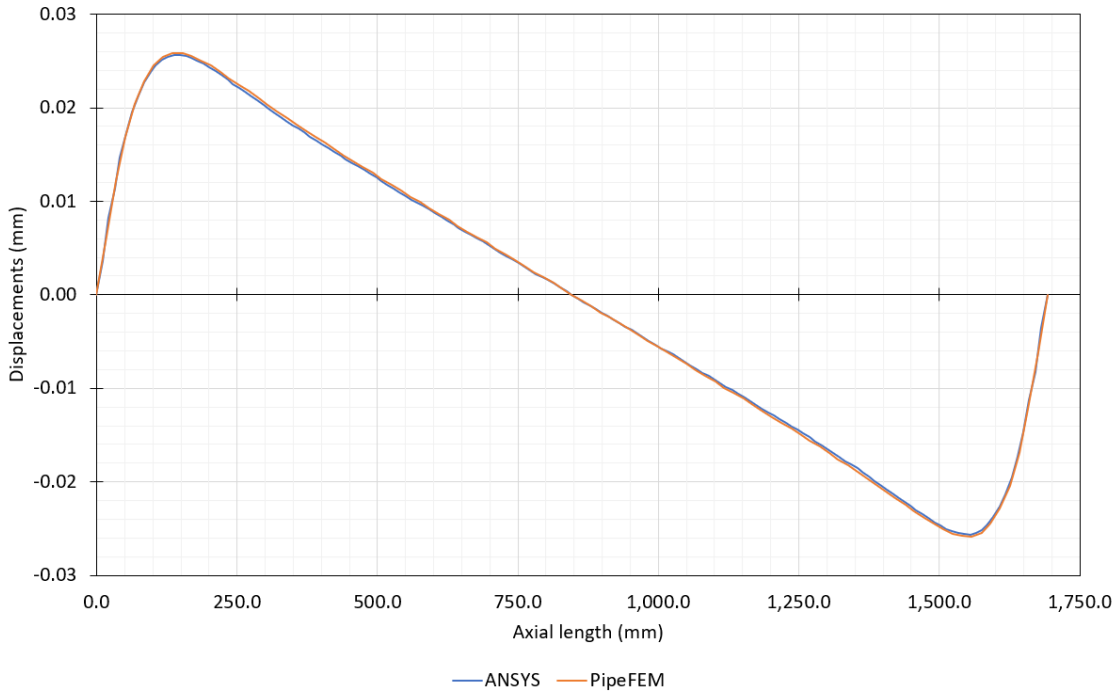

Fig. 10.34 – Circumferential displacement of a tendon in the internal armor, traction loading. Source: own authorship.

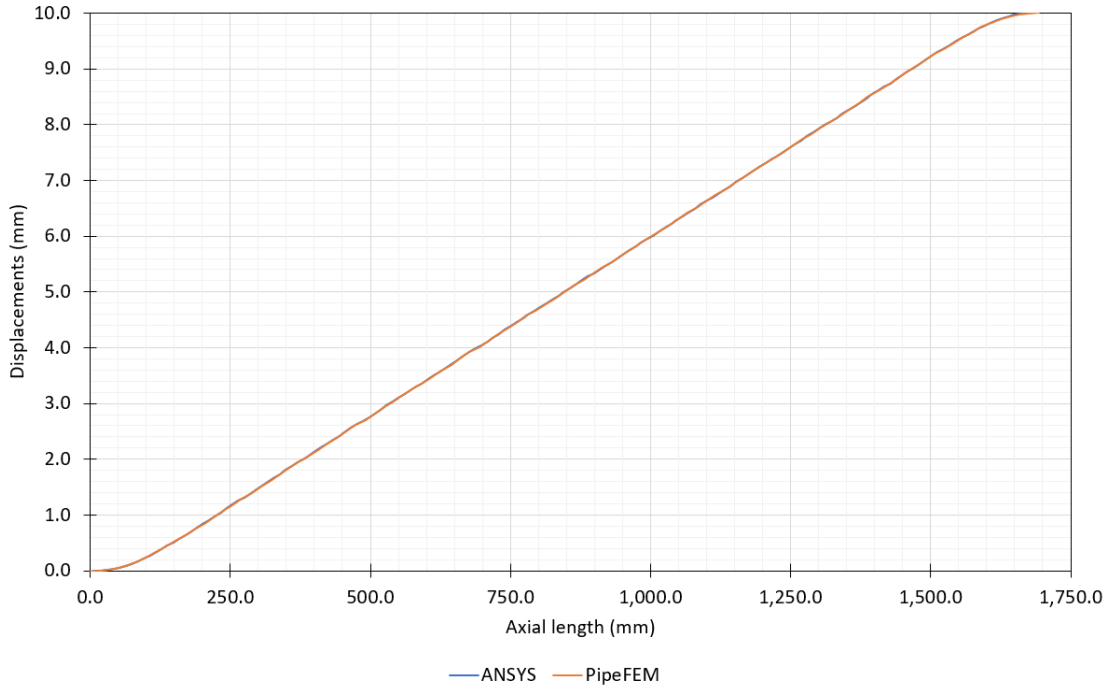

Fig. 10.35 – Axial displacement of a tendon in the internal armor, traction loading. Source: own authorship.

To solve the same simplified flexible pipe from Fig. 10.1, PipeFEM was 82 times faster than ANSYS®, a reduction of almost two orders of magnitude, with a much lower memory consumption (only 0.89% of the memory required by ANSYS®).

In comparison to the dense version of MacroFEM, a massive reduction in memory consumption was achieved, in more than three orders of magnitude. MacroFEM required more than 60 GB of RAM memory to solve the pipe from Fig. 1.19, while it can be done in PipeFEM with less than 60 MB. In this case, gains were also obtained in simulation time: the dense version of MacroFEM employs the *MKL* solver from the library "*Math.NET Numerics*", which took approximately 15 minutes to solve the problem, while PipeFEM demanded only 34 seconds to solve the same problem.

Therefore, it can be concluded that, despite the limitations of the rate of convergence of the diagonal preconditioner employed in the EBE-PCG algorithm, the implementation is very efficient in computational terms. If necessary, additional performance gains could be achieved with the implementation and development of more complex preconditioners in the future.

11 Conclusions

This work was motivated by memory and processing limitations on finite element structural analysis of flexible pipes for offshore applications. The Element-by-Element method, characterized by the global stiffness matrix elimination, was chosen for its potential in memory reduction and processing capabilities, given its scalability and ease of parallelization. After an extensive literature review on numerical methods regarding the EBE method, it was chosen the EBE Diagonal Preconditioned Conjugate Gradient Method (EBE-PCG) algorithm.

Aiming higher computational performance, the finite macroelements formulated by (PROVASI, 2013) were converted to the C++ language, parallelized and implemented in a new analysis tool, named as PipeFEM, entirely written in C++ and that explores parallelism.

A fully indexed geometry and mesh data structure was developed, with the same facilities of the item selection features found in the multi-purpose finite element package ANSYS® (that are extremely useful for contact and load applications), but that also takes advantage of the computational benefits of direct indexing and facilitates the implementation and manipulation of three-dimensional finite elements.

Regarding the numerical solution of the problem, the EBE-PCG algorithm was implemented and parallelized with OpenMP. The scalability of the PCG algorithm is directly influenced by the efficiency of the matrix-vector product, an operation that, in the element-by-element method, is computed in a local basis with the blocks that comprise the model, and that requires synchronization techniques when performed in parallel. Four different synchronization strategies were developed, being the one based on geometric- and mesh- based mappings the most efficient of them. Numerical experiments showed a reduction of almost 92% in the EBE-PCG solution time of the parallelized version in comparison to the sequential one.

In order to compare the efficiency of PipeFEM with the well-established finite element package ANSYS®, a simplified flexible pipe was modeled in both software, containing two tensile armor layers and one polymeric sheath. Convergence tests were carried out for a valid comparison. The displacement results from PipeFEM are in great agreement with ANSYS® and, thus, reliable. Regarding the construction of the flexible

pipe model, ANSYS® spent 12 minutes and 55 seconds, while PipeFEM took 0.102 seconds, a reduction of almost four orders of magnitude. To numerically solve the problem, ANSYS® spent 33 minutes and 18 seconds, against 24.27 seconds in PipeFEM, a difference of almost 82 times. In addition to this, PipeFEM presented a much lower memory consumption, 61.8MB against 6.8GB in ANSYS® (already discount the graphic interface).

In comparison to the dense version of MacroFEM, a massive reduction in memory consumption was achieved, in more than three orders of magnitude. MacroFEM required more than 60 GB of RAM memory to solve the pipe from Fig. 1.19, while it can be done in PipeFEM with less than 60 MB. In this case, gains were also obtained in simulation time: the dense version of MacroFEM employs the *MKL* solver from the library "*Math.NET Numerics*", which took approximately 15 minutes to solve the problem, while PipeFEM demanded only 34 seconds to solve the same problem.

Despite the limitations of the rate of convergence of the diagonal preconditioner employed in the EBE-PCG algorithm, the implementation is very efficient in computational terms. Therefore, with the gains obtained in processing time and memory consumption, it can be concluded that the objectives of this work were fulfilled.

As future activities, more complex preconditioners can be implemented or developed, which would enable additional performance gains.

12 References

ADELI, H., & KUMAR, S. (1995). Distributed Finite-Element Analysis on Network of Workstations — Algorithms. *Journal of Structural Engineering, 10,* 1448-1455. doi:10.1061/(ASCE)0733-9445(1995)121:10(1448)

API RP 17B . (2002). *API RP 17B - Recommended Practice for Flexible Pipe* (3rd ed.). American Petroleum Institute.

ASEN 6367 - Chapter 11. (2013). *Advanced Finite Element Methods (ASEN 6367) - Spring 2013.* Retrieved 04 26, 2016, from Department of Aerospace Engineering Sciences - University of Colorado at Boulder: http://www.colorado.edu/engineering/CAS/courses.d/AFEM.d/AFEM.Ch11.d/AFEM.Ch11.pdf

BARTELL. (2016). *Carcass Machines*. Retrieved 06 03, 2016, from Martell Machinery: http://www.bartellmachinery.com/carcass-machines?page_id=55

BARTELL. (2016). *Pad Style Taping Heads*. Retrieved 06 03, 2016, from Bartell Machinery: http://www.bartellmachinery.com/taping-heads?page_id=53

BRAGA, M., & KALLEF, P. (2004). Flexible Pipe Sensitivity to Birdcaging and Armor Wire Lateral Buckling. *23rd International Conference on Offshore Mechanics and Arctic Engineering (OMAE 2004)*, 139-146. doi:10.1115/OMAE2004-51090

CAREY, G., & JIANG, B. (1986). Element-by-element linear and nonlinear solution schemes. *Communications in Applied Numerical Methods*, 145-153. doi:10.1002/cnm.1630020205

CONTINENTAL. (2014). *High Performance Flexible Lines.* ContiTech Oil & Marine. Retrieved from http://www.taurus-emerge.com/pages/brochures/downloads/Taurus_Oil_Marine_Hoses_20141201_en.pdf

COOK, R., MALKUS, D., PLESHA, M., & WITT, R. (2002). *Concepts and Applications of Finite Element Analysis* (4rd ed ed.). New York, USA: Wiley,.

COUTINHO, A., ALVES, J., LANDAU, L., EBECKEN, N., & TROINA, L. (1991). Comparison of lanczos and conjugate gradients for the element-by-element solution of finite element equations on the ibm 3090 vector computer. *Computers & Structures, 39*(1-2), 47-55. doi:10.1016/0045-7949(91)90071-S

COUTINHO, A., ALVES, J., LANDAU, L., LIMA, E., & EBECKEN, N. (1987). On the application of an element-by-element lanczos solver to large offshore structural engineering problems. *Computers & Structures*, 27-37. doi:10.1016/0045-7949(87)90179-9

GULLERUD, A., & DODDS JR, R. (2001). MPI-based implementation of a PCG solver using an EBE architecture and preconditioner for implicit, 3-D finite element analysis. *Computers and Structures, 79(5)*, 553-575. doi:http://doi.org/10.1016/S0045-7949(00)00153-X

HUGHES, J., LEVIT, I., & WINGET, J. (1983-B). An element-by-element solution algorithm for problems of structural and solid mechanics. *Computer Methods in Applied Mechanics and Engineering, 36(2)*, 241–254. doi:http://doi.org/10.1016/0045-7825(83)90115-9

HUGHES, J., LEVIT, M., & WINGET, J. (1983-A). Element-by-Element Implicit Algorithms for Heat Conduction. *Journal of Engineering Mechanics, 109(2)*, 576-585.

HUGHES, T., & FERENCZ, R. (1987). Large-scale vectorized implicit calculations in solid mechanics on a Cray X-MP/48 utilizing EBE preconditioned conjugate gradients. *Computer Methods in Applied Mechanics and Engineering*, 215-248. doi:10.1016/0045-7825(87)90005-3

HUGHES, T., & FERENCZ, R. (1988). Fully vectorized EBE preconditioners for nonlinear solid mechanics: Applications to large-scale three-dimensional continuum, shell and contact/impact problems. *in: R. Glowinski et al., eds., Domain Decomposition Methods for Partial Differential Equations* , 261-280.

HUISMAN. (2008). *Pipe Tensioners*. Retrieved 06 03, 2016, from Huisman Equipment: http://www.huismanequipment.com/en/products/pipelay/pipelay_components/pipe_tensioners

INTEL. (2018, 02 19). *Intel® Turbo Boost Technology 2.0*. Retrieved from Higher Performance When You Need It Most: https://www.intel.com/content/www/us/en/architecture-and-technology/turbo-boost/turbo-boost-technology.html

KING, R., & SONNAD, V. (1987). Implementation of an element-by-element solution algorithm for the finite element method on a coarse-grained parallel computer. *Computer Methods in Applied Mechanics and Engineering*, 47-59. doi:10.1016/0045-7825(87)90182-4

KISS, I., BADICS, Z., GYIMOTHY, S., & PAVO, J. (2012). High locality and increased intra-node parallelism for solving finite element models on GPUs by novel element-by-element implementation. *2012 IEEE Conference on High Performance Extreme Computing (HPEC)* (pp. 1 - 5). Waltham, MA: IEEE. doi:10.1109/HPEC.2012.6408659

KISS, I., GYIMOTHY, S., BADICS, Z., & PAVO, J. (2012). Parallel Realization of the Element-by-Element FEM Technique by CUDA. *IEEE Transactions on Magnetics, 48*(2), 507 - 510. doi:10.1109/TMAG.2011.2175905

LEVIT, I. (1987). Element by element solvers of order N. *Computers & Structures, 27*(3), 357-360. doi:10.1016/0045-7949(87)90058-7

LIU, Y., ZHOU, W., & YANG, Q. (2007). A distributed memory parallel element-by-element scheme based on Jacobi-conditioned conjugate gradient for 3D finite element analysis. *Finite Elements in Analysis and Design, 43*, 494-503. doi:10.1016/j.finel.2006.12.007

MALI. (2016). *MALI Ges.m.b.H.* Retrieved 06 03, 2016, from Machines for the Cable and Wire Industries - Flat Wire: http://www.mali.at/eng/proj_02.htm

MARTÍNEZ-FRUTOS, J., & HERRERO-PÉREZ, D. (2015). Efficient matrix-free GPU implementation of Fixed Grid Finite Element Analysis. *Finite Elements in Analysis and Design, 104*, 61-71. doi:10.1016/j.finel.2015.06.005

MARTÍNEZ-FRUTOS, J., MARTÍNEZ-CASTEJÓN, P., & HERRERO-PÉREZ, D. (2015). Fine-grained GPU implementation of Assembly-Free Iterative Solver for Finite Element Problems. *Computers & Structures, 157*, 9-18. doi:10.1016/j.compstruc.2015.05.010

MUREN, J. (2007). Failure modes, inspection, testing and monitoring. *PSA - NORWAY - Flexible Pipes*.

NOUR-OMID, B., PARLETT, B., & RAEFSKY, A. (1987). Comparison of Lanczos with Conjugate Gradient Using Element Preconditioning. *Proceedings of the 1st International Conference on Domain Decomposition Methods.* Paris, France.

PDL GROUP. (2015, 09 11). *PDL Global Dynamic Analysis: FPSO and Steep WAve Risers*. Retrieved 06 05, 2016, from https://www.youtube.com/watch?v=irJBimOilNg

PROVASI, R. (2013). *Contribuição ao Projeto de Cabos Umbilicais e Tubos Flexíveis: Ferramentas de CAD e Modelo de Macro Elementos.* Tese de Doutorado, Escola Politécnica da Universidade de São Paulo.

PROVASI, R., & MARTINS, C. (2014). A Three-Dimensional Curved Beam Element for Helical Components Modeling. *Journal of Offshore Mechanics and Arctic Engineering, 136*(4). doi:10.1115/1.4027956

PROVASI, R., & MARTINS, C. A. (2013-a). A rigid connection for macro-elements with different node displacement natures. *International Offshore and Polar Engineering Anchorage, International Society of Offshore and Polar Engineers (ISOPE).* Alaska, USA.

PROVASI, R., & MARTINS, C. A. (2013-b). A Contact Element for Macro-Elements with Different Node Displacement Natures. *International Offshore and Polar Engineering Anchorage, International Society of Offshore and Polar Engineers (ISOPE).* Alaska, USA.

PROVASI, R., & MARTINS, C. A. (2013-c). A Finite Macro-Element for Orthotropic Cylindrical Layer Modeling. *Journal of Offshore Mechanics and Arctic Engineering, Volume 135, Issue 3*.

SAAD, Y. (2003). *Iterative Methods for Sparse Linear Systems* (2nd ed.). SIAM, ISBN 978-0-898715-34-7.

STROUD, I. (2006). *Boundary Representation Modelling Techniques.* London: Springer.

THIAGARAJAN, G., & ARAVAMUTHAN, V. (2002). Parallelization Strategies for Element-by-Element Preconditioned Conjugate Gradient Solver Using High-Performance Fortran for Unstructured Finite-Element Applications on Linux Clusters. *Journal of Computing in Civil Engineering*, 1-10. doi:10.1061/(ASCE)0887-3801(2002)16:1(1)

TONI, F.G. (2014). Ferramenta Eficiente para Análise Estrutural de Tubos Flexíveis Usando Macroelementos Finitos. In *Projeto de Conclusão de Curso.* Escola Politécnica da Universidade de São Paulo.

WINGET, J., & HUGHES, T. (1985). Solution algorithms for nonlinear transient heat conduction analysis employing element-by-element iterative strategies. *Computer Methods in Applied Mechanics and Engineering, 52 (1-3),* 711-815. doi:10.1016/0045-7825(85)90015-5