IMPROVING THE NUMERICAL EFFICIENCY OF A HIGH ACCURACY SHELL ELEMENT FOR SOFT TISSUES

A THESIS SUBMITTED TO

THE FACULTY OF ENGINEERING

BY

ABDAL AZIZ ABU SHARKH

IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF APPLIED SCIENCE

IN MECHANICAL ENGINEERING

OTTAWA-CARLETON INSTITUTE FOR MECHANICAL AND AEROSPACE ENGINEERING

UNIVERSITY OF OTTAWA

OTTAWA, CANADA

Aug 2019

© Abdal Aziz Abu Sharkh, Ottawa, Canada, 2019

Acknowledgement

I want to express my sincere gratitude to my parents; the ones who were with me in every step I took in life including this one and pushed me forward to be at the point I am at.

I would like to express my gratitude to my supervisor Dr. Michel Labrosse for his help and continuous support through this research. It was only possible because of his motivation, guidance, immense knowledge and professionalism.

Beside my supervisor I would like to thank my thesis committee: Dr. Jochen Lang and Dr. James McDonald for their insightful comments and questions that ensured the research was proceeding on the right track.

Last but not least, I thank all my family especially my uncle Mohammed Fathi for all his efforts, friends, community and nation for all their support.

Abstract

For the finite element (FE) simulation of relatively thin organs under complex dynamic loadings that are relevant in the biomedical engineering field, shell elements, compared to volume elements, have the potential to capture the whole thickness of the organ at once. Shell elements, are also known to feature efficiently large critical time steps, ensuring competitive computational times in dynamic structural analysis projects. As an improvement to the tools available for modelling and analysis, a new general nonlinear thick continuum-based (CB) shell FE embedded in an updated Lagrangian formulation and an explicit time integration scheme was recently developed. It can account for irregular and complex geometries, and hyper-elastic, large, nearly incompressible anisotropic 3D deformations characteristic of soft tissues. The original proof of concept was developed in MATLAB, which despite known advantages, is very slow. As a result, computational times, even for simple problems, have not been competitive. Therefore, the present work focused on re-writing the code in an efficient programming language with execution speed in mind in order to compete with the available elements which, in spite of having inferior capabilities, have better running times. In addition, a programming algorithm was needed to improve running time. Once it was implemented, the running time was reduced in half on a benchmark problem. Optimization was then exploited to introduce workarounds and design improvements that reduced running time further to 95% of its original value. The new version of the code was implemented in C++ and reached the goal of reducing running time while maintaining the expected functionality.

Contents

1 Introduction ... 1
1.1 Rationale ... 2
1.2 Objective ... 2
1.3 Contributions ... 2
1.4 Thesis organization ... 2
2 Literature review ... 4
2.1 Soft tissue constitution and mechanics ... 4
2.2 Momenan's shell element ... 4
2.2.1 Building assemblage matrices ... 12
2.3 The original MATLAB code ... 12
2.3.1 Preprocessor ... 14
2.3.2 Solver ... 15
2.3.3 Post processor ... 18
2.4 Algorithm of the original code ... 20
2.4.1 Pseudo code for the preprocessor ... 20
2.4.2 Pseudo code for the solver ... 20
2.4.3 Pseudo code of post processor ... 22
2.5 Software requirements ... 22
2.5.1 Available FEA packages ... 23
2.6 Notes on programming environments ... 25
2.7 Knowledge gap ... 26
2.7.1 Refining the existing algorithm to solve the necessary equations ... 26
2.7.2 Optimizing the code ... 26
2.7.3 Migrating the code to C++ ... 26
3 Methods ... 27
3.1 Experiment description ... 27
3.2 Refining the original software algorithm from momenan ... 38
3.2.1 Improvements on preprocessor ... 40
3.2.2 Improvements on solver ... 41
3.2.3 The new algorithms ... 41
3.2.4 Module-by-module breakdown of the solver ... 43
3.2.5 Improvements on the functions ... 44
3.3 Optimizing the code ... 48

3.3.1 Initial runs and testing methods 48
3.3.2 Vectorization 49
3.3.3 Other refactoring improvements 50
3.4 Migrating the code to C++ 50
3.4.1 Implementation 50
3.4.2 Migration of the code 51
4 Results and discussion 60
4.1 Implementation of the proposed algorithm in MATLAB 60
4.2 The optimization 61
4.3 The migrated code 63
4.4 Overall performance 63
5 Conclusion 68
5.1 Brief summary and major contributions 68
5.2 Recommendations for future work 68
6 References 70

List of Figures

Figure 1: Momenan's shell element with 9 master nodes shown as solid circles in the mid-plane of the structure. 5

Figure 2: Left: Lamina coordinate system at node $\boldsymbol{a}$ of a shell finite element, shown on a typical lamina. Right: Independence of the third axis of the lamina coordinate system from the fiber direction, as the fiber is the local normal to the lamina in the undeformed configuration (Momenan, 2016)........ 8

Figure 3: Original algorithm as developed by Momenan (2016)........ 13

Figure 4: Original preprocessor output showing a shell element mesh of a quarter cylinder to be pressurized and deformed. 15

Figure 5: The different parts of the original solver. 16

Figure 6: Example of graphical output from the original post processor. 19

Figure 7: Comparison between original and new implementation in terms of data structure set up 39

Figure 8: Elements setup and representation in a connectivity vector. 40

Figure 9: Comparison of running time on benchmark problem between MATLAB original code (OC), the MATLAB code after applying the algorithm and optimization (COA) and the C++ code (CC). 64

Figure 10: Comparison between the number of elements, and running time (in hours), for OC and COA cases. 65

Figure 11: Comparison between the running time (in hours) for 50-100 elements 67

List of Tables

Table 1: Comparison between available FEA packages. 24

Table 2: Some variables and their sizes in the original implementation, where n is the number of elements. 28

Table 3: Summary of functions 37

Table 4: Runtime of each function according to Matlab profiler 38

Table 5: Some variables and their sizes in the new implementation. 39

Table 6: Run time of the improved functions according to Matlab profiler 47

Table 7: Comparison between the original code run time for the functions and the new code after algorithm modification. 48

Table 8: Running time before optimization. 49

Table 9: Comparison between available C++ libraries for FEA. 51

Table 10: Comparison of running time on benchmark problem between the MATLAB original code (OC) and the MATLAB code after applying the algorithm (CAA). 60

Table 11: Comparison of running time on benchmark problem between the MATLAB code after applying the algorithm (CAA), and the MATLAB code after applying the algorithm and optimization (COA). 62

Table 12: Comparison of running time on benchmark problem between the MATLAB code after applying the algorithm and optimization (COA) and the C++ code (CC). 63

Table 13: Comparison between the running time of the solver for different numbers of shell elements in the benchmark problem using the MATLAB original code (OC) and the MATLAB code after applying the algorithm and optimization (COA), on the 32-GB machine. 64

Table 14: Simulations with 50, 60, 70, 80, 90 and 100 elements using the new code. 65

1 INTRODUCTION

Research has shown that some diseases develop due to changes in cell mechanics, extracellular matrix structure, or in the mechanisms by which cells sense and respond to mechanical signals. Hence, proper modelling and understanding of accurate biological tissue deformation and mechanical stress are needed and may also allow for the simulation of surgical procedures.

Different from hard tissues, such as bone, soft tissues cannot be described using classical engineering models and require advanced tools for the purpose of modeling and simulation. One of the most powerful and suitable tools for the task is finite element (FE) analysis, given the complexity of the shapes and material properties involved. Many available constitutive models for different types of soft tissues rely on the definition of a strain energy function (i.e. hyper-elasticity). The process of selecting the fitting mathematical form of the strain energy function is dependent on multiple principles of continuum mechanics, and the material constants associated with a specific strain energy function can be obtained from an adequate set of experiments.

Typically, hyper-elastic materials models are implemented using volume finite elements such as bricks (ABAQUS, 2005; LS-DYNA, 2011; Segal, 2010). As a result, several brick elements through the thickness are needed to properly capture the bending deformations of soft tissues. As the thickness of soft tissue membrane can be small (less than 1 mm), very small elements are needed. This leads to small critical time steps that increase the calculation times in dynamic analyses. On the other hand, shell elements can potentially capture the bending behaviour of the whole tissue thickness at once and be more computationally efficient.

However, not all the available shell elements can be used to model soft tissues for reasons detailed later. Momenan and Labrosse proposed a new shell element that overcomes the issues of available shell elements (Momenan and Labrosse, 2018 a, b), but their numerical implementation in MATLAB led to lengthy calculations (e.g. 46 hours for a simple problem explored in Section 3.1). Such running times defeat the purpose of using shell elements instead of brick elements. For the new element to serve its purpose, a more efficient implementation is needed.

1.1 RATIONALE

The new shell element that accurately implements incompressible, anisotropic and hyper-elastic constitutive relations in large tensile and bending deformations, as well as being insensitive to initially irregular elements needs to be coded in a computationally efficient manner to really gauge its potential with respect to existing elements.

1.2 OBJECTIVE

The goal of the present work is to achieve higher numerical performance while not affecting the accuracy of the results for the new shell finite element in order to enable and support the adoption of the new shell element in more advanced research projects.

1.3 CONTRIBUTIONS

To develop an algorithm for the shell element, a thorough understanding of the derivation of this element, and of the implementation of computational methods for numerical calculations with software code was needed.

The most important contributions of the proposed work lie in the improved speed and efficiency of the new CB shell element computations. They were achieved by the carefully thought out use of specific numerical methods. In particular, improvements in speed were achieved by implementing vectorization for matrix operations, and functions. They were also achieved by migrating the code to a more efficient language for computation, building the libraries needed for such specific operations, and properly selecting original functions that could be streamlined in the calculations. Overall efficiency was achieved by reducing the number of expanding elements, matrices multiplication operations, long loops and omitting the loop within loop code lines. The results were then thoroughly tested and verified.

1.4 THESIS ORGANIZATION

The second chapter presents some background information about soft tissue modelling, and relevant constitutive equations. It also shows the advantage of the new shell element over existing ones. It provides information about available FEA packages and programming environments that can be used for the implementation of the new shell element. It also introduces the original code algorithm. Finally, the knowledge gap to be filled by this thesis is presented.

The third chapter describes in detail the study case used in the experiment. It introduces the methods followed to improve running time, while showcasing the C++ implementation of the full program.

The fourth chapter presents the results achieved by the methods explained in the third chapter and discusses the results, along with the overall performance of the proposed implementation.

The last chapter summarizes the work and presents some recommendations for future work.

2 LITERATURE REVIEW

2.1 SOFT TISSUE CONSTITUTION AND MECHANICS

One of the main characteristics of biological soft tissues, which motivates their study by numerical analysis, is their ability to sustain non-infinitesimal (i.e. finite) deformations under normal conditions. It was shown that the aligned fibrous structure of soft tissues gives rise to anisotropic hyperelasticity in the physiological ranges of strain rates (Fung, 1967). In addition, the assumption of near-incompressibility is justified by the observation that a large portion (>70%) of the tissue volume is composed of water that appears to be tightly bound to the solid matrix (Fung, 1967).

Since soft tissues are complex in terms of their mechanical properties and their finite deformations, it is essential to study them using a suitable theoretical framework. One might assume that linear elasticity may be useful to model soft tissues as they are with bone tissues, but that is not the case due to the large strains soft tissues undergo that qualify as geometric nonlinearities. Another reason is the inherent material nonlinearity that exists in the relationship between stress and strain in soft biological tissues. Therefore, the stiffness (i.e. ratio between stress and strain) of soft tissues varies with the deformation.

2.2 MOMENAN'S SHELL ELEMENT

The presentation below aims to synthesize the development of a specific type of finite element for soft tissue dynamics, namely a shell element, as carried out by Momenan in her PhD dissertation (Momenan, 2016), and publications (Momenan & Labrosse, 2018 a&b). Full details, motivations, justifications and bibliographical references are provided in these documents, and for convenience, the same notations are used in the equations below.

Briefly, a shell element shown in Fig. 1 represents a flat or curved structure that is thinner in one direction compared to the other two directions. It is designed to capture the in-plane (membrane), as well as transverse (bending) mechanical behaviors of the structure, with typically only one shell element through the thickness of the structure. This is expected to be computationally advantageous over the non-specialized three-dimensional finite elements, for

which several elements would be required through the thickness of the structure to properly capture its membrane and bending behaviors.

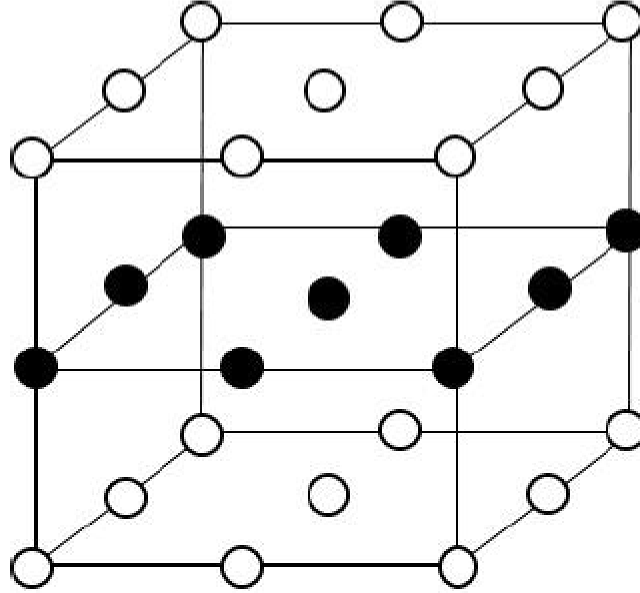

Figure 1: Momenan's shell element with 9 master nodes shown as solid circles in the mid-plane of the structure.

For any deformable structure in dynamics, derivations from Newton's second law of motion ($m\boldsymbol{a} = \boldsymbol{f}$, i.e. mass times acceleration equals force) provide the following governing equation, written in matrix form as a result of the spatial discretization of the structure into finite elements defined by nodal points (or nodes):

$$[\,^{\tau}M]\{^{\tau}_{0}\ddot{U}\} = \{^{\tau}_{0}R\} - \{^{\tau}_{\tau}F\}. \tag{2.1}$$

In this equation, known as the total (as opposed to incremental) updated Lagrangian formulation (Momenan, 2016), loads and deformations are measured from the underformed configuration (i.e. at time 0). Herein, $[\,^{\tau}M]$ is the mass matrix of the deformable structure, $\{^{\tau}_{0}\ddot{U}\}$ is the vector of the accelerations of the nodal points, $\{^{\tau}_{0}R\}$ is the vector of externally applied loads at the nodes, $\{^{\tau}_{\tau}F\}$ is the vector of internal nodal point forces, and τ is the time of the analysis.

The solution of (2.1) is classically obtained through a timewise discretization using the central difference method, whereby

$$\{{}_{0}^{\tau}\ddot{U}\} = \frac{\{{}^{\tau-\Delta\tau}_{0}U\} - 2\{{}_{0}^{\tau}U\} + \{{}^{\tau+\Delta\tau}_{0}U\}}{\Delta\tau^2}, \tag{2.2}$$

with time step$\Delta\tau$. Convergence of the solution is ensured as long as the scheme is numerically stable, which is achieved when$\Delta\tau < \Delta\tau_{\mathrm{critical}}$. In the literature, several methods have been developed to determine $\Delta\tau_{\mathrm{critical}}$ based on the material properties and dimensions of the smallest finite elements in the structure to be analyzed. Momenan also developed her specific method, as will be mentioned later, with Equation (2.17).

Additionally, methods exist in the literature to transform the full matrix $[\,{}^{\tau}M]$ in (2.1) into a diagonal equivalent matrix$[\,{}^{\tau}M_{ii}]$, such that (2.1) and (2.2) can be rewritten as a numerically efficient time-marching scheme where no computationally costly inversion of full matrices is required:

$$\{{}^{\tau+\Delta\tau}_{0}U\} = \frac{\Delta\tau^2}{[\,{}^{\tau}M_{ii}]}(\{{}_{\mathbf{0}}^{\boldsymbol{\tau}}R\} - \{{}_{\tau}^{\boldsymbol{\tau}}F\}) + 2\{{}_{0}^{\tau}U\} - \{{}^{\tau-\Delta\tau}_{0}U\}. \tag{2.3}$$

In (2.3), the terms on the right-hand side are known at time τ, and can be used to compute the solution at time $\tau + \Delta\tau$. At time 0, the approximation

$$\{{}^{-\Delta\tau}_{0}U\} = \{{}_{0}^{0}U\} - \Delta\tau\{{}_{0}^{0}\dot{U}\} + \frac{\Delta\tau^2}{2}\{{}_{0}^{0}\ddot{U}\} \tag{2.4}$$

can be used to get the solution scheme started based on known initial conditions $\{{}_{0}^{0}U\}$, $\{{}_{0}^{0}\dot{U}\}$, and $\{{}_{0}^{0}\ddot{U}\}$ which are the displacements, velocities and accelerations at time 0, respectively.

The calculation of $\{{}_{\mathbf{0}}^{\boldsymbol{\tau}}R\}$ in (2.3) requires knowledge of the externally applied point forces and moments, surface tractions (e.g. pressures) and body forces (e.g. gravity). The contributions from all these loads are distributed between the nodes of the elements of the structure, based on their mode of application and the dimensions of the elements. Again, the details are skipped for clarity, and because they are very well established in the finite element literature.

The $\{{}_{\tau}^{\boldsymbol{\tau}}F\}$ contribution in (2.3), on the other hand, stems from those forces that develop within the structure as deformations take place within it. They can readily be determined for simple materials, such as linearly elastic, isotropic, homogeneous materials. However,

complications arise in the case of interest here, where hyperelastic, anisotropic and nearly incompressible materials (representative of biological soft tissues) are to be analyzed.

As detailed in the following paragraphs, in addition to involving the constitutive equations describing the material behavior of the structure, internal forces depend on the element technology including the shape of the finite element, number and relative locations of the nodes, the number and nature of the degrees of freedom – displacements and rotations – assigned to the nodes, and the underlying mechanical theory for the element – e.g. here, shell element.

Namely, the basic equation for $\{{}^{\tau}_{\tau}F\}$ is

$$\{{}^{\tau}_{\tau}F\} = \int_{{}^{\tau}V} [{}^{\tau}_{\tau}B_L]^T \{ {}_{\tau}\sigma^l \} d\, {}^{\tau}V, \tag{2.5}$$

where ${}^{\tau}V$ represents the volume of the structure at time τ, $[{}^{\tau}_{\tau}B_L]$ is referred to in the finite element literature as the linear displacement transformation matrix, and $\{ {}^{\tau}\sigma^l \}$ gathers the Cartesian components of the Cauchy stress into a vector formatted according to the Voigt notation.

For hyperelastic materials, the Cartesian components of the Cauchy stress tensor are obtained from the strain energy density function W, that characterizes the material behavior through

$${}_{\tau}\sigma^l_{sr} = \frac{{}^{\tau}\rho}{{}^{0}\rho} {}^{\tau}_{0}F^l_{si} \frac{\partial {}^{\tau}_{0}W}{\partial {}^{\tau}_{0}E^l_{ij}} {}^{\tau}_{0}F^l_{rj}, \tag{2.6}$$

where superscript l denotes the lamina coordinate system of the material in which the constitute relationships of hyperelastic materials are typically expressed. In shell theory nomenclature, the lamina is parallel to the mid-surface of the shell, and the fiber is the local normal to the lamina in the underformed configuration (Figs. 1,2) – note that this definition of fiber is unrelated to the actual collagen fibers present in biological soft tissues. In (2.6), ρ denotes the mass density of the material at time τ or time 0, depending on the left superscripts. The Cartesian components of the Green strains (used in the context of finite, i.e. non-infinitesimal, deformations characteristic of hyperelastic materials) is denoted by ${}^{\tau}_{0}E^l_{ij}$, and ${}^{\tau}_{0}F^l_{uv}$ denotes the Cartesian components of the transformation gradient tensor, in the lamina coordinate system, at time τ, with respect to the initial configuration (i.e. at time 0).

From classical continuum mechanics theory, the Green strains in (2.6) are obtained through

$$ {}_{0}^{\tau}E_{ij}^{l} = \frac{1}{2}\left({}_{0}^{\tau}F_{ki}^{l}\,{}_{0}^{\tau}F_{kj}^{l} - \delta_{ij}\right), \tag{2.7} $$

where δ denotes the Kronecker delta symbol. On the other hand, in (2.6) and (2.7), the Cartesian components ${}_{0}^{\tau}F_{uv}^{l}$ of the transformation gradient tensor at any point of the structure consist of the derivatives of the current position vector for that point with respect to the position vector of the same point in the initial configuration.

At this stage, it is important to keep in mind that different coordinate systems are involved in the calculations. Firstly, there is the global Cartesian coordinate system, to which any point of the structure can be referred. Secondly, as illustrated in Fig. 2 (left), there is a local lamina coordinate system at every point of any shell element mapped by parameters (r, s, t).

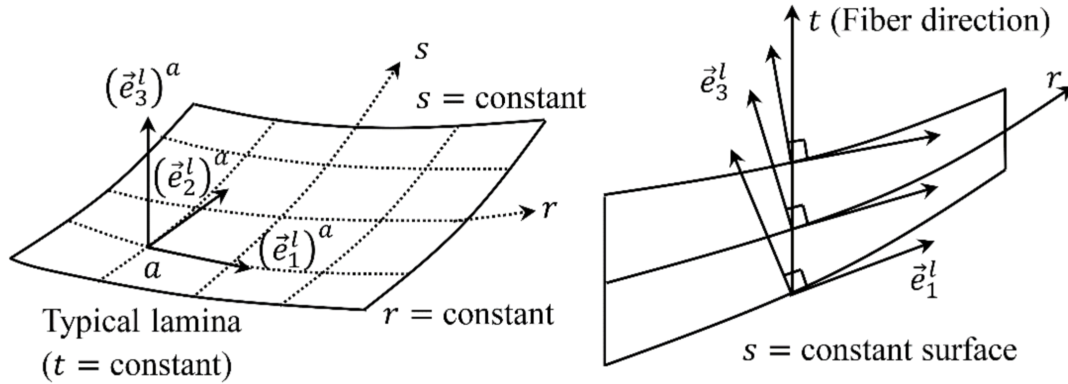

Figure 2: Left: Lamina coordinate system at node $\boldsymbol{a}$ of a shell finite element, shown on a typical lamina. Right: Independence of the third axis of the lamina coordinate system from the fiber direction, as the fiber is the local normal to the lamina in the undeformed configuration (Momenan, 2016).

Thirdly, a local fiber coordinate system must be defined to handle timewise variations in shell thickness with respect to a common framework. All three coordinate systems are related to each other through rotation matrices that depend on location and time (Momenan, 2016).

With this background, it is now possible to introduce the position vector for any point of a shell element in the structure:

$$ {}^{\beta}y(r,s,t) = \sum_{a=1}^{n_{en}} N_a(r,s)^{\beta}\overline{y_a} + \frac{t}{2}\sum_{a=1}^{n_{en}} N_a(r,s)^{\beta}h_a{}^{\beta}\hat{Y}_a, \tag{2.8} $$

where $\beta = 0$ for the initial configuration, and $\beta = \tau$ for the current configuration, ${}^{\beta}\overline{y_a}$ is the position vector of node a of a shell finite element on the mid surface in configuration β, $N_a(r,s)$ is the interpolation matrix (aka 2D shape function) associated with node a, n_{en} is the number of element nodes, ${}^{\beta}\hat{Y}_a$ is a unit vector along the fiber direction emanating from node a in configuration β, and ${}^{\beta}h_a$ is the nodal fiber length in configuration β. Briefly stated, (2.8), in combination with the relationships between the various coordinate systems, makes it possible to evaluate the Cartesian components of the transformation gradient tensor at any location, in any coordinate system, and at any time, as needed in (2.6) and (2.7).

For any given point in space, changes in positions as a function of time are classically defined as displacements and rotations (for instance, with respect to the initial configuration). The gradients of these spatial transformations are then used to define finite strains as in (2.5).

Additionally, the linearized (infinitesimal) strains $\{e^l\}$ more familiar to engineering mechanics need to be mentioned, not because they are used in the present calculations, but because the linear displacement transformation matrix $[{}_{\tau}^{\tau}B_L]$, that appears in (2.5), is normally used to compute them through

$$ \{e^l\} = [{}_{\tau}^{\tau}B_L]\{U\}, \tag{2.9} $$

where $\{U\}$ is the vector of incremental nodal point displacements and rotations over time step $\Delta\tau$. Note that $[{}_{\tau}^{\tau}B_L]$ is also involved in the determination of the critical time step mentioned previously. Using the Voigt notation,

$$ \{e^l\} = \left[\frac{\partial u_1^l}{\partial y_1^l}, \frac{\partial u_2^l}{\partial y_2^l}, \frac{\partial u_3^l}{\partial y_3^l}, 2\left(\frac{\partial u_1^l}{\partial y_2^l} + \frac{\partial u_2^l}{\partial y_1^l}\right), 2\left(\frac{\partial u_1^l}{\partial y_3^l} + \frac{\partial u_3^l}{\partial y_1^l}\right), 2\left(\frac{\partial u_2^l}{\partial y_3^l} + \frac{\partial u_3^l}{\partial y_2^l}\right)\right]^T, \tag{2.10} $$

with

$$
\left\{\begin{array}{c}
\left\{\begin{array}{c} \frac{\partial u_1^l}{\partial y_1^l} \\ \frac{\partial u_1^l}{\partial y_2^l} \\ \frac{\partial u_1^l}{\partial y_3^l} \end{array}\right\} \\
\left\{\begin{array}{c} \frac{\partial u_2^l}{\partial y_1^l} \\ \frac{\partial u_2^l}{\partial y_2^l} \\ \frac{\partial u_2^l}{\partial y_3^l} \end{array}\right\} \\
\left\{\begin{array}{c} \frac{\partial u_3^l}{\partial y_1^l} \\ \frac{\partial u_3^l}{\partial y_2^l} \\ \frac{\partial u_3^l}{\partial y_3^l} \end{array}\right\}
\end{array}\right\}
=
\left[\begin{array}{c}
\cdots [J_a^l]^{-1} \begin{bmatrix} \frac{\partial N_a}{\partial r}(\vec{e}_1^{\,l})_1^a & \frac{\partial N_a}{\partial r}(\vec{e}_1^{\,l})_2^a & \frac{\partial N_a}{\partial r}(\vec{e}_1^{\,l})_3^a & t\left(\frac{-h_a}{2}\right)\frac{\partial N_a}{\partial r}\left((\vec{e}_1^{\,l})^a\cdot(\vec{e}_1^{\,f})^a\right) & t\left(\frac{-h_a}{2}\right)\frac{\partial N_a}{\partial r}\left((\vec{e}_1^{\,l})^a\cdot(\vec{e}_2^{\,f})^a\right) \\ \frac{\partial N_a}{\partial s}(\vec{e}_1^{\,l})_1^a & \frac{\partial N_a}{\partial s}(\vec{e}_1^{\,l})_2^a & \frac{\partial N_a}{\partial s}(\vec{e}_1^{\,l})_3^a & t\left(\frac{-h_a}{2}\right)\frac{\partial N_a}{\partial s}\left((\vec{e}_1^{\,l})^a\cdot(\vec{e}_1^{\,f})^a\right) & t\left(\frac{-h_a}{2}\right)\frac{\partial N_a}{\partial s}\left((\vec{e}_1^{\,l})^a\cdot(\vec{e}_2^{\,f})^a\right) \\ 0 & 0 & 0 & \left(\frac{-h_a}{2}\right)N_a\left((\vec{e}_1^{\,l})^a\cdot(\vec{e}_1^{\,f})^a\right) & \left(\frac{-h_a}{2}\right)N_a\left((\vec{e}_1^{\,l})^a\cdot(\vec{e}_2^{\,f})^a\right) \end{bmatrix} \cdots \\
\cdots [J_a^l]^{-1} \begin{bmatrix} \frac{\partial N_a}{\partial r}(\vec{e}_2^{\,l})_1^a & \frac{\partial N_a}{\partial r}(\vec{e}_2^{\,l})_2^a & \frac{\partial N_a}{\partial r}(\vec{e}_2^{\,l})_3^a & t\left(\frac{-h_a}{2}\right)\frac{\partial N_a}{\partial r}\left((\vec{e}_2^{\,l})^a\cdot(\vec{e}_1^{\,f})^a\right) & t\left(\frac{-h_a}{2}\right)\frac{\partial N_a}{\partial r}\left((\vec{e}_2^{\,l})^a\cdot(\vec{e}_2^{\,f})^a\right) \\ \frac{\partial N_a}{\partial s}(\vec{e}_2^{\,l})_1^a & \frac{\partial N_a}{\partial s}(\vec{e}_2^{\,l})_2^a & \frac{\partial N_a}{\partial s}(\vec{e}_2^{\,l})_3^a & t\left(\frac{-h_a}{2}\right)\frac{\partial N_a}{\partial s}\left((\vec{e}_2^{\,l})^a\cdot(\vec{e}_1^{\,f})^a\right) & t\left(\frac{-h_a}{2}\right)\frac{\partial N_a}{\partial s}\left((\vec{e}_2^{\,l})^a\cdot(\vec{e}_2^{\,f})^a\right) \\ 0 & 0 & 0 & \left(\frac{-h_a}{2}\right)N_a\left((\vec{e}_2^{\,l})^a\cdot(\vec{e}_1^{\,f})^a\right) & \left(\frac{-h_a}{2}\right)N_a\left((\vec{e}_2^{\,l})^a\cdot(\vec{e}_2^{\,f})^a\right) \end{bmatrix} \cdots \\
\cdots [J_a^l]^{-1} \begin{bmatrix} \frac{\partial N_a}{\partial r}(\vec{e}_3^{\,l})_1^a & \frac{\partial N_a}{\partial r}(\vec{e}_3^{\,l})_2^a & \frac{\partial N_a}{\partial r}(\vec{e}_3^{\,l})_3^a & t\left(\frac{-h_a}{2}\right)\frac{\partial N_a}{\partial r}\left((\vec{e}_3^{\,l})^a\cdot(\vec{e}_1^{\,f})^a\right) & t\left(\frac{-h_a}{2}\right)\frac{\partial N_a}{\partial r}\left((\vec{e}_3^{\,l})^a\cdot(\vec{e}_2^{\,f})^a\right) \\ \frac{\partial N_a}{\partial s}(\vec{e}_3^{\,l})_1^a & \frac{\partial N_a}{\partial s}(\vec{e}_3^{\,l})_2^a & \frac{\partial N_a}{\partial s}(\vec{e}_3^{\,l})_3^a & t\left(\frac{-h_a}{2}\right)\frac{\partial N_a}{\partial s}\left((\vec{e}_3^{\,l})^a\cdot(\vec{e}_1^{\,f})^a\right) & t\left(\frac{-h_a}{2}\right)\frac{\partial N_a}{\partial s}\left((\vec{e}_3^{\,l})^a\cdot(\vec{e}_2^{\,f})^a\right) \\ 0 & 0 & 0 & \left(\frac{-h_a}{2}\right)N_a\left((\vec{e}_3^{\,l})^a\cdot(\vec{e}_1^{\,f})^a\right) & \left(\frac{-h_a}{2}\right)N_a\left((\vec{e}_3^{\,l})^a\cdot(\vec{e}_2^{\,f})^a\right) \end{bmatrix} \cdots
\end{array}\right]
\left\{\begin{array}{c} \vdots \\ u_1^a \\ u_2^a \\ u_3^a \\ \theta_1^a \\ \theta_2^a \\ \vdots \end{array}\right\}
\tag{2.11}
$$

where u_1^a, u_2^a, u_3^a are the Cartesian components of the translational displacements of node a with respect to the global coordinate system, and θ_1^a and θ_2^a are the rotational displacements about the unit vector in the fiber direction emanating from node a**.** These are the five degrees of freedom considered at each node of the element.

Because the Green strain tensor is symmetric, the Voigt notation is used to take advantage of the fact that, out of its 9 strain components, 6 are unique and non-zero in general, and can be arranged into the following vector:

$$\{\,{}_{\tau}E^l\} = [\,{}_{\tau}E_{11}^l \quad {}_{\tau}E_{22}^l \quad {}_{\tau}E_{33}^l \quad {}_{\tau}E_{12}^l \quad {}_{\tau}E_{13}^l \quad {}_{\tau}E_{23}^l]^T \tag{2.12}$$

In addition, because the material incompressibility characteristic of the soft tissues of interest is achieved when (Momenan, 2016)

$$\det\left(2\,{}_0^{\tau}E_{ij}^l + \delta_{ij}\right) - 1 = 0, \tag{2.13}$$

which is a linear equation in normal strain ${}_{\tau}E_{33}^l$, the material incompressibility constraint can be taken advantage of to express ${}_{\tau}E_{33}^l$ in (2.12) as a function of the other non-zero Green strain components (Momenan, 2016). In turn, this makes it possible to express the strain energy density function W used in (2.6) in terms of five strain components instead of six, and ensures that the constitutive equation in the lamina automatically satisfies the zero normal stress condition characteristic of shells, whereby $\frac{\partial\,{}_0^{\tau}W}{\partial\,{}_0^{\tau}E_{33}^l} = 0$ (Momenan, 2016).

As mentioned earlier, the time-marching solution scheme requires that the time step value used for computation stay below that of the critical time step. Calculation of the critical time step involves the determination of the so-called stiffness matrix defined as

$$\left[{}_{\tau}^{\tau}K_L\right] = \int_s \int_r \int_{-1}^{+1}\left[{}_{\tau}^{\tau}B_L^l\right]\left[{}_{\tau}C^l\right]\left[{}_{\tau}^{\tau}B_L^l\right]J\, dt\, dr\, ds \quad (2.14)$$

where $\left[{}_{\tau}^{\tau}K_L\right]$ is the stiffness matrix, $\left[{}_{\tau}^{\tau}B_L^l\right]$ is obtained from Equations (2.9-2.11), the constitutive tensor matrix $\left[{}_{\tau}C^l\right]$ is constructed using the Voigt notation and the components obtained from

$${}_{\tau}C_{mnpq}^l = \frac{{}^{\tau}\rho}{{}^{0}\rho}\, {}_0^{\tau}F_{mi}^l\, {}_0^{\tau}F_{nj}^l \frac{\partial^2\, {}_0^{\tau}W}{\partial\, {}_0^{\tau}E_{ij}^l\, \partial\, {}_0^{\tau}E_{rs}^l}\, {}_0^{\tau}F_{pr}^l\, {}_0^{\tau}F_{qs}^l, \quad (2.15)$$

and J is the determinant of the Jacobian matrix of transformation from lamina to fiber coordinates (Momenan, 2016). Calculation of the critical time step also involves the determination of the mass matrix already shown in Equation (2.1) and detailed as

$$\left[{}^{\tau}M^{consistent}\right]_a = \int_s \int_r \int_{-1}^{+1}\left[{}^{\tau}N\right]_a^T \left[{}^{\tau}N\right]_a \; {}^{\tau}\rho\, J\, dt\, dr\, ds, \quad (2.16)$$

where $\left[{}^{\tau}M^{consistent}\right]_a$ is the consistent mass matrix associated with Node a, $\left[{}^{\tau}N\right]_a$ is the interpolation matrix. The consistent mass matrix in (2.16) is further transformed using a lumping technique into a diagonal mass matrix $\left[{}^{\tau}M_{ii}\right]$ as used in Equation (2.2) (Momenan, 2016). The critical time step is ultimately derived (in an ongoing fashion, at each time step) from the largest eigenvalue of the system

$$\left[{}^{\tau}M_{ii}\right]^{-1}\left[{}_{\tau}^{\tau}K_L\right]\{\phi\} = \omega^2\{\phi\}. \quad (2.17)$$

Overall, Momenan's element is a quadrilateral shell element with nine nodes: four corner nodes, four mid-side nodes, and one central node (Fig. 1). The additional parameters appearing in (2.11) include rotation matrices between coordinate systems, transformation gradient tensors, and components of axes of specific coordinates expressed in the global coordinate system (Momenan, 2016). Although this presentation is incomplete by design, the purpose of showing (2.11) is to illustrate the complexity of the calculations associated with the proposed implementation of seemingly simple equations such as (2.3), which is repeated here for convenience:

$$\left\{{}_0^{\tau+\Delta\tau}U\right\} = \frac{\Delta\tau^2}{\left[{}^{\tau}M_{ii}\right]}\left(\left\{{}_0^{\tau}R\right\} - \left\{{}_{\tau}^{\tau}F\right\}\right) + 2\left\{{}_0^{\tau}U\right\} - \left\{{}_0^{\tau-\Delta\tau}U\right\}. \quad (2.18)$$

The focus of the present work is to make the numerical implementation of (2.18) as efficient as possible with respect to computation time.

2.2.1 Building assemblage matrices

In the finite element method, the structure to be analyzed is discretized into finite elements. Each element has a certain number of nodes, and adjacent elements share common nodes. The degrees of freedom (e.g. displacements, rotations) at two shared nodes are identical, while the loads are shared. Based on these principles, so-called assemblage matrices for the whole structure are constructed from individual matrices for each element in the model. It is the connectivity table that allows one to describe which local nodes within one individual element are used to construct that element in the whole structure.

2.3 THE ORIGINAL MATLAB CODE

The original shell element described above was implemented by Momenan in MATLAB to obtain a proof of concept (Momenan, 2016). As illustrated in Fig. 3, the script consisted of three main parts namely: the preprocessor, the solver and the post processor. A stop clock was implemented to keep track of the time steps and to determine the total running time. Some alerting messages were coded as well to keep track of when the main steps started and ended.

If mash refinement is needed

Preprocessor

FE Solver

Postprocessor

- Creation of geometry, mesh & connectivity table
- Material description
- Definition of loading & boundary conditions.

- Verification of input data
- Construction of shape functions
- Nodal discretization
- Integration points

- Adaption of geometry and fiber coordinate system

- Evaluation of Jacobian, lamina coordinate system, and measures of deformation
- Adaption of interpolation and stain-displacement transformation metrics
- Evaluation of constitutive relations
- Evaluation of internal and external force vectors
- Evaluation of stiffness and mass metrics

- Fiber length update algorithm
- Evaluation of critical time step

- FE Assemblage & application of boundary conditions
- FE Solution
- Adaption of displacements

Loop over load increments

Loop over number of elements

Loop over integration points

- Visualization of deformed geometry
- Stress & strain distribution
- Presentation of numerical results

Figure 3: Original algorithm as developed by Momenan (2016).

2.3.1 Preprocessor

The preprocessor combined the geometry files received from the user to create the finite element mesh and connectivity table that were to be used, and defined the variables necessary to contain the data. It also assigned variables for the material properties provided by the user, as well as the loading and boundary conditions for a specific problem to be solved.

The input to the preprocessor was a text file with all the data needed to process the element calculations. Variables, matrices (2D arrays) and 1D arrays were initialized and assigned in this stage. Some of them were fully defined and fixed in size, while some of them were defined later or expanded in size during calculations.

The expected output of the preprocessor was the variables ready to be in the solver, as well as a figure file that showed the initial finite element mesh representing the structure under study.

This part of the script contained unnecessary variables as well as many expanding arrays and vectors (with sizes larger than needed). Although the definition of such variable arrays and vectors did not affect the speed of the program at the beginning, it represented a waste of memory and imposed keeping up with all these definitions through the program. An additional coding efficiency remark is that some of these variables were repetitive, such as incremental variables that could easily be reset (or scope-controlled) instead of being saved to memory.

The script in this part did not have many comments to describe the functionality of each variable and help and distinguish between the actual variables that were to be used in the solver and the counters. This caused confusion and may lead to misinterpretation.

The running time of the preprocessor was about 35 seconds after which the mesh figure popped up (Fig. 4). The program then printed a message that the preprocessor part was completed, and the solver started.

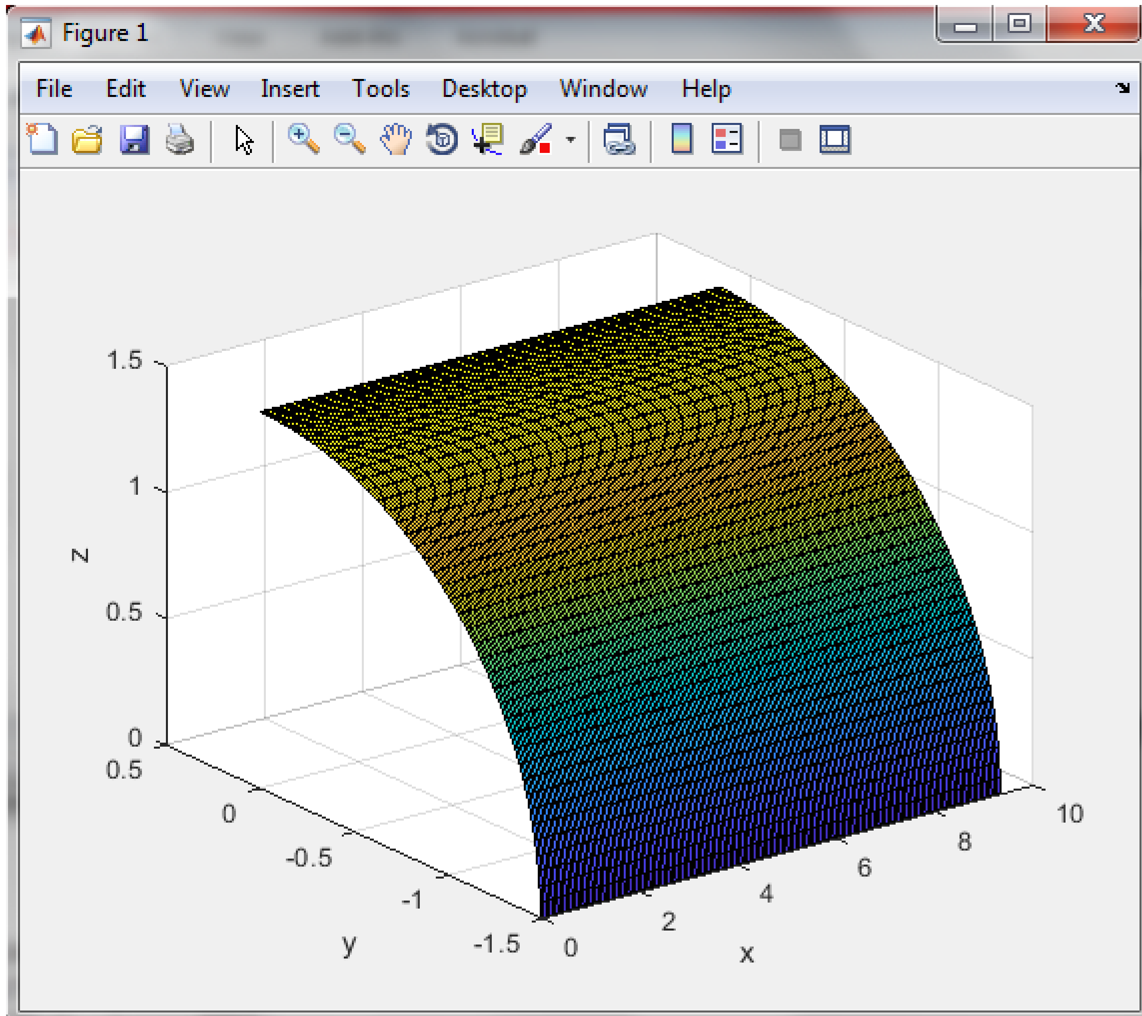

Figure 4: Original preprocessor output showing a shell element mesh of a quarter cylinder to be pressurized and deformed.

2.3.2 Solver

The second part of the script was the solver, which was the longest and most important part of the program. The solver used input data from the preprocessor and internal variables in a series of calculations. The solver had five sub-parts as shown in Fig. 5 and detailed below.

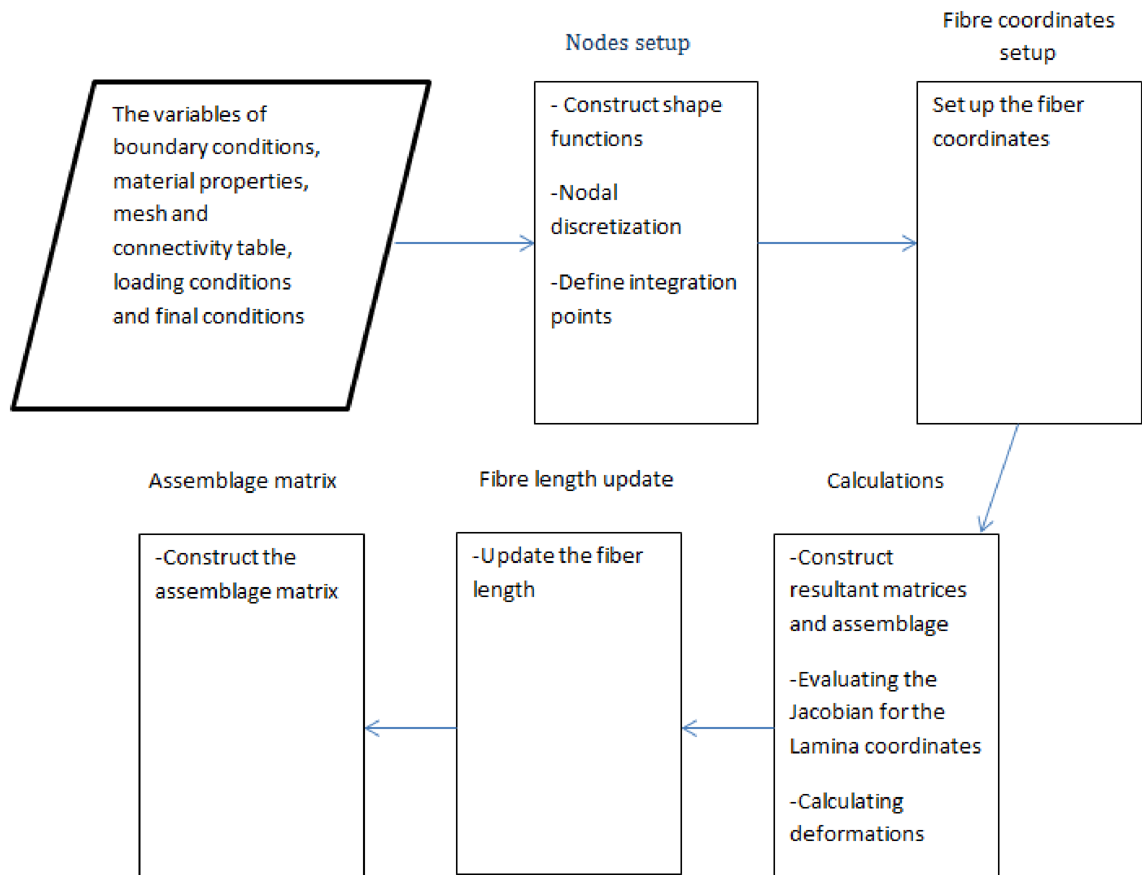

Figure 5: The different parts of the original solver.

2.3.2.1 Nodes setup

The first sub-part was responsible for verifying the input, constructing the shape functions and spatial nodal discretization, as shown in (2.1) and (2.2), and defining the integration points for the finite elements.

2.3.2.2 Fiber coordinates setup

The second sub-part prepared the environment needed for the calculations according to the geometry and fiber input. Such environment was handled through the definition of multiple coordinate systems in which the calculations were to be done.

2.3.2.3 Calculations

The third sub-part was the longest sub-part since it handled the main body calculations and constructing the matrices and the assemblage of the solution. It started by evaluating the

Jacobian matrix for the lamina in its respective coordinate system. It then went through the calculations of deformations under the given loads. The next step consisted of adapting the interpolation and strain displacement transformation matrices. Then came the evaluation of constitutive relations and internal and external force vectors. It finally evaluated the stiffness (for determination of the time step size) and mass matrices. These equations were repeated at every integration point of each element in the structure.

2.3.2.4 Fiber length update

The fourth sub-part updated the fiber length (to account for the material's near incompressibility) according to the deformation results obtained from the third sub-part and went over all the elements. Since the fiber length was affected, it was essential to make changes to the fiber coordinate system accordingly, which relied on the second sub-part of the solver.

2.3.2.5 Assemblage matrix

The fifth sub-part constructed the final assemblage by combining the results of all the element matrices. It then applied the boundary conditions. The displacements for the current time step were then determined. The process then restarted from the second sub-part to go through the calculations at the next time step. The end condition of the while loop is determined by the number of equilibrium iterations, as explained next.

A time step corresponds to a subdivision in the time-marching solution process used in transient dynamic analysis, as described in Equation (2.3). Many time steps are typically needed such that the structure under analysis may be simulated for the required duration. In addition, because of the nonlinearities involved in the problem to be solved, several equilibrium iterations, or sub steps are needed before the calculations in one time step converge.

The input to the solver was the data received as part of the problem and the variables initialized by the preprocessor along with the mesh setup. The solver consisted of 2,000 lines in a "while" loop to construct the assemblage system of equations for the whole structure at a given time step.

Within the while loop, there were 25 "for" loops. This code structure added more complexity and affected the readability of the code. Upon closer investigation, it was clear that if

this could be refactored to be more concise or maybe redesigned to ensure no redundant code was there, both performance (in terms of time and space) and readability would benefit.

Readability is critical in any modern library. The dynamic nature of software projects with teams (sometimes from around the world) working on the same project and team members working on code components they have not written, adds to the importance of readability. The use of nested loops seemed excessive in this regard. Our assumption was that it would be beneficial to explore how necessary these loop levels were, and if there was a way to simplify the code in general. The rest of the elements within the "while" loop were arithmetic operations for matrices and vectors, namely multiplications, divisions and additions.

The vectors were dynamically resizable and expanded according to the result of the operation applied to them. In many cases, a vector had to be expanded a couple of times before reaching its final size. The solver (similarly to the preprocessor) had many comments, most of them being code syntax that was commented out.

With the shell element developed, the running time of the solver to simulate the pressurization of a dog artery on a 4-GB RAM intel(R) Core(TM) i3-2350M CPU @ 2.30 GHz system was 350 hours; and on a 32-GB RAM with two Intel Xeon E5640 2.67 GHz 8-core processor, it was 46 hours. With a regular eight-node brick element model, the benchmark experiment in Section 3.1 took on average 8-15 min in commercial code LS-Dyna (Momenan 2016, Momenan and Labrosse 2018 b). The exceedingly long running time experienced with the shell element motivated us to search for ways to improve the solver.

The outputs of the solver were the results to be displayed by the post processor and printed as a text file containing the data for reference and future use.

2.3.3 Post processor

The post processor was the last step of the program. It presented the results of the calculations in a figure that was saved as a file. The input to the post processor was the data that had been output by the solver.

The main function of the post processor was the visualization of the deformed geometry, along with the stress and strain distributions.

For example, one possible output of the post processor was an image file that showed the pressure band in the wall of the pressurized dog artery under consideration (Fig. 6). The post processor also produced a text file containing the results from the solver, including but not limited to the number of iterations, pressure profiles, forces and geometry data.

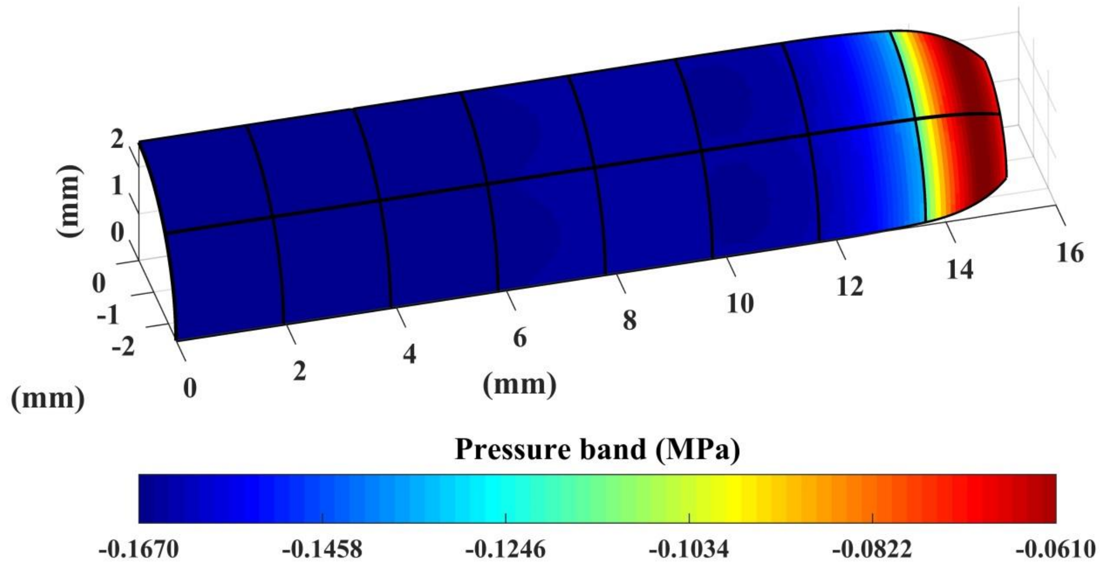

Figure 6: Example of graphical output from the original post processor.

The post processor contained 500 lines of code. The code featured instructions for writing and saving the text file for the values obtained. It also had the necessary function calls for plotting the image and saving it. It did not offer much room for major improvements.

Unlike the other two parts, the comments were few, and all of them were commented out code syntax. The running time of the post processor was 15-17 minutes depending on the size of the data being visualized and saved, and the latter depended on the number of time steps involved, which typically varied between 500 and 750.

2.4 ALGORITHM OF THE ORIGINAL CODE

It is common practice to describe algorithms using pseudo code. Although the original code was never presented in this manner, we wrote the following pseudo-code description to facilitate comparison between the original code and our proposed implementation.

2.4.1 Pseudo code for the preprocessor

1. **Input**: Text file with user specification
2. **Output**: picture file with shell elements in model
3. Initialize geometry properties
4. Initialize material properties
5. Initialize boundary conditions
6. Create the mesh grid
7. Initialize the shell element matrices
8. Generate and display shell elements in model
9. Open file
10. Save initial values
11. Close file
12. Print end of preprocessor

2.4.2 Pseudo code for the solver

1. **Input**: boundary conditions constants, mesh grid matrices, shell element matrices, material constants, geometry constants, connectivity table matrix, loading constants, loading matrices
2. **Output:** Assemblage matrix, external forces matrix, fiber length matrix and nodal displacement

3. **While** (present value < the final boundary condition value) **Do**:
4. Set the assemblage matrix to the previous assemblage matrix
5. Add the displacement to the assemblage matrix
6. Set fiber coordinate to zeros array
7. Set lamina to previous Jacobian transformation matrix
8. Set Jacobian transformation matrix of the current calculations
9. Set loop number to loop number plus one
10. **For** each element in the assemblage
11. **For** each node
12. Set degrees of freedom for the node
13. If the time step = value corresponding to the maximum load to be applied to the structure in a ramp-like fashion as a function of time
14. Plot the model
15. End if
16. End For
17. End for
18. **For** each element in the fiber length matrix
19. **For** each node

```
20.           Set the fiber coordinates relations
21.           For each nodal point in the node
22.                   Set the Gaussian nodes
23.                   Set the Gaussian points to zero
24.                   Set the Jacobean transformation matrix from fiber to lamina coordinates
25.                   Set the Lamina system
26.                   Initialize the tensors
27.                   Reorder the nodes
28.          End for
29.        End For
30.      End For
31.      For each element in the assemblage matrix
32.           For each node
33.                   Enter the nodal values
34.           End For
35.      End for
36.      For each element in the assemblage matrix
37.          For each node
38.              For each point
39.                  Calculate the edge forces on each point
40.              End For
41.          End For
42.      End For
43.      For each element in the assemblage matrix
44.          For each node
45.              For each point
46.                  Calculate the node forces at each point
47.              End For
48.          End For
49.      End For
50.      For each element in the assemblage matrix
51.          For each node
52.              For each point
53.                  Calculate the surface forces at each point
54.              End For
55.          End For
56.      End For
57.       For each element in the assemblage matrix
58.          For each node
59.              For each point
60.                  Calculate the external forces at each point
61.               End For
62.          End For
63.       End For
64.       For each element in the assemblage matrix
65.          For each node
```

```
66.             For each point
67.                 Calculate the internal forces at each point
68.             End For
69.             Add the surface, edge and node forces and set external forces to the result
70.         End For
71.     End For
72.     For each point in forces resultant matrix
73.             Enter values
74.     End For
75.   End For
76. End While
```

2.4.3 Pseudo code of post processor

1. **Input**: Assemblage matrix, external forces matrix, fiber length matrix and nodal displacement
2. **Output**: text file, picture file
3. Save values in an external file
4. Print out the results of the assemblage, forces and geometry matrices
5. Generate the pressure band distribution
6. Display the pressure band distribution
7. Save the picture

2.5 SOFTWARE REQUIREMENTS

For an improved implementation of the shell element discussed above, a software package is needed with certain requirements. The ideal software for the implementation would be an FEA software package compatible with the following:

1- Soft tissue modelling;

2- Explicit time integration;

3- Hyperelasticity material properties (defined herein as deformations up to 100%);

4- Anisotropic material properties;

5- Incompressible material properties;

6- CB shell elements;

7- Large rotational strains.

There are many available packages for FEA analysis and some of them that can be used for soft tissues analysis are explored below.

2.5.1 Available FEA packages

Momenan (Momenan, 2016) carried out a thorough review of current commercial and open source FEA packages that are generally used for soft tissue analyses, including ABAQUS, ANSYS, LS-DYNA and ADINA. It is important to note that, while these codes feature some of capabilities listed in our requirements, they are not open codes and as such, they do not enable users to define their own complex finite element. In addition, they do not feature all the required capabilities simultaneously.

For instance, LS-Dyna (“LS_DYNA”, 2018) supports large deformations, elastic materials, non-linearity and changing boundary conditions. It also provides a wide selection of elements and specifically shell elements with three, four, six and eight nodes including 3D shell elements. However, the shell elements are limited to small rotational strains and the software library does not support anisotropic hyperelastic material models for them, requiring the use of general 3D elements instead.

Similarly, ADINA ("ADINA", 2018) features advanced materials models and finite strain capabilities, it does not support anisotropic material modeling with shell elements.

In this context, we explored the possibility of using FEA packages that were potentially suitable for soft tissue analysis and allowed users to define their own finite element. In particular, we considered the following:

2.5.1.1 FeBio

FeBio is an open source FEA package written in C++ for biomedical applications. It includes FEpreview, FEBio and FePostview. The software offers a variety of viscoelastic, biphasic and isotropic as well as anisotropic non-linear elastic materials with large deformations for use in simulations.

The software library includes three, four and six nodal shell elements. However, the shell element mesh is not customizable which causes inconvenience since the shell element of interest is not among the traditional shell elements. In addition, large rotational strains were not mentioned as a feature in the available literature (Maas, Ellis, Ateshian & Weiss, 2012).

2.5.1.2 DUNE

Dune ("DUNE Numerics", 2018) is a modular toolbox based on C++ that supports FEA implementation. It is one of the few programs that support biological FEA applications (Jehl et al., 2014). It supports Lagrangian shape functions, as well as explicit and implicit time integrations. However, the literature available referred to material properties that were different from our needs (i.e. isotropic materials as opposed to hyperplastic anisotropic material we are working on) (Jehl et al., 2014; Dedner, 2007; Dedner, 2010).

2.5.1.3 MoFEM

MoFEM is a C++ based library that supports shell elements and customization, which gives flexibility in design. However, we could not identify applications in hyper-elasticity ("MoFEM ", 2018).

Table 1 presents a summary of our findings regarding the suitability of existing programs for our needs.

Table 1: Comparison between available FEA packages.

	Time integration	Anisotropic Hyperelasticity	Incompressible materials	Combined with CB shell element
ABAQUS	Explicit	No	Yes	No
ADINA	Explicit	No	Yes	No
DUNE	Explicit	No	N/A	No
FEBio	Implicit	No	Yes	No
LS-Dyna	Explicit	No	Yes	No
MoFEM	Explicit	No	Yes	No

Since none of the existing programs seemed to be suitable for our purposes, we were left with the option to implement a new one. The programming language of choice should at least have the following features:

1- Libraries that implement FEA operations.

2- An inherent speed advantage for calculations. Languages with close control over hardware offer advantages in terms of complex mathematical operations.
3- Powerful visualization tools in order to show the output of the experiment.

2.6 NOTES ON PROGRAMMING ENVIRONMENTS

C++ is one of the most widely used programming languages in different applications. Many of today's operating systems, system drivers, browsers and games use C++ as their core language. C++ ranks 2nd in popularity according to 2018 IEEE spectrum Top Programming Language ranking (Cass, 2018). It is also fast and because it is an extended version of C, the C part of it is very low level. This offers an interesting boost in speed that high-level languages like Python and Java might not feature.

The language has many libraries and software programs that were developed to solve FEA problems. Lib mesh (libMesh team, 2018) and Sparselizard ("Sparselizard finite element C++ library", 2018) are general FEA libraries that have been implemented for C++. Although these libraries enjoy widespread use, they have not been used in a manner that would demonstrate that they were suitable for our specific mechanical engineering equations and applications (Bauman et al., 2016; Palmer et al., 2014; Gijsenbergh et al., 2019).

On the other hand, MATLAB is commonly used for complex computations and calculations. It is optimized for operations involving matrices and vectors. Many mechanical engineering and biomedical applications have been implemented using MATLAB. FEA can be easily implemented since most of it is matrix and vector operations. The original code that we started from on was implemented in MATLAB. ("MATLAB", 2018)

Overall, there has been a noted trend for developers to choose C++ to implement FEA packages, while some packages were implemented in MATLAB and Fortran. This, along with advantages related to operation speed and memory control encouraged us to build the new package in C++.

2.7 KNOWLEDGE GAP

From Section 2.2, we can clearly see the potential of the new shell element developed by Momenan, but for it to be applicable and practical, there was a need for implementing it in a suitable software environment. The desired shell element theory had not been implemented into an efficient open source code. Therefore, software needed to be developed to handle the model presented. This had to be done through the development of an algorithm to solve the necessary equations and optimizing the code.

2.7.1 Refining the existing algorithm to solve the necessary equations

In the present work, a computational algorithm will be developed and presented for the Momenan shell element. It is expected to boost efficiency by reducing the time and memory cost of the program.

2.7.2 Optimizing the code

In the present work, we will use the advantages of MATLAB to improve the speed of the program.

2.7.3 Migrating the code to C++

We will migrate the code from MATLAB to C++ to make the solution more accessible to a general audience and enable future enhancements without the limitation of proprietary software.

3 METHODS

3.1 EXPERIMENT DESCRIPTION

For a clear assessment of performance of the program, we chose as criterion the running time to solve a certain problem. The benchmark example at hand was also used by Momenan, and involved the numerical simulation of the experimental quasi-static pressurization of a dog artery (Momenan, 2018). Briefly, an unpressurized artery of 1.21 mm internal radius and 0.56 mm thickness, and made of known hyperelastic and anisotropic material, was pressurized from 0 to 26.6 kPa (160 mmHg) in 0.1 s. The programming environment in which the numerical experiment was implemented MATLAB version R2017b for academic use. To assess the influence of RAM and processor, all implementations were completed on Windows 64-bit machines, with either an 4-GB RAM intel(R) Core(TM) i3-2350M CPU @ 2.30 GHz, or 32-GB RAM with two Intel Xeon E5640 2.67 GHz 8-core processor.

The geometrical and material information about the dog artery, as well as modeling information such as the number of finite element and the global boundary and loading conditions to be used was saved in a text file, which served as an input to the preprocessor.

The preprocessor then extracted the relevant data, and initiated the matrices, variables and constants and set the known values from the text file. In the dog artery problem, 50 shell elements were used, and the pressure load was applied as a time-wise ramp in 500 increments. Every element was described by nine nodes, each of which was defined by three coordinates. A summary of variables and their sizes as implemented in the original code is presented in Table 2. In this table, the elements array contains the element numbers. The coordinate's vector (expressed in the Cartesian, lamina and fiber coordinate systems, respectively) contain the positions of each node in relation to the surrounding nodes. The connectivity table identifies the constitution of the elements in terms of their global node numbers. The termination condition set the maximum number of iterations to be run for equilibrium.

The preprocessor sets the boundary conditions for surfaces, lines and points, as well as the loads applied, according to the input file. The program distributed the boundary conditions and loads to relevant finite elements and eventually to the relevant finite element nodes. Numerical

integrations using Gauss points were then carried out as needed, and the shell elements representing the initial geometry was plotted. The connectivity table listing the finite elements and their constitutive nodes was initialized, and the solver was started.

Table 2: Some variables and their sizes in the original implementation, where *n* is the number of elements.

Variable name	Program part where the structure is first created	Size	Input data type
Elements	Preprocessor	2D array [*n*,1]	Array [double]
Coordinates	Preprocessor	Vector [9]	2D array [double]
Connectivity table	Preprocessor	2D array [*n*,*n*]	double
Nodal points	Solver (Nodes setup)	2D array [1,*n*]	double
Termination condition	Solver	Constant = 2000	int
Element interpolation matrices	Solver (Calculations)	2D array [*n*,*n*]	double
Coordinates of assemblage matrix elements	Solver (Assemblage matrix)	Vector [9]	2D array [double]
Gauss points	Solver (Calculations)	2D array [*n*,*n*]	double

Each input in the element matrix is a coordinate vector (i.e. an array of 9 elements) and every input of the coordinate vector is the connectivity table (i.e. a $n \times n$ matrix). Each input in the connectivity table is of type double. The solver started with setting up the Jacobian transformation matrices which contained the derivatives of the current coordinates. The definition of some constant tensors happened afterwards. The while loop of the iterations then started by initiating the assemblage matrix. It also initiated seven force matrices, namely, the node, surface, edge, internal and external force matrices, along with internal and external boundary condition assemblage matrices.

The solver started looping over the elements in the assemblage matrix and set the initial values for the iteration. It then reset the Jacobian transformation matrix to the derivatives of the current coordinates with respect to the previous time.

The fiber coordinates system was then defined for each element. Gauss integration happened for every global node and the Jacobian matrix was reset again to suit the definition of the lamina coordinates. A set of tensors were then defined for the material properties. The solver then looped over every element and every node in the assemblage matrix to verify the connectivity and set the nodes in the right order.

The solver then looped over the force assemblage matrices one by one to enter the values of the forces applied according to the elements and nodes, by looping over every element and every node in the assemblage matrix.

The post processor was finally launched and all values were saved in a text file. The values were used to construct the deformed geometry of the dog artery and show specific results, such as stresses, strains or pressure bands.

For convenience of the presentation, we separated functions under analysis into sub-divisions according to the mathematical equations they represent. We also compared their performance between original and new implementations using the Matlab profiler tool, which allows one to track execution time information for individual lines or groups of lines in the code.

The subdivisions are the following:

1- **Updated Lagrangian formulation using explicit time integration**

Function ULExplicit implements Equation (2.4). It takes in velocity, displacement and acceleration vectors, and outputs the updated Lagrangian formulation using explicit time integration equations. This function is part of the nodes setup of the solver mentioned in 2.3.2.1.

Pseudo-code:

Inputs: initial displacement $\{U\}$, velocity $\{\dot{U}\}$ and acceleration $\{\ddot{U}\}$vectors and difference in time variable $\Delta\tau$.

Output: a vector with the input updated Lagrangian formulation using explicit time integration for position covering all elements $\{{}^{-\Delta\tau}_{0}U\}$ (to be used in Equations (2.1) to (2.3)). Vector size is *n*.

a. Initialize output vector $\{{}^{-\Delta\tau}_{0}U\} = \{0\}$
b. For each element *i*

c. $\quad {}^{-\Delta\tau}_{0}U_i = U_i - \Delta\tau\dot{U}_i + \frac{\Delta\tau^2}{2}\ddot{U}_i$

d. End for

2- **Position vector**

Function PositionV implements Equation (2.8). It takes in a vector of number of nodes in the element and performs calculations using the position vector at a node of a shell finite element on the mid-surface in the initial configuration, as well as the interpolation matrix associated with the element, and the nodal fiber length in initial configuration. It outputs a vector with the position of any point of a shell element in the structure. This function is part of fiber coordinates setup and calculations mentioned in Sections 2.3.2.2 and 2.3.2.3.

Pseudo code:

Inputs: number of nodes of an element n_{en} , position vector at a node of a shell finite element on the mid-surface in the initial configuration ${}^{\beta}_{0}\overline{y_a}$, element interpolation matrix $N_a(r,s)$, nodal fiber length in initial configuration${}^{\beta}_{0}h_a$ and unit vector ${}^{\beta}_{0}\hat{Y}_a$

Output: Position vector of all nodes in an element ${}^{\beta}y(r,s,t)$. Vector size is n_{en}.

a. Initialize lamina position sum variable lSum =0

b. Initialize fiber position sum variable fSum=0

c. For every input a in interpolation matrix input $N_a(r,s)$

d. For each "r,s" node position

e. lSum = $N_a(r,s) \times {}^{\beta}_{0}\overline{y_a}$

f. End For

g. For each "r,s" node position

h. fSum = $N_a(r,s) \times {}^{\beta}_{0}h_a \times {}^{\beta}_{0}\hat{Y}_a$

i. End for

j. For each node position in "r,s,t"

k. ${}^{\beta}y(r,s,t)$ = lSum + fSum

l. End For

m. End For

3- **Transformation matrix**

TransMtx implements Equation (2.11). It takes in the components of the unit vectors of the lamina and fiber coordinate systems, the derivatives of the element interpolation matrix

as well as the Jacobian matrix of the lamina coordinate system at a given node to output the matrix shown in (2.11). Note that in the following pseudo code implementation we are showing the order of inputs to the matrix. The order of the matrix is fixed regardless of the size of the problem. Also the constant of time t and the fiber length h_a get multiplied by interpolation matrix and the interpolation derivative matrix before they get passed to the function as an input. This function is part of calculations mentioned in Section 2.3.2.3.

Pseudo code:

Input: unit vectors of the lamina and fiber coordinate systems$\{\vec{e}_i^l\}^a$ $\{\vec{e}_i^f\}^a$, derivative of the element interpolation matrix derivative $[\frac{\partial N_a}{\partial s}]$, the interpolation matrix $[N_a]$ and the Jacobian matrix at a given node $[J_a^l]$.

Output: matrix in Equation (2.11) which represents the displacement transformation matrix setup for all nodes. Matrix is *n*×*n*.

a. For every input *a* in Jacobian matrix $[J_a^l]$
b. Input *a* in Equation (2.11) in corresponding order
c. End For
d. For every input *a* in derivative interpolation matrix $[\frac{\partial N_a}{\partial s}]$
e. For component *i*
f. $m = \frac{\partial N_a}{\partial s}_a \times \vec{e}_i^{l\,a} \times \vec{e}_i^{f\,a}$
g. Input *m* in Equation (2.11) in corresponding order
h. End For
i. End For
j. For every input *a* in interpolation matrix $[N_a]$
k. For component *i*
l. $k = N_{a_{ij}} \times \vec{e}_i^{l\,a} \times \vec{e}_i^{f\,a}$
m. Input *k* in Equation (2.11) in corresponding order
n. End For
o. End For

4- **Cauchy stress tensor**

Function CStress implements Equation (2.6). The function takes in the local mass densities in the initial and current configurations, the partial derivatives of the strain energy function describing the hyperelastic material with respect to the Cartesian components of the Green strains tensor, and the Cartesian components of the transformation gradient tensor in the lamina coordinate system with respect to the initial configuration. This function is part of calculations mentioned in Section 2.3.2.3.

Pseudo code:

Input: mass densities ρ, partial derivatives of the strain energy function $\partial {}_0^\tau W$, deformation characteristic matrix, ${}_0^\tau E_{ij}^l$ and the components of the transformation gradient tensor ${}_0^\tau F_{si}^l$, ${}_0^\tau F_{si}^l$.

Output: matrix of the local components of the Cauchy stress tensor ${}_\tau\sigma_{sr}^l$ for all elements. Matrix size is *n*×*n*.

a. For every Gaussian point *ij*

b. ${}_\tau\sigma_{sr}^l = \frac{\partial {}_0^\tau W}{\partial {}_0^\tau E_{ij}^l}$

c. End For

d. For every Gaussian point *si*

e. ${}_\tau\sigma_{sr}^l = \frac{{}^\tau\rho}{{}^0\rho} \times {}_0^\tau F_{si}^l \times {}_\tau\sigma_{sr}^l$

f. ${}_\tau\sigma_{sr}^l = {}_\tau\sigma_{sr}^l \times {}_0^\tau F_{rj}^l$

g. End For

5- **Green strain tensor**

Function GStrain implements Equation (2.7) to compute the Cartesian components of the Green strain tensor at any point of interest from the components of the transformation gradient tensor and the identity matrix used to represent the Kronecker function. This function is part of calculations mentioned in Section 2.3.2.3.

Pseudo code:

Input: components of the transformation gradient tensor ${}_0^\tau F_{ki}^l$, ${}_0^\tau F_{kj}^l$, Kronecker function matrix δ_{ij}

Output: Matrix that represents the Cartesian components of the Green strain tensor ${}_{0}^{\tau}E_{ij}^{l}$ covering all elements. Matrix is size *n*×*n*.

a. For every *ki* component of transformation gradient tensor
b. ${}_{0}^{\tau}E_{ij}^{l} = {}_{0}^{\tau}F_{ki}^{l} \times {}_{0}^{\tau}F_{kj}^{l}$
c. End For
d. For every *ij* components of transformation gradient tensor
e. ${}_{0}^{\tau}E_{ij}^{l}$=1/2 × (${}_{0}^{\tau}E_{ij}^{l} - \delta_{ij}$)
f. End For

6- **Plane stress state**

PlStress conditions the Green strains to be compatible with 1) the incompressibility constraint implemented exactly by solving linear Equation (2.13) for the normal Green strain component in the lamina, and 2) the plane stress state described by the zero normal stress condition. This function is part of calculations mentioned in Section 2.3.2.3.

Pseudo code:

Input: components of the Green strain tensor ${}_{0}^{\tau}E_{ij}^{l}, \delta_{ij}$

Output: Boolean to confirm plane stress values

a. For each component of Green strain tensor
b. $u = (2 \times {}_{0}^{\tau}E_{ij}^{l}) + \delta_{ij}$
c. End For
d. u = Det (resulting matrix) – 1
e. If $u = 0$
f. Return true
g. Else
h. Return false

7- **Force vector matrix**

Function ForceV implements Equation (2.5). The function embeds a volume integration method using Gauss points, and takes in the linear displacement transformation matrix, the components of the Cauchy stress tensor, and the determinant of the Jacobian matrix of transformation from the lamina to fiber coordinates systems. It outputs a matrix containing

the internal forces vectors for all the elements. This function is part of calculations mentioned in Section 2.3.2.3.

Pseudo code:

Input: linear displacement transformation matrix from (2.11) $[{}^{\tau}_{\tau}B_L]^T$ and the components of the Cauchy stress tensor from (2.6) using the Voigt notation $\{\ {}_{\tau}\sigma^l\ \}$, and determinant of Jacobian matrix. J^0

Output: internal force matrix according to (2.5) $\{{}^{\tau}_{\tau}F\}$ for all elements. Matrix size is $n \times 1$.

a. For every *il* in linear displacement transformation matrix components.

b. $\{{}^{\tau}_{\tau}F\} = \left[{}^{\tau}_{\tau}B_{Lij}\right]^T \{\ {}_{\tau}\sigma^l_i\}$

c. For every *ij* Gaussian point

d. $\{{}^{\tau}_{\tau}F\} = \{{}^{\tau}_{\tau}F\} \times J^0$

e. End For

8- **Stiffness matrix**

Function StiffMtx implements Equations (2.14) and (2.15). The function embeds a volume integration method using Gauss points, and takes in the linear displacement transformation matrix, the Cartesian components of the Green strain tensor, the Cartesian components of the transformation gradient tensor in the lamina coordinate system with respect to the initial configuration, and the determinant of the Jacobian matrix of transformation from the lamina to fiber coordinates systems. It outputs the stiffness matrix for all the elements. This function is part of calculations mentioned in Section 2.3.2.3.

Pseudo code:

Input: Strain energy gradient matrix $\partial^2 {}^{\tau}_{0}W$, linear displacement transformation matrix from (2.11) $[{}^{\tau}_{\tau}B^l_L]$, the components of the Green strain tensor ${}^{\tau}_{0}E^l_{ij}\partial, {}^{\tau}_{0}E^l_{rs}$ and of the transformation gradient ${}^{\tau}_{0}F^l_{pr}, {}^{\tau}_{0}F^l_{qs}, {}^{\tau}_{0}F^l_{mi}, {}^{\tau}_{0}F^l_{nj}$, and the determinant of the Jacobian matrix J, mass densities ${}^{\tau}\rho,\ {}^{0}\rho$.

Output: stiffness matrix according to (2.14) and (2.15) $[{}^{\tau}_{\tau}K_L]$ covering all elements. Matrix size is $n \times 1$.

a. For every *rs* Gauss point

b. For every *ij* in linear displacement transformation element

c. $d = {}_{0}^{\tau}F_{pr}^{l} \times {}_{0}^{\tau}F_{qs}^{l}$

d. End For

e. For every *ij* in linear displacement transformation element

f. $e = {}_{0}^{\tau}F_{mi}^{l} \times {}_{0}^{\tau}F_{nj}^{l}$

g. End For

h. For every *ij* in linear displacement transformation element

i. $g = \frac{\partial^2 {}_{0}^{\tau}W}{\partial {}_{0}^{\tau}E_{ij}^{l} \partial {}_{0}^{\tau}E_{rs}^{l}}$

j. End For

k. For every *ij* in linear displacement transformation element

l. ${}_{\tau}C_{mnpq}^{l} = \frac{{}^{\tau}\rho}{{}^{0}\rho} \times d \times e \times g$

m. End For

n. For every *ij* in linear displacement transformation element

o. $[{}_{\tau}^{\tau}K_L] = \left[{}_{\tau}^{\tau}B_L^l\right]\left[{}_{\tau}C^l\right]$

p. End For

q. For every *ij* in linear displacement transformation element

r. $[{}_{\tau}^{\tau}K_L] = [{}_{\tau}^{\tau}K_L]\left[{}_{\tau}^{\tau}B_L^l\right]J$

s. End For

t. End For

9- **Mass matrix**

Function MassMtx implements Equation (2.16). The function embeds a volume integration method using Gauss points, and takes in element the interpolation matrix, the current mass density of the material in the element, and the determinant of the Jacobian matrix of transformation from the lamina to fiber coordinates systems. It outputs the stiffness matrix for all the elements. This function is part of calculations mentioned in Section 2.3.2.3.

Pseudo code:

Input: Gauss point matrix, element interpolation matrix $[\,{}^{\tau}N]_a$, element interpolation matrix transpose $[\,{}^{\tau}N]_a^T$ mass density ${}^{\tau}\rho$, and Jacobian determinant J

Output: mass matrix according to (2.16) $\left[\,^{\tau}M^{consistent}\right]_a$ covering all elements. The matrix size is $n{\times}n$.

a. For every node a
b. $\left[\,^{\tau}M^{consistent}\right]_a = [\,^{\tau}N]_a^T[\,^{\tau}N]_a$
c. End for
d. For every node a
e. $\left[\,^{\tau}M^{consistent}\right]_a = \left[\,^{\tau}M^{consistent}\right]_a \times \,^{\tau}\rho \times J$
f. End for

10- **Discretization**

Function Discretization implements the central-difference timewise discretization described in Equation (2.2). It takes in displacement vectors at three different times and the time step variable. It outputs the average acceleration vector. This function is part of nodes setup mentioned in Section 2.3.2.1.

Pseudo code:

Input: displacement vectors $\{^{\tau-\Delta\tau}_{0}U\}, \{^{\tau}_{0}U\}, \{^{\tau+\Delta\tau}_{0}U\}$ and time step $\Delta\tau^0$

Output: acceleration vector according to Equation (2.2) $\{^{\tau}_{0}\ddot{U}\}$ covering all elements. The vector size is n.

11- **Critical time step**

Function TScritical implements Equation (2.17). It takes in stiffness matrix described in (2.14), and the diagonalized mass matrix obtained after lumping the matrix obtained form (2.16). It outputs the critical time step for all the elements. This function is part of fiber length update mentioned in Section 2.3.2.4.

Input: stiffness matrix $[^{\tau}_{\tau}K_L\,]$ and diagonalized mass matrix $[\,^{\tau}M_{ii}\,]^{-1}$

Output: Matrix with critical time step values according to (2.17) $\omega^2\{\phi\}$ covering all elements. The matrix size is $n{\times}n$.

a. For each element ii in diagonalized mass matrix
b. $\omega^2\{\phi\} = [\,^{\tau}M_{ii}\,]^{-1}[^{\tau}_{\tau}K_L\,]\{\phi\}$
c. End for

A summary of the functions described above is in Table 3. Table 4 shows the performance of each function according to the profiler and the percentage it needs of the total time. As can be seen from Table 5, most of the functions that took more time to run were functions including matrix multiplications and looping over each element for every operation, e.g. GStrain, TransMtx, StiffMtx, CStress, TScritical. As a result, assemblage, which called these functions repeatedly as shown in the pseudo-code, also took significant runtime.

Table 3: Summary of functions

	Input	Output
ULExplicit	$\{U\}, \{\dot{U}\}, \{\ddot{U}\}, \Delta\tau.$	$\{{}^{-\Delta\tau}_{0}U\}$
PositionV	n_{en} , ${}^{\beta}_{0}\overline{y_a}$, $N_a(r,s)$, ${}^{\beta}_{0}h_a$, ${}^{\beta}_{0}\hat{Y}_a$	${}^{\beta}y(r,s,t)$
TransMtx	$\{\vec{e}^{l}_{i}\}^a$ $\{\vec{e}^{f}_{i}\}^a$, $[\frac{\partial N_a}{\partial s}]$, $[N_a]$,$[J^{l}_{a}]$.	matrix in Equation (2.11)
CStress	ρ, $\partial {}^{\tau}_{0}W$, ${}^{\tau}_{0}E^{l}_{ij}$,${}^{\tau}_{0}F^{l}_{si}$, ${}^{\tau}_{0}F^{l}_{si}$.	${}_{\tau}\sigma^{l}_{sr}$
GStrain	${}^{\tau}_{0}F^{l}_{ki}$, ${}^{\tau}_{0}F^{l}_{kj}$, δ_{ij}	${}^{\tau}_{0}E^{l}_{ij}$
PlStress	${}^{\tau}_{0}E^{l}_{ij}$, δ_{ij}	Boolean to confirm plane stress values
ForceV	$[{}^{\tau}_{\tau}B_L]^T$,$\{ {}_{\tau}\sigma^{l} \}$, J^0	$\{{}^{\tau}_{\tau}F\}$
StiffMtx	$\partial^2 {}^{\tau}_{0}W$, $[{}^{\tau}_{\tau}B^{l}_{L}]$, ${}^{\tau}_{0}E^{l}_{ij}\,\partial$, ${}^{\tau}_{0}E^{l}_{rs}$, ${}^{\tau}_{0}F^{l}_{pr}$, ${}^{\tau}_{0}F^{l}_{qs}$, ${}^{\tau}_{0}F^{l}_{mi}$, ${}^{\tau}_{0}F^{l}_{nj}$, J, ${}^{\tau}\rho$, ${}^{0}\rho$	$[{}^{\tau}_{\tau}K_L]$
MassMtx	$[\,{}^{\tau}N]_a$, $[\,{}^{\tau}N]^T_a$, ${}^{\tau}\rho$, J	$[\,{}^{\tau}M^{consistent}]_a$
Discretization	$\{{}^{\tau-\Delta\tau}_{0}U\}$, $\{{}^{\tau}_{0}U\}$, $\{{}^{\tau+\Delta\tau}_{0}U\}$ and $\Delta\tau^0$	$\{{}^{\tau}_{0}\ddot{U}\}$
TScritical	$[{}^{\tau}_{\tau}K_L\,]$ and $[\,{}^{\tau}M_{ii}\,]^{-1}$	$\omega^2\{\phi\}$

Table 4: Runtime of each function according to Matlab profiler

	Run time (h)	Percentage
ULExplicit	0.87	1.8%
PositionV	2.03	4.35%
TransMtx	3.90	8.3%
CStress	3.28	7.01%
GStrain	3.93	8.4%
PlStress	1.97	4.2%
ForceV	2.61	5.56%
StiffMtx	3.49	7.43%
MassMtx	2.25	4.8%
Discretization	0.78	1.67%
TScritical	3.06	6.52%
Assemblage	5.76	12.28%
Solver	46.93	100%

3.2 REFINING THE ORIGINAL SOFTWARE ALGORITHM FROM MOMENAN

The original algorithm of the program in our hands was shown in Section 2.5. It was improved upon, as detailed in the following sections. In the original implementation, for the benchmark problem with 50 elements, the elements needed 9 matrices of 50 inputs, each identifying the corresponding axes of the three coordinates. For improved numerical efficiency, in the new implementation, the nine matrices were replaced by nine vectors of 9 inputs. The 9 coordinate matrices had many zero inputs with 9 significant inputs at most, representing the relations between the elements in a certain axis in a certain coordinate. Hence, the zero matrices

(i.e. sparse matrices) were reduced to a vector with the 9 significant inputs representing the connection of an element to the neighbouring elements around it. And the coordinates vector of 9 inputs that represent each axis contains one 9 input vector. Fig 7 shows the difference between the original and new implementation.

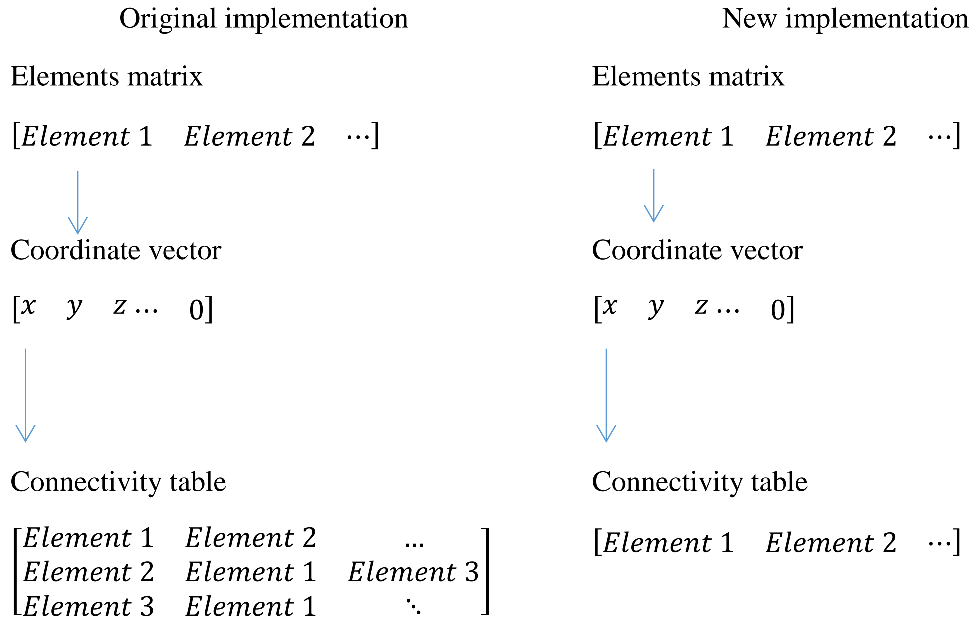

Figure 7: Comparison between original and new implementation in terms of data structure set up

The same thing was done to the assemblage coordinates and the Gauss points. The setup of the benchmark experiment after modifications is provided in Table 5.

Table 5: Some variables and their sizes in the new implementation.

Variable name	Size	Input Data type
Elements	2D array [n,1]	Array [double]
Coordinates	Vector [9]	Array [double]
Connectivty table	Vector [9]	double
Nodal points	2D array [1,n]	double
Termination condition	Constant = 2000	int
Element interpolation matrix	2D array [n,n]	double
Coordinates of assemblage matrix elements	Vector [9]	Array [double]
Gauss points	Vector [9]	double

The connectivity table stores the position value of each element in the respective axis. The first value in the vector is the value of the selected element. The next value is the element to the right of the selected element and the order continues in a clockwise fashion. Some of the boundary elements' vectors have zero values assigned which represent no connection. Figure 8 shows an example of the elements setup and its representation in a vector. Note that the number of elements and their orientation is identified by the user in the input file which the preprocessor receives.

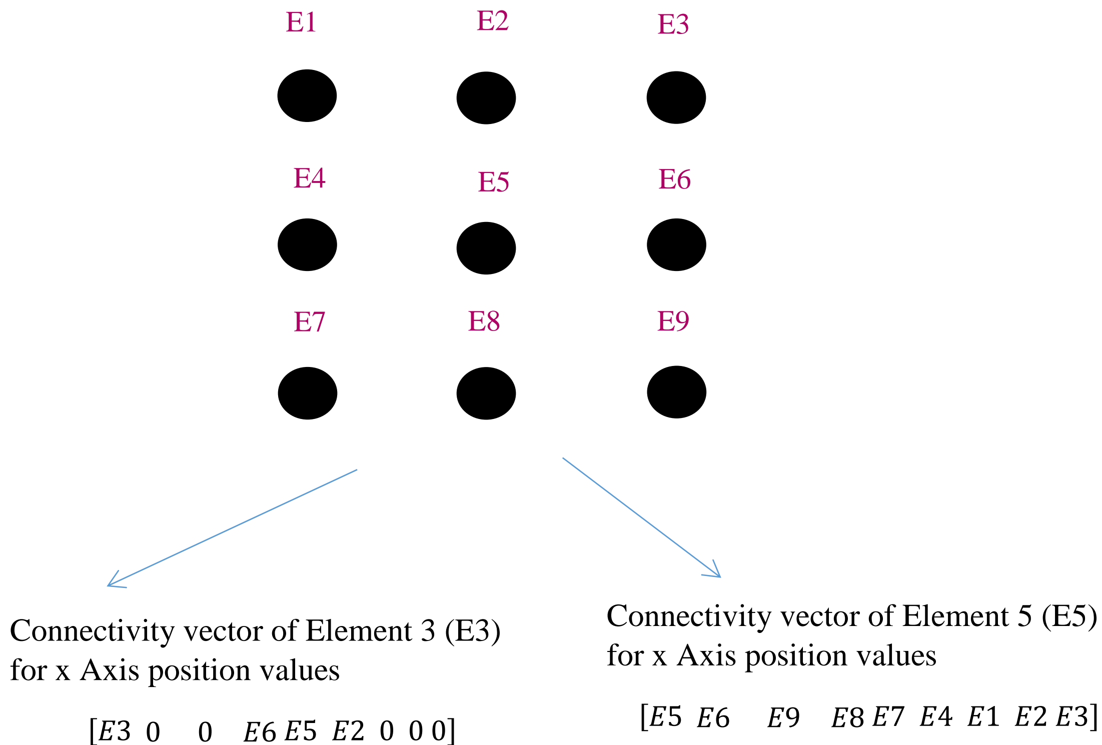

Figure 8: Elements setup and representation in a connectivity vector.

3.2.1 Improvements on preprocessor

The preprocessor algorithm of the new code was the same as the original one, but some more variables and constants, originally in the solver, were added to it. For instance, the initialization of the assemblage matrix was moved from the solver to the preprocessor to avoid the repetitive initialization of the assemblage matrix every single time the program looped over the solver code, which took longer running time. Similarly, the fiber length and the force load matrices were moved out from the solver loop to the preprocessor to avoid long running time and repetitive initializations.

Some other constants and variables that were not mentioned in the general algorithm were moved to the preprocessor. All counters and frequently redefined constants were moved out of the solver's loop to avoid reinitializing them every time the program loops.

3.2.2 Improvements on solver

We applied improvements to the solver algorithm to reduce the running time. One of the main improvements of the solver was obtained by merging the steps from 26-29 and from 30-35 of the pseudo-code. The original program looped twice over the assemblage matrix, once (from 31-35) for the nodal values after fixing the fiber length, and a second time (from 36-60) for calculating the forces exerted on the nodes. The new algorithm looped over the assemblage matrix once to enter the values and calculate the forces exerted on them at the same time instead of looping over it six times. The node variable itself was a vector of seven extra constants which were the force values added to the nodal points. Simply put, instead of looping three times over nested loops of the size of assemblage matrix twice, the new script loops once over the assemblage matrix with seven elements in a vector.

Lines 63-65 were eliminated in the new algorithm since there was no need for a force resultant matrix anymore. The elements of this matrix were instead added as another vector element to the nodal points in the assemblage matrix, which increased the readability of the program.

The if statement in lines 13-15 were removed to save running time since it was inside a loop, and the post processor will plot the final product in all cases.

3.2.3 The new algorithms

3.2.3.1 Pseudo-code of the preprocessor

1. **Input**: Text file with user specification
2. **Output**: picture file with the shell element
3. Initialize geometry properties
4. Initialize material properties
5. Initialize boundary conditions
6. Create the mesh grid
7. Initialize the shell element matrices
8. Generate and display the shell element
9. Open file
10. Save initial values

11. Close file
12. Print end of preprocessor
13. Initiate the assemblage matrix

3.2.3.2 Pseudo-code of the solver

1. **Input**: boundary conditions constants, mesh grid matrices, shell element matrices, material constants, geometry constants, connectivity table matrix, loading constants, loading matrices
2. **Output:** Assemblage matrix, external forces matrix, fiber length matrix and nodal displacement
3. **While** (present value < termination value) **Do**:
4. Set the assemblage matrix to the previous assemblage matrix
5. Add the displacement to the assemblage matrix
6. Set fiber coordinate to zeros array
7. Set Lamina to previous Jacobian transformation matrix
8. Set Jacobian transformation matrix of the current calculations
9. Set loop number to loop number plus one
10. **For** each element in the assemblage
11. **For** each node
12. Set degrees of freedom for each node
13. End For
14. End for
15. **For** each element in the fiber length matrix
16. **For** each node
17. Set the fiber coordinates relations
18. **For** each nodal point in the node
19. Set the Gaussian nodes
20. Set the Gaussian points to zero
21. Set the Jacobian transformation matrix from fiber to lamina coordinates
22. Set the lamina coordinate system
23. Initialize the tensors
24. Reorder the nodes
25. End for
26. End For
27. End For
28. **For** each element in the assemblage matrix
29. **For** each node
30. Enter the nodal values
31. Calculate the edge forces at each node
32. Calculate the node forces at each node
33. Calculate the surface forces at each node
34. Calculate the external forces at each node
35. Calculate the internal forces at each node
36. End For
37. Add the surface, edge and node forces and set the external forces to the result

38. End For
39. End For
40. End While

3.2.4 Module-by-module breakdown of the solver

To provide a better understanding of what the solver does exactly, we broke it down into five parts as discussed in Section 2.3.2.

3.2.4.1 Nodes set up

The first sub-part consisted of lines 3-5 in the algorithm. This sub-part was responsible for verifying the input by validating the boundary conditions and the previous assemblage matrix data on line 4. The shape functions were then constructed according to the given data, the previous data from the previous iteration, and the nodal discretization and Gauss integration points definitions on line 5.

3.2.4.2 Fiber coordinates setup

The second sub-part consisted of line 6. To start working on the fiber coordinates, the matrices and variables of the fiber coordinates needed to be reset every single time the program looped. Geometry and fiber input were then used to construct the Jacobian matrices to transform coordinates from the global to the fiber coordinate systems.

3.2.4.3 Calculations

The third sub-part consisted of lines 7-20. Lines 7-9 present the change in the coordinates systems from fiber to lamina through the Jacobian matrices constructed from the given data from the user, the fiber coordinates matrix and the previous lamina matrix. Afterwards, the program incremented the counter to keep track of the iterations, whose maximum number was also set by the user as input to the preprocessor. The process of updating the fiber length started after that, and continued over the next steps in lines 10-14. The program then went through the calculations of deformations. The next step consisted in adapting the interpolation and strain displacement transformation matrices. Then came the evaluation of constitutive relations and internal and external force vectors. The program finally evaluated the stiffness (for determination of the time step size) and mass matrices. These equations were repeated at every integration point of each element in the structure.

3.2.4.4 Fiber length update

The fourth sub-part consisted of lines 21-27 and updated the fiber length (to account for the material's near incompressibility) according to the deformation results obtained from the third sub-part and went over all the elements. Since the fiber length was affected, it was essential to make changes to the fiber coordinate system accordingly, which relied on the second sub-part of the solver.

3.2.4.5 Assemblage matrix

The fifth sub-part consisted of lines 28-40 and constructed the final assemblage matrix by combining the results of all the element matrices. It then applied the boundary conditions. The displacements for the current time step were then determined. The process then restarted from the second sub-part to go through the calculations at the next time step.

The input to the solver was the data received as part of the problem and the variables initialized by the preprocessor along with the mesh setup. The solver consisted of 2,000 lines in a "while" loop to construct the assemblage system of equations for the whole structure at a given time step.

Within the while loop of the code, there were 25 "for" loops. This code structure added more complexity and affected the readability of the code. Upon closer investigation, it was clear that if this could be refactored to be more concise or maybe redesigned to ensure no redundant code was there, both performance (in terms of time and space) and readability would benefit. Readability is critical in any modern library. The dynamic nature of software projects with teams (sometimes from around the world) working on the same project and team members working on code components they have not written, adds to the importance of readability. The use of nested loops seemed excessive in this regard. Our assumption was that it would be beneficial to explore how necessary these loop levels were and if there was a way to simplify the code in general. The rest of the elements within the "while" loop were arithmetic operations for matrices and vectors, namely multiplications, divisions and additions.

3.2.5 Improvements on the functions

Algorithm improvements were applied to some of the functions that were shown in Section 3.1. Note that the output size and type didn't change for any of the functions.

1-**Position vector**

The original position vector function shown in 3.1 was performing a loop that goes over each matrix entry (i.e. position, velocity and acceleration) in order to multiply the entries to get the position vector of nodes. The new position vector function multiplies the matrices and vectors using vectorization in Matlab.

Pseudo code:

Inputs: position vector at a node of a shell finite element on the mid-surface in the initial configuration ${}^{\beta}_{0}\overline{y_a}$, element interpolation matrix $N_a(r,s)$, nodal fiber length in initial configuration ${}^{\beta}_{0}h_a$ and unit vector ${}^{\beta}_{0}\hat{Y}_a$

Output: Position vector of nodes ${}^{\beta}y(r,s,t)$

a. ${}^{\beta}y(r,s,t) = N_a(r,s) \times {}^{\beta}_{0}\overline{y_a} + N_a(r,s) \times {}^{\beta}_{0}h_a \times {}^{\beta}_{0}\hat{Y}_a$

2- **Cauchy stress tensor**

The original implementation looped over the three input matrices to calculate the matrix of Cauchy stress tensor and looped over vector elements to do the division. In the new implementation, the matrix and vector operations are done directly avoiding loops.

Pseudo code:

Input: mass densitiesρ, partial derivatives of the strain energy function $\partial {}^{\tau}_{0}W$, deformation characteristic matrix, ${}^{\tau}_{0}E^{l}_{ij}$and the components of the transformation gradient tensor ${}^{\tau}_{0}F^{l}_{si}$, ${}^{\tau}_{0}F^{l}_{si}$.

Output: matrix of the local components of the Cauchy stress tensor ${}_{\tau}\sigma^{l}_{sr}$

a. ${}_{\tau}\sigma^{l}_{sr} = \frac{{}^{\tau}\rho}{{}^{0}\rho} \times {}^{\tau}_{0}F^{l}_{si} \times \frac{\partial {}^{\tau}_{0}W}{\partial {}^{\tau}_{0}E^{l}_{ij}} \times {}^{\tau}_{0}F^{l}_{rj}$

3- **Green strain tensor**

The original implementation looped over transformation gradient tensor matrix to perform multiplication and input data, while the new implementation does that directly with matrix operations.

Pseudo code:

Input: components of the transformation gradient tensor${}^{\tau}_{0}F^{l}_{ki}$, ${}^{\tau}_{0}F^{l}_{kj}$, Kronecker function matrix δ_{ij}

Output: Cartesian components of the Green strain tensor ${}^{\tau}_{0}E^{l}_{ij}$

a. $^{\tau}_{0}E^{l}_{ij} = 1/2 \times (^{\tau}_{0}F^{l}_{ki} * ^{\tau}_{0}F^{l}_{kj} - \delta_{ij})$

4- **Force vector matrix**

The original implementation looped over the input matrices twice in a separate manner to perform multiplication while the new implementation uses direct matrix operations instead.

Pseudo code:

Input: linear displacement transformation matrix from (2.11) $[^{\tau}_{\tau}B_L]^T$ and the components of the Cauchy stress tensor from (2.6) using the Voigt notation$\{\ _{\tau}\sigma^{l}\ \}$, and determinant of Jacobian matrix. J^0

Output: internal force matrix according to (2.5) $\{^{\tau}_{\tau}F\}$

a. $\left[^{\tau}_{\tau}B_{Lij}\right]^T\ \{\ _{\tau}\sigma^{l}_{i}\} \times J^0$

5- **Stiffness matrix**

The original implementation function looped over all the input matrices to get the stiffness matrix while the new implementation used matrix operations to calculate the results.

Pseudo code:

Input: Strain energy gradient matrix $\partial^2\,^{\tau}_{0}W$, linear displacement transformation matrix from (2.11) $\left[^{\tau}_{\tau}B^{l}_{L}\right]$, the components of the Green strain tensor $^{\tau}_{0}E^{l}_{ij}\partial, ^{\tau}_{0}E^{l}_{rs}$ and of the transformation gradient $^{\tau}_{0}F^{l}_{pr}, ^{\tau}_{0}F^{l}_{qs}, ^{\tau}_{0}F^{l}_{mi}, ^{\tau}_{0}F^{l}_{nj}$, and the determinant of the Jacobian matrix J, mass densities $^{\tau}\rho,\ ^{0}\rho$.

Output: stiffness matrix according to (2.14) and (2.15) $[^{\tau}_{\tau}K_L]$

a. $_{\tau}C^{l}_{mnpq} = \frac{^{\tau}\rho}{^{0}\rho}\, ^{\tau}_{0}F^{l}_{pr} \times\ ^{\tau}_{0}F^{l}_{qs} \frac{\partial^2\, ^{\tau}_{0}W}{\partial\, ^{\tau}_{0}E^{l}_{ij}\partial\, ^{\tau}_{0}E^{l}_{rs}}\, ^{\tau}_{0}F^{l}_{mi} \times\ ^{\tau}_{0}F^{l}_{nj}$

b. $[^{\tau}_{\tau}K_L] = \left[^{\tau}_{\tau}B^{l}_{L}\right]\left[\ _{\tau}C^{l}\right]\left[^{\tau}_{\tau}B^{l}_{L}\right]J$

6- **Mass matrix**

In the original implementation, the interpolation matrix was transformed inside the function while in the new implementation; it is transformed outside the function and given as an input to the function. The original implementation function was looping over all the input matrices for multiplication while the new implementation uses matrix operations.

Pseudo code:

Input: Gauss point matrix, element interpolation matrix $[\ ^{\tau}N]_a$, element interpolation matrix transpose $[\ ^{\tau}N]^T_a$ mass density $^{\tau}\rho$, and Jacobian determinant J

Output: mass matrix according to (2.16) $\left[\,{}^{\tau}M^{consistent}\right]_a$

a. $\left[\,{}^{\tau}M^{consistent}\right]_a = [\,{}^{\tau}N]_a^T[\,{}^{\tau}N]_a \times {}^{\tau}\rho \times J$

The run time was significantly reduced after the algorithm improvements, as shown in Tables 6 and 7.

Table 6: Run time of the improved functions according to Matlab profiler.

	Run time (h)	Percentage
Position vector	0.67	2.9%
CStress	1.18	5.2%
GStrain	1.43	6.2%
PlStress	0.52	2.3%
ForceV	1.8	7.9%
StiffMtx	1.34	5.9%
MassMtx	1.09	4.8%
TScritical	1.67	7.3%
Assemblage	3.96	17.36%
Solver	22.8	100%

Table 7: Comparison between the original code run time for the functions and the new code after algorithm modification.

	Original run time	New run time	Reduction percentage
Position vector	2.03	0.67	67%
CStress	3.28	1.18	63.9%
GStrain	3.93	1.43	63.6%
PlStress	1.97	0.52	73.6%
ForceV	2.61	1.8	31.1%
StiffMtx	3.49	1.34	61.7%
MassMtx	2.25	1.09	51.6%
TScritical	3.06	1.67	45.5%
Assemblage	5.76	3.96	31.25%
Solver	46.93	22.8	51.4%

3.3 OPTIMIZING THE CODE

Once an algorithm is correct (meaning that it generates the expected output given a specific input), the program that implements it can be optimized with respect to its environment. Since the initial implementation was in MATLAB, the optimization was done in MATLAB. The optimization was carried out in three stages as detailed below.

3.3.1 Initial runs and testing methods

Initial runs and tests were necessary to determine the parts which could be improved, as well as to provide benchmark values to compare the program before and after improvements.

Table 8*: Running time before optimization.*

	32 GB-RAM	4 GB-RAM
Preprocessor	35 seconds	2.86 min
Solver	46.93 hours	352.65 hours
Post processor	16.47 min	3.21 hours

The initial runs indicated that there was a lot of room for improvement for the solver, a good likelihood of improvement for the post processor, and a slim chance of any improvement for the preprocessor.

A function called "time ETA" was developed to test the running time of each line. The data extracting command lines had the highest running time in the preprocessor, whereas in the post processor, it was exporting the data to the file and arranging them. In a way, this was expected as file input/output operations can be costly in time. The matrix operations and the nested loops had the most running time in the solver.

3.3.2 Vectorization

MATLAB is optimized for vectors and matrices operations, and the process of changing the loops and scalar-oriented code to achieve this is called vectorization. It increases the readability of the code, decreases the possibility of errors and makes the code run faster. Matlab was designed to perform matrix and vector operations with high efficiency. Therefore, any operation that can be converted into matrix operation can expect an increase in performance. "*The main constructs that are most amenable to matrix operations are loops. The basic distinction is that vectorization results in one function call in total, as opposed to one per iteration, thus greatly reducing the overhead"* (Mathworks, 2018).

Simply put, if a built-in function can be applied to an array, vectorization is much faster than a loop approach. However, when large temporary arrays are required, the benefits of vectorization can be outweighed by the expensive allocation of memory if the processor cache is exceeded.

Many of the matrix operations such as multiplication, division and addition were done using loops in the original code. This was avoided in the new code and all matrix operations were vectorized when possible.

Many of the loops and the nested loops in the original code were vectorized in the new code as well. For example, in the original code:

3.3.3 Other refactoring improvements

Other improvements were applied to the original code to reduce running time. For instance, initialization of vectors was designed to be flexible with respect to the number of entries at every iteration, instead of enforcing a fixed number of elements and then initializing that number every single time it expands.

Some variables were removed, because they were not used after being calculated. In addition, some definitions of constants and variables were made to facilitate changing their values later.

3.4 MIGRATING THE CODE TO C++

Since the program was intended for future web applications, the software had to be migrated to an open-source language that allows users to customize it and add to it. C++ is a widely used programming language in FEA calculations and programs, as discussed in Section 2.3.4, and we chose it for its speed.

3.4.1 Implementation

We used Visual Studio 2017 with optimization compiler option –O. C++ has many FEA libraries that are already established and tested for good performance. Some of these libraries are Vega FEM ("Vega FEM Library", 2018), getFEM++ ("GetFEM++ “, 2018), libmesh (libMesh team, 2018) and MFEM ("MFEM", 2018). All these libraries have implemented the basic FEA equations and some of them took a further step and applied properties for some materials and the following table summarizes the abilities of each library that we were interested in.

Table 9: Comparison between available C++ libraries for FEA.

	Anisotropic hyperelasticy	Incompressible materials	CB shell element
Vega FEM	No	Yes	No
getFEM++	No	Yes	No
Libmesh	No	Yes	No
MFEM	No	Yes	No

From Table 9, it can be seen that all the libraries offer the similar functionality; hence we chose to move forward with MFEM library. However, some functions needed to be developed and added to the existing library to encompass the full FEA implementation of the new CB shell element, as described in Equations (2.7-2.12).

3.4.2 Migration of the code

The solver represented the bulk of the computation time. In our example in Table 8 the solver took 46 hours compared to just 16 minutes for the post processor. Clearly, the priority lay with improving the solver. Therefore, we focused on migrating the preprocessor and the solver to C++. Improving how the solution was visualized could be done by improving the post processor, which had a lower priority, and may be addressed in future work.

There were two steps to the process; namely, migrating the code and completing the equations missing in the FEA library, as discussed next.

3.4.2.1 Migrating the preprocessor

The input of the preprocessor, as explained in Section 2.5.1, was a text file with the model specifications such as boundary conditions, material constants, external loading and other needed constants and variables. Migrating the preprocessor to C++ consisted of initializing variables, constants, counters and matrices along with dynamic arrays and vectors, in a way similar to what was done in the new MATLAB preprocessor, with the difference that the picture output of the initial geometry of the structure to be simulated was not implemented in C++ because of the insignificance in the running time of the preprocessor. The MFEM library "mfem.hpp" was included in the implementation to use the "mesh" function. The "mesh"

function can read the mesh from a text file and use the dimensions directly instead of defining variables for them by using the function "Dimension ()" (Kolev et al, 2010). In addition we read the used the rest variables in text file (76 variables) were utilized in the implementation.

3.4.2.2 Migrating the solver

The solver migration involved moving the code to the C++ language and implementing the missing equations from the MFEM library. The MFEM library implements many common FEA functionalities such as functions for discretization, connectivity tables, stiffness matrices, position, displacement and force vector calculations and stress and strain calculations. All of these functions are implemented in a classical fashion (Dobrev et al., 2018) which means that all the equations for shell elements are based on Cartesian and lamina coordinates, but not on fiber coordinates. Therefore, all the functions in our implementation that need fiber coordinates were either coded from scratch or modified to include fiber coordinates.

Functions 2, 4-7, 9, 10 & 11 in Section 3.1 were implemented using existing functions in MFEM because fiber coordinates had no impact on the underlying equations. Some functions included classical formulas and only fiber coordinates formulas needed to be added, such as in functions 2 and 4.

The remaining functions 1, 3 and 8 needed to be coded from scratch. These functions were written in a function-based implementation according to the algorithm shown in Section 3.2.5. Although the assemblage matrix function existed in MFEM library, its modification to include fiber coordinates was more complex than rewriting it from scratch; therefore, the latter approach was followed.

The solver's algorithm implemented in C++ is the one shown in Section 3.2.3.2. The C++ solver implementation used the same input as the MATLAB version, but the output of the C++ solver created a text file in addition to the output of the MATLAB solver, and this file contained the resultant matrices. The text file is used as an input for the post processor since we have to move from a program to another and the variables need to be identified.

The breakdown of the solution "while" loop in C++ is the same breakdown implemented in Matlab. It has the five modules mentioned in Section 3.2.4 namely:

a. Nodes setup
b. Fiber coordinates setup
c. Calculations
d. Fiber length update
e. Assemblage matrix

The code to saves the data acquired in a text file which was not part of the Matlab implementation since the solver and post processor are in the same platform.

A subclass "SHELL" was added in "MFEM_GEOM" library of type Geometry (class Geometry has point, segment, prism, tetrahedron, triangle, square and cube types). It was added to characterize the 9-nodded new shell element. The specifications of the shell element namely the number of edges, the integration points, degrees of freedom etc. as in Section 2.2 were implemented according to Momenan description (Momenan, 2016).

3.4.2.3 Algorithm for MFEM current and novel functions

The main functions we considered in C++ are the same main function in Matlab mentioned in 3.1. The algorithm of these functions will be demonstrated in this section and all of the MFEM functions mentioned in the following section are from (Kolev et al, 2010). For convenience the functions are divided into three categories, the existing functions, the modified functions and the programmed functions (novel). All of these features were written as independent function to increase the code modularity (i.e. each algorithm was written as a function in a library bundle under the name "Nshell.hpp"). In order to understand MFEM library better, a few classes will be introduced first.

MFEM_ARRAY is an array class that MFEM library adds to the original C++ array class some functions for 2D, 3D and block operations to facilitate the matrix operations performed for FEA analysis.

MFEM_TABLE is a dedicated class to construct the connectivity table. The class has an integer entry to specify the number of elements and a function connect() to connect the elements and the nodes. It also helps adding and removing rows or columns and adding and removing connections.

MFEM_TIC_TOC is a timer class with functions that when set on keeps track of the run time. "Tic()" is for starting timing and "Toc()" is to stop. This function is used to time the total running time of the solver.

MFEM_BLOCKMATRIX is a 2D array class that takes as an input an array which is in the library implementation is an integer array but we changed it in our implementation to a double type array. This class has some handy functions that helped in our implementation. EliminateRowCol is a function within this class that takes as an input the MFEM_BLOCKMATRIX with specific row and col to be removed and re-arrange the block accordingly. The chosen row or column can be a zero row or column which can later reduce the number of variables under operation. It also has Inverse(), Transpose and Mult() for matrix inverse, transpose and multiplication respectively. The class becomes handy when we are dealing with larger matrices with scarce zero entries such as Equation 2.11.

MFEM_BLOCKOPERATOR is a library with more MFEM_BLOCKMATRIX operations. IsZeroBlock() and SetDiagonalBlock() are two functions that were used in our implementation. The first function checks if the whole matrix is a zero-entry matrix and if a condition to check rows or columns of zeros is entered it checks for it. As for the second function it helps in setting up some tensors and modulus that have only diagonal matrices that will undergo operations and switch in order of entries. There is also MFEM_BLOCKVECTOR which does all the above but for vectors instead of matrices.

MFEM_DTENSOR is a tensor creator class that takes in two integers that represent the number of rows and columns in the tensor matrix, the data type name in our case it is Scalar since Scalar is a double data type and that's what our entries are and the entries to be stored in the tensor. The class has many operating functions that are useful to our implementation. Mult() and Inverse() are functions that perform multiplication and inverse of tensors that was used in the implementation. It also allows matrix-tensor operations since it is very similar and convenient.

MFEM_ELEMENT is a class that identifies the element. As mentioned earlier in 3.4.2.2 the Shell element specifications were added to the MFEM_GEOM which is used as a type for the MFEM_ELEMENT class. The class uses all of the functions of MFEM_GEOM. We identify elements in terms of MFEM_ELEMENT because most of the functions and operators in the MFEM library take either MFEM_ELEMENT of an array of them as an input.

MFEM_MATRIX is a matrix class with all the general matrix operations of multiplication, inverse, transpose and print. The extra features in the MFEM_MATRIX class are the features of a sparse matrix and diagonal policy of keeping of zeroing (make all entries zero) of the diagonal when a row or a column is eliminated. There's also MFEM_VECTOR which is a vector class with all vector operations and a function to check the number of elements inside the vector.

MFEM_COMMUNICATION is a class with functions that facilitate the construction of relations and constraints among a group of nodes. It has GroupTopology(),MyRank(),NGroups() and Create() for all the previous three are functions within the class. The functions help arranging the master nodes and slave nodes relationship along with connecting nodes between two elements if needed. It also helps applying certain limitations to certain groups in our case material properties.

TCoefficient is a class that defines the coefficient and where to apply them. It has five Boolean arguments to mark if the coefficient is a constant, uses coordinates, uses Jacobian, uses attributes or element indices and one integer entry to identify weather this coefficient is for a scalar, a vector or a matrix. The class is a template class that is used in various functions in the implementation we did. It was used mainly with constant coefficients in the calculations. Some of the functions used were Sub(), Add(), Mult(), Div ().

Note that all the functions output's sizes are the same, what have changed is the MFEM defined data type that holds it (i.e. MFEM_MATRIX instead of 2D Array etc.) which is similar to what the Matlab code has.

3.4.2.3.1 Existing functions

Functions 5-7 and 9-11 in Section 3.1 were implemented using existing functions in MFEM because fiber coordinates had no impact on the underlying equations.

Green strain tensor

Function GStrain implementation identifies Kronecker function as a TCoefficient used as a constant. It also defines the components of the transformation gradient tensor as MFEM_MATRIX. The output which is the Cartesian components of the Green strain tensor

is identified as a MFEM_DTENSOR. The function uses matrix multiplication and subtracting coefficient functions.

Pseudo code:

Input: components of the transformation gradient tensor ${}_{0}^{\tau}F_{ki}^{l}$, ${}_{0}^{\tau}F_{kj}^{l}$, Kronecker function matrix δ_{ij}

Output: Cartesian components of the Green strain tensor ${}_{0}^{\tau}E_{ij}^{l}$

a. ${}_{0}^{\tau}E_{ij}^{l} = 1/2 \times ({}_{0}^{\tau}F_{ki}^{l} \times {}_{0}^{\tau}F_{kj}^{l} - \delta_{ij})$

Plane stress state

PlStress was coded using the function SetPreconditioner() in the solvers.hpp library. The function activates the solver only if the output of the condition is true. The function takes in a Boolean variable which is in our case the output of PlStress function. ${}_{0}^{\tau}E_{ij}^{l}, \delta_{ij}$ were set as MFEM_DTENSOR and MFEM_MATRIX respectively.

Pseudo code:

Input: components of the Green strain tensor ${}_{0}^{\tau}E_{ij}^{l}, \delta_{ij}$

Output: Boolean to confirm plane stress values

a. Det $(2 \times {}_{0}^{\tau}E_{ij}^{l} + \delta_{ij}) - 1$
b. If the result = 0
c. Return true
d. Else
e. Return false

Force vector matrix

Function ForceV defines the linear displacement transformation matrix as MFEM_MATRIX, the components of the Cauchy stress tensor as MFEM_DTENSOR, and the determinant of the Jacobian matrix as a Tcoefficient of type constant. Its outputs MFEM_MATRIX containing the internal forces vectors for all the elements.

Pseudo code:

Input: linear displacement transformation matrix from (2.11) $[{}_{\tau}^{\tau}B_{L}]^{T}$ and the components of the Cauchy stress tensor from (2.6) using the Voigt notation $\{ {}_{\tau}\sigma^{l} \}$, and determinant of Jacobian matrix. J^{0}

Output: internal force matrix according to (2.5) $\{{}_{\tau}^{\tau}F\}$

a. $\left[{}_{\tau}^{\tau}B_{Lij}\right]^T \ \{ {}_{\tau}\sigma_i^l\} \times J^0$

Mass matrix

This function is implemented in MFEM solvers library in the same way we implemented in Matlab with disregarding the interpolation matrix transpose as an entry. The only difference is the input types which in this case will be MFEM_MATRIX for Gauss point matrix, element interpolation matrix $[\ {}^{\tau}N]_a$ and Tcoefficient for mass density ${}^{\tau}\rho$, and Jacobian determinant$J$. The function's output is MFEM_MATRIX with mass matrix entries.

Pseudo code:

Input: Gauss point matrix, element interpolation matrix$[\ {}^{\tau}N]_a$, element interpolation matrix transpose $[\ {}^{\tau}N]_a^T$ mass density ${}^{\tau}\rho$, and Jacobian determinant J

Output: mass matrix according to (2.16) $\left[\ {}^{\tau}M^{consistent}\right]_a$

a. $\left[\ {}^{\tau}M^{consistent}\right]_a = [\ {}^{\tau}N]_a^T[\ {}^{\tau}N]_a \times \ {}^{\tau}\rho \times J$

Discretization

Function Discretization is implemented using function SetMeshGen() in mesh.hpp library with two inputs one to set the condition (i.e. addition, subtraction etc.) and the other one is the input of the desired values to operate on (e.g. constant, array, matrix etc.). In our case the condition is division and the inputs are displacement vectors $\{{}^{\tau-\Delta\tau}_{0}U\}, \{{}^{\tau}_{0}U\}, \{{}^{\tau+\Delta\tau}_{0}U\}$ and time step $\Delta\tau^0$. The first three are implemented using MFEM_ VECTOR and the last one is a Tcoefficient constant type. The output is MFEM_VECTOR.

Pseudo code:

Input: displacement vectors $\{{}^{\tau-\Delta\tau}_{0}U\}, \{{}^{\tau}_{0}U\}, \{{}^{\tau+\Delta\tau}_{0}U\}$ and time step $\Delta\tau^0$

Output: acceleration vector according to (2.2) $\{{}^{\tau}_{0}\ddot{U}\}$

3.4.2.3.2 Modified functions

Some functions included classical formulas and only fiber coordinates formulas needed to be added, such as in functions 2 and 4.

Position vector

The mesh.hpp library SetPosition() function implements the first half of the position vector function that we have (i.e. $N_a(r,s) * {}^{\beta}_{0}\overline{y_a}$) so we had to modify the function and add to it the second part ($N_a(r,s) \times {}^{\beta}_{0}h_a \times {}^{\beta}_{0}\hat{Y}_a$). In order to do that we had to change the inputs in

SetPosition() function from one MFEM_MATRIX $N_a(r,s)$ and one MFEM_VECTOR ${}^{\beta}_{0}\overline{y_a}$ to include another MFEM_VECTOR ${}^{\beta}_{0}h_a$ and one Tcoefficient ${}^{\beta}_{0}\hat{Y}_a$. The function was changed and moved to Nshell.hpp library.

Pseudo code:

Inputs: position vector at a node of a shell finite element on the mid-surface in the initial configuration ${}^{\beta}_{0}\overline{y_a}$, element interpolation matrix $N_a(r,s)$, nodal fiber length in initial configuration${}^{\beta}_{0}h_a$ and unit vector ${}^{\beta}_{0}\hat{Y}_a$

Output: Position vector of nodes ${}^{\beta}y(r,s,t)$

a. ${}^{\beta}y(r,s,t) = N_a(r,s) \times {}^{\beta}_{0}\overline{y_a} + N_a(r,s) \times {}^{\beta}_{0}h_a \times {}^{\beta}_{0}\hat{Y}_a$

Cauchy stress tensor

In solvers.hpp library, there is a function for solving stress values called STRSolver(). The function takes in the current configurations, the partial derivatives of the strain energy function describing the material property with respect to the Cartesian components of the Green strains tensor in the form of MFEM_MATRIX and MFEM_DTENSOR (i.e. ${}^{\tau}_{0}F^{l}_{si} \times \partial{}^{\tau}_{0}W$ and $\partial{}^{\tau}_{0}E^{l}_{ij}$). The function was modified by implementing the equation below and adding the missing inputs (i.e. ${}^{\tau}_{0}F^{l}_{si}$, ${}^{0}\rho$ and ${}^{\tau}\rho$) in the form of MFEM_MATRIX and Tcoefficient.

Pseudo code:

Input: mass densities ρ, partial derivatives of the strain energy function $\partial{}^{\tau}_{0}W$, deformation characteristic matrix, ${}^{\tau}_{0}E^{l}_{ij}$and the components of the transformation gradient tensor ${}^{\tau}_{0}F^{l}_{si}$, ${}^{\tau}_{0}F^{l}_{si}$.

Output: matrix of the local components of the Cauchy stress tensor ${}_{\tau}\sigma^{l}_{sr}$

a. ${}_{\tau}\sigma^{l}_{sr} = \frac{{}^{\tau}\rho}{{}^{0}\rho} \times {}^{\tau}_{0}F^{l}_{si} \times \frac{\partial{}^{\tau}_{0}W}{\partial{}^{\tau}_{0}E^{l}_{ij}} \times {}^{\tau}_{0}F^{l}_{rj}$

3.4.2.3.3 Programmed functions (novel)

The remaining functions updated Lagrangian formulation using explicit time integration, transformation matrix and stiffness matrix needed to be coded from scratch. Each algorithm was written as a function in a library bundle under the name "Nshell.hpp" according to the algorithm shown in Section 3.2.5 using MFEM_MATRIX, MFEM_VECTOR and MFEM_DTENSOR classes for convenience of operations. The names of the functions are the same names in 3.2.5.

Although the assemblage matrix function existed in MFEM library in class fe.hpp (under the name Project()), its modification to include fiber coordinates was more complex than rewriting it from scratch; therefore, the latter approach was followed. The new assemblage matrix implementation reorders the resulting position entries in Cartesian, lamina and fiber coordinates in a matrix of the elements number size with a vector of 9 of double type entry indicating the value of position in in the three axes of each coordinate under the function name shellProject(). The function takes in the force matrix, connectivity table, mass density matrix and transformation matrix to produce the assemblage matrix as an output.

4 RESULTS AND DISCUSSION

4.1 IMPLEMENTATION OF THE PROPOSED ALGORITHM IN MATLAB

Thanks to proper algorithm development, improvements to the code were noticeable. The original code was intended to test the concepts of the theoretical work for proof of concept purposes. As a result, it was sub-optimally organized and running time was long. The proposed algorithm, on the other hand, used Matlab to produce a unique and fully functional program that saved time and memory.

The division of the algorithm into three parts helped to determine where the improvements could be made, and what parts of the code took large amounts of running time during the process. The solver turned out to be the most time-consuming part. Some of the coding standards and preferred practices that had been ignored in the original code were enforced in the new proposed algorithm.

One of the main contributions of the algorithm regarding the improvement of the running time was achieved by reducing the number of repeated and nested loops.

Table 10: Comparison of running time on benchmark problem between the MATLAB original code (OC) and the MATLAB code after applying the algorithm (CAA).

	32-GB RAM		4-GB RAM	
Algorihm part	OC	CAA	OC	CAA
Preprocessor	35 seconds	35 Seconds	2.86 min	2.23min
Solver	46.93 hours	22.8 hours	352.65 hours	210.36 hours
Post processor	16.47 min	11.84 min	3.21 hours	2.12 hours

Table 10 shows that the speed of the solver in the benchmark problem considered was improved by up to 50% over the original code speed. The running time of the original code was 46.93 hours on a 32 GB RAM machine while it was 22.8 hours on the same machine with the algorithm applied to the new code. There was no difference between the speed of the preprocessor of the original and the new code on the 32 GB RAM machine. However, the

running time was reduced on the 4 GB RAM machine by 22%. The post processor running time was reduced by 28% on the 32 GB-RAM machine and 34% on the 4 GB-RAM machine.

The algorithm of the preprocessor was barely modified compared to its original version. This explains why the new algorithm was not significantly faster than the original on the 32 GB-RAM machines, although taking some of the matrix definitions out of the preprocessor apparently reduced the running time slightly on the 4 GB-RAM.

The solver was affected the most by the new algorithm, with a 50% reduction time on the 32 GB-RAM machines. Such reduction was achieved by eliminating the five for loops that looped over 3,375,000 elements at every time step. Additional improvement was achieved by removing the plotting function from the while loop to avoid spending time on plotting in the middle of the iterations.

The running time of the post processor was reduced by the new algorithm by 28% on the 32 GB-RAM machines. The elimination of five large matrices led to fewer loops for the post processor to enter the data into the output text file. It also helped the plotting function to loop only once over one large matrix, instead of multiple times.

4.2 THE OPTIMIZATION

The optimization of the code was only possible after organizing the code according to the new algorithm discussed in the previous section. It had a positive impact on the running time as shown in Table 11. By vectorizing most of the matrix operations, the loops that featured matrix additions and multiplications carried out manually were replaced with direct matrix additions and multiplications. Vectorizing arrays and loops along with other minor changes mentioned in Section 3.3.2 reduced the running time by 90% of the original running time.

Table 11: Comparison of running time on benchmark problem between the MATLAB code after applying the algorithm (CAA), and the MATLAB code after applying the algorithm and optimization (COA).

	32-GB RAM		4-GB RAM	
Algorithm part	CAA	COA	CAA	COA
Preprocessor	35 seconds	34 seconds	2.23 min	1.96 min
Solver	22.8 hours	59 min	210.36 hours	5.14 hours
Post processor	11.84 min	2.63 min	2.12 hours	30.06 min

The preprocessor's running time was reduced by one second on the 32 GB-RAM machines after the optimization, while it was reduced by 12% on the 4 GB-RAM machines. On the other hand, the running time of the solver was reduced by 95% of the original running time on the 32 GB-RAM machines, and 97.5% on the 4 GB-RAM machines. The post processor's running time was reduced by 78% on the 32GB-RAM machine, and by 75% on the 4 GB-RAM machines.

The benefits of optimization on the preprocessor were limited due to constants' definitions and matrices and vectors initialization to zero. On the other hand, the running time of the solver was significantly affected by the optimization. The non-optimized solver featured many matrix and vector operations which had a long run time during execution. The vectorization of these operations had a major positive impact on the code performance. The loops and their number were reduced throughout the code. Eliminating some redundant variables and loops resulted in time reduction from 22.8 hours to 59 minutes. Those steps significantly affected the memory performance. In order to find out the exact impact on the memory allocation, a memory profiling analysis will be part of the future work as we improve the two implementations.

The post processor benefited from the optimization because majority of the data that was exported to the output text file were matrices and vectors' elements. After optimization, most of these matrices and vectors were transferred smoothly and in a faster fashion from MATLAB out to the text file.

4.3 THE MIGRATED CODE

As shown in Table 12, there was a notable difference in favour of C++ compared to MATLAB regarding the running time of the solver, indicating that C++ calculations were done faster.

Table 12: Comparison of running time on benchmark problem between the MATLAB code after applying the algorithm and optimization (COA) and the C++ code (CC).

	32-GB RAM		**4-GB RAM**	
	COA	CC	COA	CC
Preprocessor	35 seconds	39 seconds	1.96 min	2.04 min
Solver	59 min	43 min	5.14 hours	4.36 hours

The C++ preprocessor was slower by 4 seconds on the 32 GB-RAM machines, and slower by 19 seconds on the 4 GB-RAM machines, but these differences did not have a significant impact on the overall performance of the simulation. The C++ solver was faster by 27% while the C++ post processor had the same running time. On the 32 GB-RAM machines, the total running time of the MATLAB code was 62.33 min and that of the C++ code was 46.33 min. The C++ running time was 25% less than the MATLAB code, which was significant.

4.4 OVERALL PERFORMANCE

The optimized code tested on the benchmark problem ended up running 20 times faster than the original code as shown in Fig. 9. The optimized code followed a clear algorithm as shown fully in Section 3.2.3. The C++ code was faster than the MATLAB optimized code.

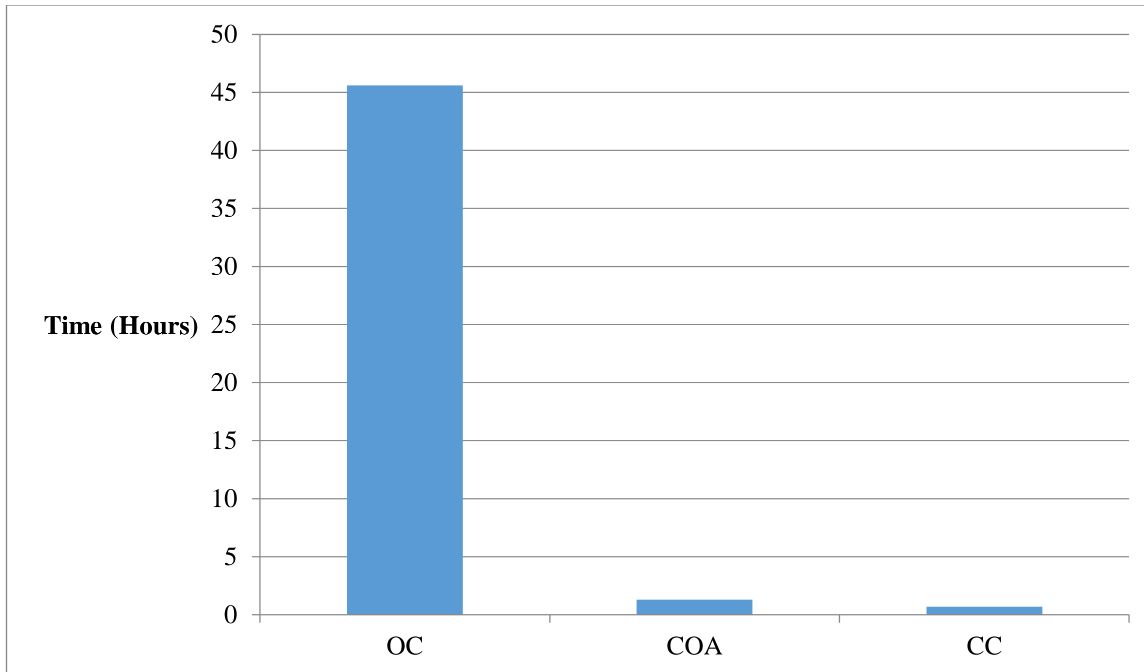

Figure 9: Comparison of running time on benchmark problem between MATLAB original code (OC), the MATLAB code after applying the algorithm and optimization (COA) and the C++ code (CC).

To further probe the magnitude of the improvements to the implementation of Momenan's shell finite element, the benchmark problem was modified to include more elements. Table 13 shows the results of the running time of the solver using the MATLAB original code and the MATLAB code after applying the algorithm and optimization. The benchmark problem tests the same initial geometry which were discretized with 50, 100 and 250 shell elements. The results exhibited discrepancy between the 50, 100 and 250 elements in the new implementation.

Table 13: Comparison between the running time of the solver for different numbers of shell elements in the benchmark problem using the MATLAB original code (OC) and the MATLAB code after applying the algorithm and optimization (COA), on the 32-GB machine.

	OC	COA
50 elements	46.93 hours	59 min
100 elements	98.6 hours	8.96 hours
250 elements	242.47 hours	13.69 hours

As illustrated in Fig. 10, there was a direct correlation between the number of elements and the process time.

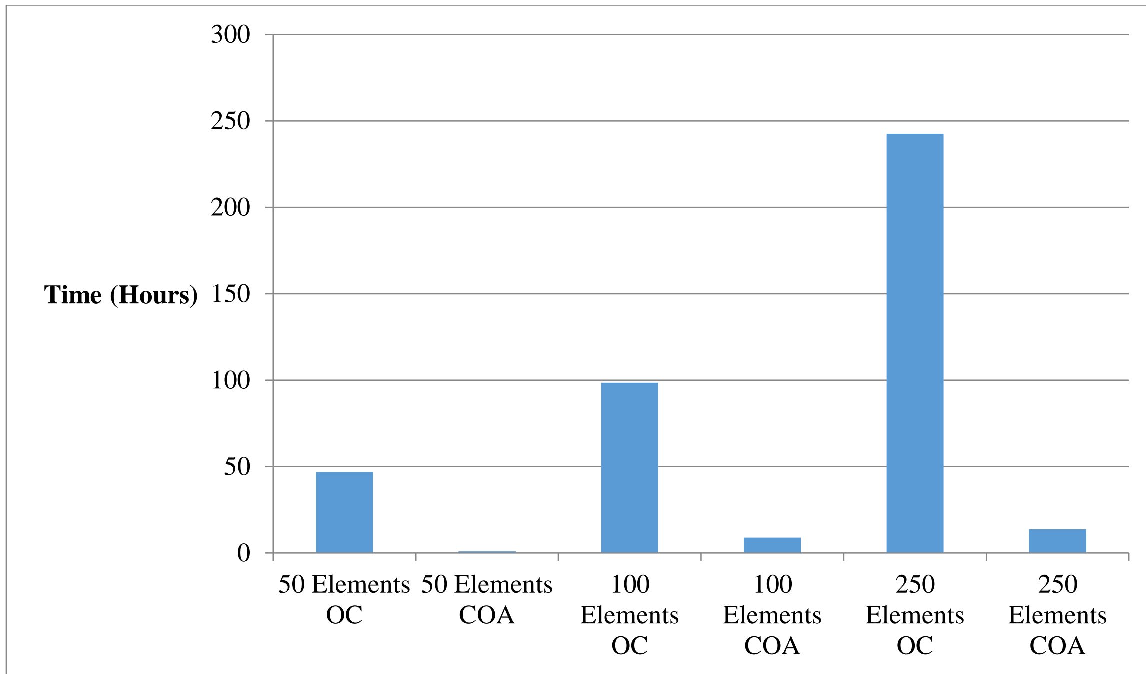

Figure 10: Comparison between the number of elements, and running time (in hours), for OC and COA cases.

Furthermore, simulations were conducted to identify the starting point at which the discrepancy occurs in the new implementation. Table 14 shows the results of more simulations using the new code with 50, 60, 70, 80, 90 and 100 elements respectively.

Table 14: Simulations with 50, 60, 70, 80, 90 and 100 elements using the new code.

	COA	Matrix size and memory
50 elements	59 min	2500 input, 20 KB
60 elements	1.26 hours	3600 input, 28.8 KB
70 elements	1.85 hours	4900 input, 39.2 KB
80 elements	2.53 hours	6400 input, 51.2 KB
90 elements	4.97 hours	8100 input, 64.8 KB
100 elements	8.96 hours	10000 input, 80 KB

As can be seen from Figure 11, the running time varied linearly with the number of elements until the 90 elements mark where it started to increase exponentially. The run time for 90 elements was double that for 80 elements, and was half that for 100 elements. Given that

1- The solver needs 2000 iterations and produces, at each time step, an assemblage matrix and other matrices of similar size scale.

2- The data type for input variables is double (i.e. 64bits = 8 bytes),

in the case of 50 elements, this gives 50×50 assemblage matrix size (i.e. 2500 inputs of 8 bytes each), for a total of 20 KB. If the matrix is not removed from memory for the duration of the iterations, this means 40MB (i.e. 20 KB × 2000) in total are used for only the assemblage matrix. If we consider the 100 elements case, using the exact same calculation, then the matrix size will be 160 MB over the 200 iterations. This example shows that doubling the elements number results in quadrupling the memory size. This quadratic growth explains fast memory exhaustion and the quadratic increase in the number of operations (statement). Therefore, the quadratic increase and complexity explains the gap seen in the performance of the solver from 50 elements to 100 elements to 250 elements.

Since the assemblage matrix is not the only matrix of this size that is recalculated during the 2000 iterations, it can be assumed that the performance discrepancy is caused by the drawback of vectorization mentioned in Section 3.3.2: when large temporary arrays are required, the benefits of the vectorization can be dominated by the expensive memory allocation, when it does not fit into the processor cache. This suggests that there is room for improvement for the space complexity. Typically, at any given iteration of the computations, some of the results might coincide with those at previous iterations and by default, the solver re-computes the values at these iterations even though it has already computed their values and found that they are not converged. If the calculations typically take a long run time in execution, this could make the solver run significantly longer. Exploring the Matlab-cache function and relationship, and whether the memory problem is similar to the sparse matrix vector multiplication problem (Toledo, 1997) is recommended for future work.

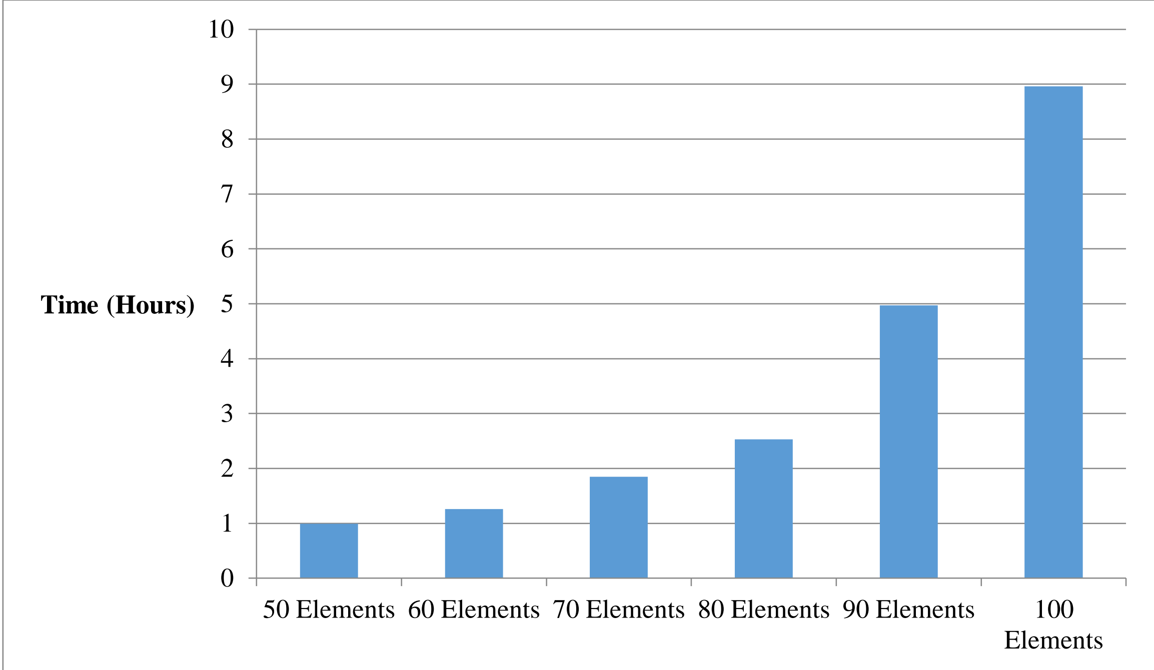

Figure 11: Comparison between the running time (in hours) for 50-100 elements

5 CONCLUSION

5.1 BRIEF SUMMARY AND MAJOR CONTRIBUTIONS

Available shell finite elements models have shortcomings such as not accounting for hyper-elastic, nearly-incompressible, and anisotropic materials, which limits their application in the context of biological soft tissue modelling. Momenan developed a new shell finite element that overcame these defects and implemented it in MATLAB for proof of concept. The script was executing in 46 hours for a benchmark problem on a 32-GB RAM machine. This running time needed to be reduced for the element to be practical and implementable in real world applications. A programming algorithm was developed to improve the running time and, once implemented, reduced it to 50% of its original value. Further optimization and vectorization were implemented, to reduce running time further to 95% of its original value. Implementation in C++ was also carried out to ensure the running time was reduced in a more efficient manner. The program finally ran 20 times faster than the original one, such that it should be possible to implement the new finite element in practical applications, although on the benchmark problem considered, we were not able to reduce the running time to values comparable to those obtained with general 3D finite elements implemented in a commercial code.

5.2 RECOMMENDATIONS FOR FUTURE WORK

Making the program more efficient is a continual effort that requires updating as allowed by technology. The code was improved and the achieved running time was drastically faster than the original one, but there is still room for improvement from theories that can be applied to reduce the running time. One lead is the improvement through making use of possible patterns (e.g. sparsity) in matrix computations. This would allow for faster computations. Further speed improvements could also be achieved using cloud-based computation. However, it is only effective if the code itself can run in a fast and efficient enough manner, which has been much improved through the present work.

Space complexity techniques and memory allocation best practices can be explored and implemented in order to benefit larger implementations. Other memory allocation techniques such as multi-tier caching or parallel computing can be explored as well. For Matlab specifically,

exploration of the relation between execution and cache is recommended. More research can be done on Matlab compilers as well to handle vectorization better (for example by using Just-In-Time compiler (Chevalier-Boisvert et al, 2010)).

Using different compilation options on C++ compiler may also affect the calculating time. As mentioned previously, the one we used was –O, but other options like –O1, -O2 and –O3 could be explored as well. Memory management can also be considered in the future work as it can affect the program run time (Novark et al, 2009).

For consistency, the post processor should also be migrated to C++ and in the long run the code would probably need to be rewritten from scratch in C++, where the functions needed would be specific to the shell element implementation, instead of being general functions that might take longer running time and more memory space.

6 REFERENCES

- ABAQUS, I. (2013). ABAQUS. Retrieved from http://129.97.46.200:2080/v6.13/ on May 21, 2018
- ADINA - Finite Element Analysis Software. (2018). Retrieved from http://www.adina.com/ on December 17th, 2018.
- Bauman, P. T., & Stogner, R. H. (2016). GRINS: a multiphysics framework based on the libmesh finite element library. *SIAM Journal on Scientific Computing*, 38(5), S78-S100.
- Cass, S. (2018). The 2018 Top Programming Languages. Retrieved from https://spectrum.ieee.org/at-work/innovation/the-2018-top-programming-languages on December 21st, 2018.
- Chevalier-Boisvert, M., Hendren, L., & Verbrugge, C. (2010, March). Optimizing MATLAB through just-in-time specialization. In *International Conference on Compiler Construction* (pp. 46-65). Springer, Berlin, Heidelberg.
- DUNE - DUNE Numerics. (2018). Retrieved from https://www.dune-project.org/ on December 19th, 2018.
- Dedner, A., Lüthi, M., Albrecht, T., & Vetter, T. (2007). Curvature guided level set registration using adaptive finite elements. In *Joint Pattern Recognition Symposium* (pp. 527-536). Springer, Berlin, Heidelberg.
- Dedner, A., Klöfkorn, R., Nolte, M., & Ohlberger, M. (2010). A generic interface for parallel and adaptive discretization schemes: abstraction principles and the DUNE-FEM module. *Computing*, 90(3-4), 165-196.
- Dobrev, V. A., Knupp, P., Kolev, T. V., & Tomov, V. Z. (2018). Towards simulation-driven optimization of high-order meshes by the target-matrix optimization paradigm. *27th International Meshing Roundtable*.
- Dubois-Pelerin, Y., & Zimmermann, T. (1993). Object-oriented finite element programming: III. An efficient implementation in C++. *Computer Methods in Applied Mechanics and Engineering*, *108*(1-2), 165-183.
- FEBiO – Software Suite. (2018). Retrieved from https://febio.org/ on December 17th, 2018.

- Fung, Y. C. (1967). Elasticity of soft tissues in simple elongation. American Journal of Physiology-Legacy Content, 213(6), 1532-1544.
- GetFEM++ Homepage — GetFEM++. (2018). Retrieved from http://getfem.org/ on December 21st, 2018.
- Gijsenbergh, P., Halbach, A., Jeong, Y., Torri, G. B., Billen, M., Demi, L., ... & Rochus, V. (2019). Characterization of polymer-based piezoelectric micromachined ultrasound transducers for short-range gesture recognition applications. *Journal of Micromechanics and Microengineering*, 29(7), 074001.
- Jehl, M., Dedner, A., Betcke, T., Aristovich, K., Klöfkorn, R., & Holder, D. (2014). A fast parallel solver for the forward problem in electrical impedance tomography. *IEEE Transactions on Biomedical Engineering*, 62(1), 126-137.
- Kolev, Tzanio, & Dobrev, Veselin. (2010, June 21). MFEM: Modular Finite Element MethodsLibrary. [Computersoftware].https://github.com/mfem/mfem.doi:10.11578/dc.20171025.1248.
- Labrosse, M. R., Lobo, K., & Beller, C. J. (2010). Structural analysis of the natural aortic valve in dynamics: from unpressurized to physiologically loaded. *Journal of biomechanics*, 43(10), 1916-1922.
- Labrosse, M. R., Beller, C. J., Boodhwani, M., Hudson, C., & Sohmer, B. (2015). Subject-specific finite-element modeling of normal aortic valve biomechanics from 3D+ t TEE images. *Medical image analysis*, 20(1), 162-172.
- libMesh team. (2018). libMesh - A C++ Finite Element Library. Retrieved from http://libmesh.github.io/ on December 21st, 2018.
- LS-DYNA. (2018). Retrieved frim http://www.lstc.com/products/ls-dyna on December 21st, 2018.
- Maas, S. A., Ellis, B. J., Ateshian, G. A., & Weiss, J. A. (2012). FEBio: finite elements for biomechanics. *Journal of biomechanical engineering*, 134(1), 011005.
- MATLAB - MathWorks. (2018). Retrieved from https://www.mathworks.com/products/matlab.html on June 7th, 2018
- MFEM - Finite Element Discretization Library. (2018). Retrieved from http://mfem.org/ on December 21st, 2018.

- MoFEM landing page. (2018). Retrieved from http://mofem.eng.gla.ac.uk/mofem/html/ on December 17th, 2018.
- Momenan B, Labrosse MR. A New Continuum-Based Thick Shell Finite Element for Soft Biological Tissues in Dynamics: Part 1-Preliminary Benchmarking Using Classic Verification Experiments. *arXiv preprint* arXiv:1801.04029, 2018a.
- Momenan B, Labrosse MR. A New Continuum-Based Thick Shell Finite Element for Soft Biological Tissues in Dynamics: Part 2-Anisotropic Hyperelasticity and Incompressibility Aspects. *arXiv preprint* arXiv:1801.04027, 2018b.
- Momenan B. Development of a thick shell finite element for soft tissue dynamics, PhD dissertation, University of Ottawa, 2016.
- Novark, G., Berger, E. D., & Zorn, B. G. (2009, June). Efficiently and precisely locating memory leaks and bloat. In *ACM Sigplan Notices* (Vol. 44, No. 6, pp. 397-407). ACM.
- Palmer, G. E., Barnhardt, M., Amar, A. J., Kirk, B., & Chen, Y. K. (2014). Coupled CFD-ablation response model simulations using the libMesh framework. In *11th AIAA/ASME Joint Thermophysics and Heat Transfer Conference* (p. 2123).
- Segal, G. (2010). *Programmers Guide.* In Sepran. Netherlands: Ingenieursbureau Sepra.
- Sparselizard finite element C++ library. (2018). Retrieved from http://www.sparselizard.org/ on December 21st, 2018.
- Toledo, S. (1997), Improving the memory-system performance of sparse-matrix vector multiplication, IBM J. Res. Dev., 41,711–725.
- Vega FEM Library. (2018). Retrieved from http://run.usc.edu/vega/ on December 21st, 2018.