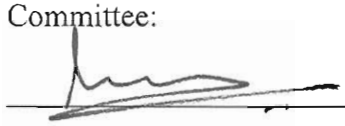
COMPUTATIONAL MEMBRANE MODEL USING FINITE ELEMENT METHOD FOR FLEXIBLE-WING-BASED MICRO AIR VEHICLES

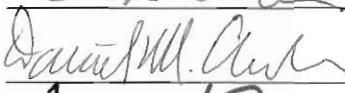
by

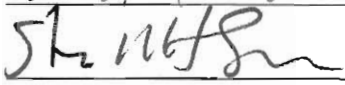
Michael Garrity
A Thesis
Submitted to the
Graduate Faculty
of
George Mason University
in Partial Fulfillment of
The Requirements for the Degree
of
Master of Science
Mathematics

Committee:

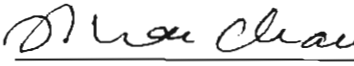

Dr. Padmanabhan Seshaiyer, Thesis Director


Dr. Tim Sauer, Committee Member


Dr. Daniel Anderson, Committee Member


Dr. Stephen Saperstone, Department Chairperson


Dr. Timothy Born, Associate
Dean for Academic and Student Affairs,
College of Science


Dr. Vikas Chandhoke, Dean,
College of Science

Date: 7/26/2011

Summer Semester 2011
George Mason University
Fairfax, VA

Computational Membrane Model Using Finite Element Method for Flexible-Wing Based Micro Air Vehicles

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science at George Mason University

By

Michael J. Garrity
Bachelor of Science
George Mason University, 2010

Director: Padmanabhan Seshaiyer, Associate Professor
Department of Mathematical Sciences

Summer Semester 2011
George Mason University
Fairfax, VA

DEDICATION

This is dedicated to my amazing girlfriend Jenny and all my family for their support.

ACKNOWLEDGEMENTS

I would first like to thank Dr. Padmanabhan Seshaiyer for his help, support, and patience. Also, I would like to extend my gratitude to other committee members for always being there to assist me on any questions regarding research or other classwork.

Secondly I would like my family for their support and advice throughout my entire education. They have always been there for me and supported me on every decision I have had to make. I feel that it is because of their constant encouragement that I have achieved my successes. Likewise, my friends have always kept me going forward and gave me the strength I needed to overcome any obstacle to get to where I am now.

Last but not least, I would like to give the biggest thanks to my girlfriend Jenny and our amazing dog, Darwin. They have been there each and every day, comforting and supporting me along the way, and I could not ask for anything more.

TABLE OF CONTENTS

Page

List of Tables……vii

List of Figures……viii

Abstract……ix

CHAPTER I - Introduction……1

CHAPTER II - Background and Methods……5

2.1 Characterization of Strain……5

2.2 Computing Strain of Isotropic Membranes……9

2.3 Description of Strain-energy Functions……11

CHAPTER III –Methodology for Membrane Solution……15

3.1 Finite Element Discretization……15

3.2 Principle of Virtual Work……17

3.2.1 External Forces……18

3.2.2 Internal Forces……19

3.2.3 Equilibrium Solution……20

CHAPTER IV – Constructing a Computational Algorithm……22

4.1 Initial Implementation……22

4.2 Preprocessor……23

4.3 Processor and Postprocessor……24

4.4 High Performance Programming……27

CHAPTER V – Numerical Results and Discussion…….....28

CHAPTER VI – Conclusion and Future Research……35

APPENDIX……38

REFERENCES……..58

LIST OF TABLES

Table Page
2.1: Constitutive Models……..……..13

5.1: Center-Node Deflection of Neo-Hookean Model...……...29

5.2: Membrane Model Comparisons between Programming Languages……..30

5.3: Center-Node Deflection of Constitutive Models…..……..31

LIST OF FIGURES

Figure Page
1.1: Successful Examples of Rigid Wing MAVs……..…..1

1.2: Representative Wing Structure of an MAV…….....3

2.1: Deformation of an Arbitrary Body……...6

2.2: Deformation of an Isotropic Membrane……..……...……....10

4.1: Main Algorithm for Membrane Model ……...……...……...23

4.2: Preprocessor Flowchart……...……....24

4.3: Processor Flowchart……..……..……26

4.4: Postprocessor Flowchart…….....26

5.1: Comparison of Center-Node Deflections at Various Pressures…….……...……..32

5.2: Membrane Deformation (Pressure = 0.25)...……..……...…....32

5.3: Membrane Deformation (Pressure = 0.75)...……....…….33

5.4: Membrane Deformation (Pressure = 1.25)……..…….……..…...33

5.5: Membrane Deformation (Pressure = 1.75)..……....……...……34

ABSTRACT

COMPUTATIONAL MEMBRANE MODEL USING FINITE ELEMENT METHOD FOR FLEXIBLE-WING BASED MICRO AIR VEHICLES

Michael Garrity, M.S.

George Mason University, 2011

Thesis Director: Dr. Padmanabhan Seshaiyer

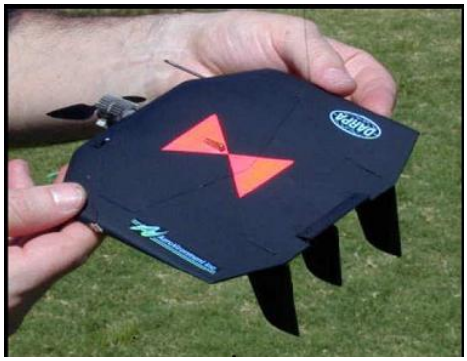
Current designs for Micro Air Vehicles (MAVs) consist of a flexible wing with rigid battens for support. This paper analyzes the development and evaluation of strain on the membrane from various constitutive models used for the flexible-wing-based MAVs. A dependable accurate prediction of its aerodynamic performance requires efficient modeling techniques. In this work, a new C++ code has been developed to simulate the computational membrane methodology described in this thesis. Our computational model of the membrane incorporates a finite element analysis that uses Principle of Virtual Work. We develop and implement a computational algorithm to understand the mechanics of a thin "rubber-like" membrane material of the wing. We also study the performance of the method with different constitutive models. The overall objective in our model is to solve for the deflection of the deformed membrane.

CHAPTER I

INTRODUCTION

With recent improvements in technology, many enhancements of Unmanned Air Vehicles have occurred. In particular, the Micro Air Vehicle (MAV) belongs to a class of miniature aircraft which have a maximal wingspan of about 15 centimeters (6 inches), small enough to fit into a hand, and can travel at relatively low speeds of less than 10 m/s. MAVs cannot simply be made by the same design as current aircrafts at a smaller size. Since an MAV is much smaller, it will have a lower Reynolds number which could lead to sudden increases in drag and loss of efficiency [12]. Therefore, these vehicles were designed in unique smaller and less expensive designs than larger UAVS.

a) The *Microstar*



b) The *Black Widow*

Figure 1.1: Successful Examples of Rigid Wing MAVs

MAVs were originally developed for military use by installing a camera for reconnaissance and surveillance. These missions were designed to search and collect data at specific locations by navigating through confined spaces. Though military missions were the primary uses of MAVs, they have also been used for other various applications. MAVs were designed for travelling through busy cities and operating inside buildings. Another suitable use for an MAV is image collecting by placing or receiving sensors at various locations, and to possibly be used as communication devices. Further developments of MAVs continue to lead to more complex tasks such as search and rescue missions, border surveillance, and climate studies.

The development of an MAV with precise functionality in all applications is still underway. Original designs used an optimized rigid wing structures even with the lack of stabilization due to its unsteady nature. Early successful designs of rigid wing designs included AeroVironment's "Black Widow" as well as the DARPA-funded MAV, "Microstar" [12], (see Figure 1.1). Recent studies have led to an MAV design using a flexible wing. A flexible membrane was first designed and developed at the University of Florida. Here, a comparison in performance of both the flexible membrane and rigid based wing structures was studied [6]. MAVs were shown to gain more stability and agility with the flexible wing structure which could lead to smoother flights. Constructed using thin membrane-based materials, flexible wings can adapt to unstable flight conditions, such as gusty winds, extending the MAVs capabilities.

A flexible wing of an MAV is constructed with a carbon fiber skeleton and thin rubber membrane material, as well as three other major components (see Figure 1.2). For

its main purpose, the vehicle has a central wing box which holds any equipment necessary for its mission. It also has number of battens attached to a leading edge spar which act as the control for the wings movement. The number of battens and their positioning on the flexible wing is what leads to the adaptive shape such that there is added stability during flight.

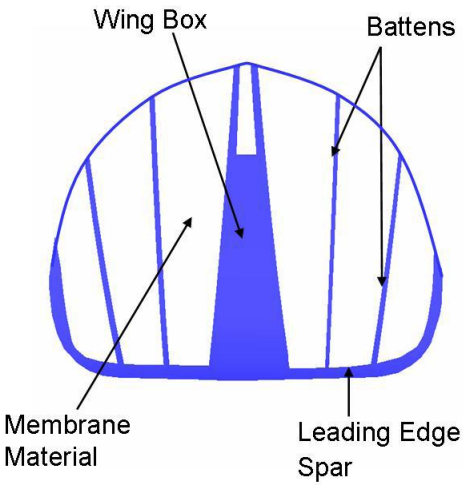


Figure 1.2: Representative Wing Structure of an MAV

The mechanics of the membrane itself provides a major importance to the overall structure of the MAV as it is a special case relating to the deformation of shells [7]. These membranes are considered to be very thin shell structures which are usually modeled to be resistant to bending. Therefore similar to biological membranes of soft tissues, these mechanics can also be applied to the deformation of the rubber membrane structures of MAVs.

In this thesis, we will analyze different constitutive models of the membrane's material of the flexible wing. We will model the membrane under multiple assumptions discussed in the following chapters. In Chapter II, we show how to characterize the strain of the membrane during deformation. In Chapter III, we discuss the incorporation of the finite element method and Principle of Virtual Work as well as solving for the answer by using Newton-Raphson method and LU decomposition. Chapter IV will show the construction of the computational algorithm of the membrane model using methodology from the previous two chapters. The proposed work will lead to precise computational results for the given system with a comparison of each constitutive model's effects on representing the membrane's deformation.

CHAPTER II

BACKGROUND AND METHODS

Due to the "membrane-like" nature of a flexible wing, we must discuss how an arbitrary body undergoes a deformation over time and take account the amount strain occurring on the body depending on its thin rubber membrane material. The main objective of our membrane model is to evaluate the position of the membrane as it deforms. In this chapter, we develop a methodology for computing the strain.

2.1 Characterization of Strain

An arbitrary body can be shown in two different states. There is the body's initial state (β_0) when it is undeformed, and there is its deformed state (β_t) after a given time *t*. Each arbitrary state has an associated position vector. For the initial state, the associated position vector is denoted as $\vec{X} = (X_1, X_2, X_3)$ with respect to the reference (undeformed) coordinate system, $\hat{E}_A$, (A = 1, 2, 3). At time *t*, the position vector for the deformed state is described as $\vec{x} = (x_1, x_2, x_3)$ with respect to the reference (undeformed) coordinate system, $\hat{e}_i$, (i = 1, 2, 3). The orthonormal bases, $\hat{E}_A$ and $\hat{e}_{i,}$ are related by a known translation and rotation [4], where these orthonormal bases can be aligned together when describing the deformation of an arbitrary body (see Figure 2.1).

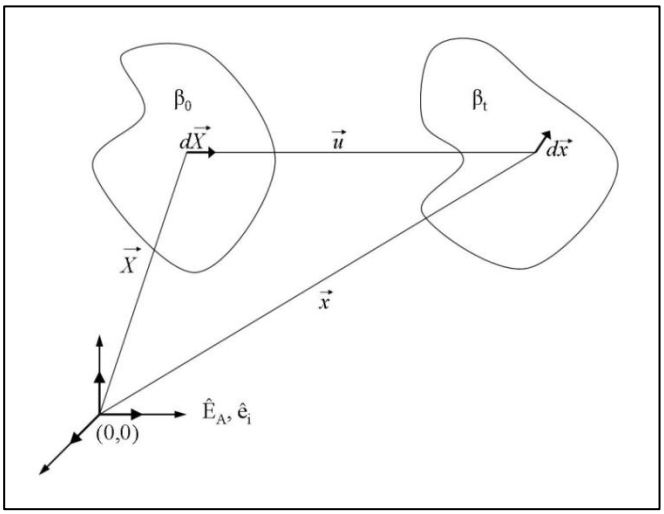


Figure 2.1: Deformation of an Arbitrary Body

Using Einstein notation, the position vectors for both the undeformed and deformed state can be written as:

$$\vec{X} = X_A \hat{E}_A = X_1 \hat{E}_1 + X_2 \hat{E}_2 + X_3 \hat{E}_3, \text{ and} \tag{2.1}$$

$$\vec{x} = x_A \hat{e}_A = x_1 \hat{e}_1 + x_2 \hat{e}_2 + x_3 \hat{e}_3 \tag{2.2}$$

Every particle has a motion associated with each position. The position vectors for both the undeformed and deformed bodies are shown in Figure 2.1. There also exists a displacement vector, $\vec{u}$ describing the movement between bodies, and it can be expressed as

$$\vec{u} = \vec{x} - \vec{X}. \tag{2.3}$$

To characterize the oriented differential line segments, we will set $d\vec{X}$ and $d\vec{x}$ as the differential position vectors of the particles in the states β_0 and β_t, respectively. To determine a relationship between the two states, use the chain rule incorporating both differential position vectors using the following formula:

$$d\vec{x} = \frac{\partial \vec{x}}{\partial \vec{X}}\, d\vec{X}. \tag{2.4}$$

The above equation is our obtained tensor, leading us to compute our fundamental measures of the deformation, denoted as $\tilde{F}$, also known as the *deformation gradient* [4]. The deformation gradient can be represented by

$$\tilde{F} = \frac{\partial \overrightarrow{x_i}}{\partial \overrightarrow{X_A}}\, \hat{e}_i \otimes \hat{E}_A, \tag{2.5}$$

and may be represented as a 3 x 3 matrix we have

$$\tilde{F} = \begin{pmatrix} \frac{\partial \overrightarrow{x_1}}{\partial \overrightarrow{X_1}} & \frac{\partial \overrightarrow{x_2}}{\partial \overrightarrow{X_1}} & \frac{\partial \overrightarrow{x_3}}{\partial \overrightarrow{X_1}} \\ \frac{\partial \overrightarrow{x_1}}{\partial \overrightarrow{X_2}} & \frac{\partial \overrightarrow{x_2}}{\partial \overrightarrow{X_2}} & \frac{\partial \overrightarrow{x_3}}{\partial \overrightarrow{X_2}} \\ \frac{\partial \overrightarrow{x_1}}{\partial \overrightarrow{X_3}} & \frac{\partial \overrightarrow{x_2}}{\partial \overrightarrow{X_3}} & \frac{\partial \overrightarrow{x_3}}{\partial \overrightarrow{X_3}} \end{pmatrix} \tag{2.6}$$

Note that $\tilde{F}$ is called a two-point tensor consisting of orthonormal bases from both coordinate systems. This causes a major disadvantage when using this tensor due to the difficulty in computing $\tilde{F}$ since it is not symmetric [14]. Since not enough information is known about both orthonormal bases, we will use another more convenient tensor that

will be easier to compute. This tensor will be a symmetric, one-point tensor independent of rigid body motion, $\tilde{C}$, such that

$$\tilde{C} = \tilde{F}^{\mathrm{T}}\tilde{F} \tag{2.7}$$

$$= \left(\frac{\partial \overrightarrow{x_i}}{\partial \overrightarrow{X_A}}\, \hat{e}_i \otimes \hat{E}_A\right)^{\mathrm{T}} \left(\frac{\partial \overrightarrow{x_J}}{\partial \overrightarrow{X_B}}\, \hat{e}_j \otimes \hat{E}_B\right)$$

$$= \frac{\partial \overrightarrow{x_i}}{\partial \overrightarrow{X_A}}\, (\hat{E}_A \otimes \hat{e}_i) \frac{\partial \overrightarrow{x_J}}{\partial \overrightarrow{X_B}}\, (\hat{e}_j \otimes \hat{E}_B)$$

$$= \frac{\partial \overrightarrow{x_i}}{\partial \overrightarrow{X_A}} \frac{\partial \overrightarrow{x_J}}{\partial \overrightarrow{X_B}}\, \left(\hat{E}_A \otimes \hat{e}_i\right) \cdot \left(\hat{e}_j \otimes \hat{E}_B\right). \tag{2.8}$$

Recall that since $\hat{e}_i$ and $\hat{e}_j$ are orthonormal bases, the product, $\hat{e}_i \cdot \hat{e}_j = \delta_{ij}$, which is the Kronecker delta. By computing the dot product, $\delta_{ij} = 0$ when $i \neq j$, and we have

$$\tilde{C} = \frac{\partial \overrightarrow{x_i}}{\partial \overrightarrow{X_A}} \frac{\partial \overrightarrow{x_i}}{\partial \overrightarrow{X_B}}\, \left(\hat{E}_A \otimes \hat{E}_B\right). \tag{2.9}$$

Using the Einstein notation, we will denote $\tilde{C}$ as

$$\tilde{C} = \mathrm{C}_{AB}(\hat{E}_A \otimes \hat{E}_B), \tag{2.10}$$

where,

$$C_{AB} = \frac{\partial x_1}{\partial X_A}.\frac{\partial x_1}{\partial X_B} + \frac{\partial x_2}{\partial X_A}.\frac{\partial x_2}{\partial X_B} + \frac{\partial x_3}{\partial X_A}.\frac{\partial x_3}{\partial X_B} \tag{2.11}$$

We call the tensor $\tilde{C}$, defined in equation (2.9), the *right Cauchy-Green strain tensor* [13], since $\tilde{C}$ is only defined in the reference configuration, β_0. It is also important to note that in the absence of motion in the initial undeformed state, then $\tilde{C} = \tilde{I}$, Therefore

we will measure another tensor that will equal $\tilde{0}$ when there is no deformation. Hence, we define, $\tilde{E}$, called the *Green strain* as

$$\tilde{E} = \frac{1}{2}(\tilde{C} - \tilde{I}). \tag{2.12}$$

Using equations (2.10) and (2.11) we can deduce that

$$E_{AB} = \frac{1}{2}\left[\frac{\partial \overrightarrow{x_\iota}}{\partial \overrightarrow{X_A}} \frac{\partial \overrightarrow{x_\iota}}{\partial \overrightarrow{X_B}} - \delta_{AB}\right]. \tag{2.13}$$

Note that the Green strain will prove to be useful when it comes to modeling nonlinear membrane behavior.

2.2 Computing Strain of Isotropic Membranes

For this section, the standard assumption that membranes are thin enough to not be affected to bending will still hold true. With this supposition, we can adequately approximate a three-dimensional membrane in only two dimensions. Our goal is to form the two-dimensional Cauchy-Green strain tensor discussed in the previous section that can be used for this membrane.

With the arbitrary body, we characterized the strain using orthonormal bases ($\hat{E}_A$and $\hat{e}_i$) in both the undeformed and deformed states accordingly. Our calculations for an isotropic membrane will no longer used fixed bases. Instead, we will use two-dimensional "element-wise" curvilinear bases, also known as tangential vectors, (G_α and g_α) which will associate with convecting coordinates ξ^1 and ξ^2in the undeformed and deformed states, respectively. Our two-dimensional assumption holds for a membrane

without significant thickness. Using these coordinates and bases we can define the position vectors using Einstein notation to get

$$\vec{\mathrm{X}} = \xi^1 G_1 + \xi^2 G_2, \text{ and} \tag{2.14}$$

$$\vec{x} = \xi^1 g_1 + \xi^2 g_2 \tag{2.15}$$

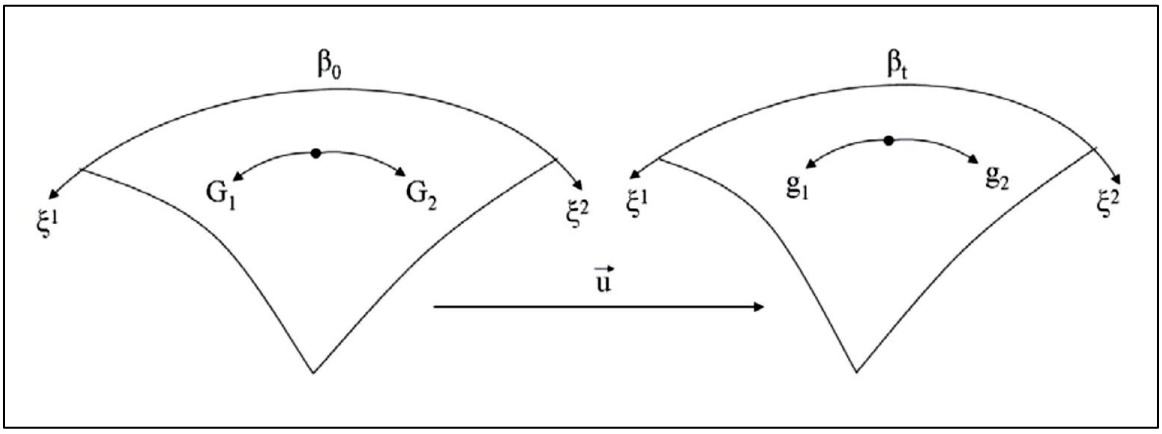


Figure 2.2: Deformation of an Isotropic Membrane

Though the curvilinear bases hold for any reference frame, we still want to use orthonormal bases to calculate the strain tensor. Hence, we will define orthonormal bases ($\hat{I}_1$ and $\hat{I}_2$) along with their associated coordinates (s_1 and s_2) in terms of the tangential vectors, G_1 and G_2. This will give us an orthonormal coordinate system corresponding to the undeformed state of the membrane [14].

$$\hat{I}_1 = \frac{G_1}{|G_1|}, \quad \hat{I}_n = \frac{G_1 \text{ x } G_2}{|G_1 \text{ x } G_2|}, \quad \hat{I}_2 = \hat{I}_n \text{ x } \hat{I}_1 \tag{2.16}$$

Thus, we now have an orthonormal basis, and we can define a position vector of the undeformed state as

$$\vec{\mathrm{X}} = s_1\hat{I}_1 + s_2\hat{I}_2, \tag{2.17}$$

and by letting A = 1,2, we can get,

$$\hat{I}_A = \frac{\partial\vec{\mathrm{X}}}{\partial s_A}. \tag{2.18}$$

Therefore, similar to the previous section, we can now calculate our deformation gradient, $\tilde{F}$, followed by our right Cauchy-Green strain tensor, $\tilde{C}$. By solving for the deformation gradient first, we get

$$\tilde{F} = \frac{\partial\vec{\mathrm{x}}}{\partial\vec{\mathrm{X}}} = \frac{\partial\xi^i}{\partial s_A}\left(g_i \otimes \hat{I}_A\right). \tag{2.19}$$

Associated with $\tilde{F}$ we can define the corresponding *two-dimensional* right Cauchy-Green strain tensor which is given by,

$$\tilde{C} = \mathrm{C}_{AB}\left(\hat{I}_A \otimes \hat{I}_B\right) \tag{2.20}$$

$$= \frac{\partial\vec{\mathrm{x}}}{\partial s_A}\cdot\frac{\partial\vec{\mathrm{x}}}{\partial s_B}\left(\hat{I}_A \otimes \hat{I}_B\right)$$

$$= \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}.$$

2.3 Description of Strain-Energy Functions

We will now describe the stress and strain on the membrane. As in previous sections, it will still hold that any bending is insignificant to the membrane model. Hence

we can model the behavior of our hyperelastic membrane implying the existence of a two-dimensional strain-energy function. We will define the strain-energy functions, $w(\tilde{C})$, where w is a function of two principal invariants of the Cauchy-Green strain tensor [4], $\tilde{C}$,

$$w = w(I_1, I_2) \tag{2.21}$$

where the principal invariants are,

$$I_1 = \text{trace}(\tilde{C}) = tr(\tilde{C}) = \mathrm{C}_{11} + \mathrm{C}_{22}, \text{ and} \tag{2.22}$$

$$I_2 = \frac{1}{2}(\text{tr}(\tilde{C})^2 - tr(\tilde{C}^2)) = \det(\tilde{C}) = \mathrm{C}_{11}\mathrm{C}_{22} - \mathrm{C}_{12}\mathrm{C}_{21}. \tag{2.23}$$

To derive the corresponding three-dimensional invariants, we must first define the appropriate three-dimensional Cauchy-Green strain tensor. Assuming that the membrane is incompressible, it can be derived that $\det(\tilde{F}) = 1$ and it follows that $\det(\tilde{C}) = 1$. This will be considered the isotropic hyperelasticity case. With this, we can define the Cauchy-Green strain tensor in plane stress as,

$$\tilde{C} = \begin{pmatrix} C_{11} & C_{12} & 0 \\ C_{21} & C_{22} & 0 \\ 0 & 0 & C_{33} \end{pmatrix}, \tag{2.24}$$

where,

$$C_{33} = \frac{1}{\mathrm{C}_{11}\mathrm{C}_{22} - \mathrm{C}_{12}\mathrm{C}_{21}} = \frac{1}{I_2} = I_2^{-1}. \tag{2.25}$$

Hence, we can define our three-dimensional invariants of $\tilde{C}$, $\bar{I}_1$ and $\bar{I}_2$, in a similar manner. In addition, we can express these invariants in terms of corresponding two-dimensional invariants by

$$\bar{I}_1 = I_1 + I_2^{-1}, \text{ and} \tag{2.26}$$

$$\bar{I}_2 = I_2 + I_1 I_2^{-1}. \tag{2.27}$$

From here we will seek the form of a strain-energy function w($\tilde{C}$) for our chosen representation of our "rubber-like" membrane cover [1]. A table of constitutive models for the material tested in our membrane model is presented below [2].

Table 2.1: Constitutive Models

Membrane Material	**Strain-Energy Function w($\tilde{C}$)**
Neo-Hookean	$w = \mu[(\bar{I}_1 - 3)]$
Mooney-Rivlin	$w = \mu[(\bar{I}_1 - 3) + \alpha(\bar{I}_2 - 3)]$
Polynomial form of Rivlin Model	$w = \sum_{i+j=0}^{n} C_{ij}[(\bar{I}_1 - 3)^i(\bar{I}_2 - 3)^j]$
Arruda-Boyce	$w = \mu\left[\frac{1}{2}(\bar{I}_1 - 3) + \frac{1}{20N}\left(\bar{I}_1{}^2 - 9\right) + \frac{11}{1050N^2}\left(\bar{I}_1{}^3 - 27\right) + \frac{19}{7000N^3}\left(\bar{I}_1{}^4 - 81\right) + \frac{519}{673750N^4}\left(\bar{I}_1{}^5 - 243\right)\right]$
Yeoh	$w = \sum_{i=1}^{n} C_i\,(\bar{I}_1 - 3)^i$

Specific values for all material parameters have been chosen from previous historical data. A commonly used membrane material tested in previous models is Mooney-Rivlin, where μ and α are the material parameters, with α being a dimensionless parameter [14]. This model is best to use when predicting the behavior of the membrane through deformation under general loading conditions [9]. The next strain-energy function can be defined when α = 0, which describes a Neo-Hookean membrane. For the Yeoh and Polynomial Rivlin membrane materials, the strain-energy functions can take the form of the Mooney-Rivlin and Neo-Hookean materials with special choices for i and j. Finally note that in the Arruda-Boyce strain-energy function is the first five terms of the *inverse Langevian function* [1], where N represents the stretch at which stress starts to increase without limit. The aim of this model is to represent the behavior of the membrane material while requiring only a small number of "physically based" parameters [9]. The effects of each constitutive model will be further discussed in the results section.

At this point, we can now use the strain-energy functions and the derivatives of the invariants. We can solve for the two-dimensional second Piola-Kirchoff stress consisting of both the isotropic and anisotropic components given by [13],

$$\tilde{S} = 2\,\frac{\partial w}{\partial \tilde{C}} = 2\left[\frac{\partial w}{\partial I_1}\frac{\partial I_1}{\partial \tilde{C}} + \frac{\partial w}{\partial I_2}\frac{\partial I_2}{\partial \tilde{C}}\right]. \tag{2.28}$$

Using all of the membrane mechanics discussed above, we can now go forth to explain the methodology used in obtaining a solution of the membrane deformation for different constitutive models.

CHAPTER III

METHODOLOGY FOR MEMBRANE SOLUTIION

To accurately find the position of our deformed membrane, we must first partition the membrane into a finite number of elements. Then we can apply the Principle of Virtual Work for each element. In the following sections, we will go into further detail of each step of the methodology.

3.1 Finite Element Discretization

Through the process of our model, the membrane will be partitioned into a finite number of elements [11]. We can do this by employing a *Total Langrangian* formulation where all elements are defined with respect to the undeformed configuration [14]. We will first define the isoparametric element shape function matrix **N** by,

$$\mathbf{N} = [N_1 \;\; N_2 \;\; N_3 \;\; N_4], \tag{3.1}$$

where,

$$N_1(r,s) = 0.25(1-r)(1-s) \tag{3.2}$$

$$N_2(r,s) = 0.25(1+r)(1-s) \tag{3.3}$$

$$N_3(r,s) = 0.25(1+r)(1+s) \tag{3.4}$$

$$N_4(r,s) = 0.25(1-r)(1+s). \tag{3.5}$$

Note that the shape functions are defined on a reference square (-1, 1) x (-1, 1), such that r,s ε (-1, 1) [3].

From the definition of shape functions, we can also obtain local derivatives of the shape function,

$$\begin{bmatrix} \frac{\partial \mathbf{N}}{\partial \xi^1} \\ \frac{\partial \mathbf{N}}{\partial \xi^2} \end{bmatrix} = \begin{bmatrix} \frac{\partial N_1}{\partial \xi^1} & \frac{\partial N_1}{\partial \xi^2} & \frac{\partial N_1}{\partial \xi^3} & \frac{\partial N_1}{\partial \xi^4} \\ \frac{\partial N_2}{\partial \xi^1} & \frac{\partial N_2}{\partial \xi^2} & \frac{\partial N_2}{\partial \xi^3} & \frac{\partial N_2}{\partial \xi^4} \end{bmatrix}. \tag{3.6}$$

Next, we can now approximate the original position vectors by using the shape function matrix, **N** as,

$$\vec{X} = \mathbf{N}\vec{X}^e, \text{ and} \tag{3.7}$$

$$\vec{x} = \mathbf{N}\vec{x}^e. \tag{3.8}$$

Recall that the position vectors, $\vec{X}$ and $\vec{x}$, refer to the undeformed and deformed states respectively, each with their corresponding element coordinates as well. From here we can also obtain $\delta\vec{x}$ by,

$$\delta\vec{x} = \delta\vec{x}^e \tag{3.9}$$

The curvilinear bases can now be rewritten to incorporate the derivatives of equations (3.7) and (3.8) with respect to ξ^i, yielding

$$G_1 = \frac{\partial \mathbf{N}}{\partial \xi^1}\vec{X}^e \tag{3.10}$$

$$G_2 = \frac{\partial \mathbf{N}}{\partial \xi^2}\vec{X}^e \tag{3.11}$$

$$g_1 = \frac{\partial \mathbf{N}}{\partial \xi^1}\vec{x}^e \tag{3.12}$$

$$g_2 = \frac{\partial \mathbf{N}}{\partial \xi^2} \vec{x}^e. \tag{3.13}$$

Similarly we have

$$\frac{\partial \vec{X}}{\partial s_\alpha} = \frac{\partial \mathbf{N}}{\partial s_\alpha} \vec{X}^e \tag{3.14}$$

$$\frac{\partial \vec{x}}{\partial s_\alpha} = \frac{\partial \mathbf{N}}{\partial s_\alpha} \vec{x}^e. \tag{3.15}$$

One can also construct the Jacobian transformation matrix, **J**, expressing the components as

$$J_{AB} = \frac{\partial s_\alpha}{\partial \xi^i} = G_A \,.\, \hat{I}_B \tag{3.16}$$

Combining this equation with equation (3.6), we can compute

$$\begin{bmatrix} \frac{\partial \mathbf{N}}{\partial s_1} \\ \frac{\partial \mathbf{N}}{\partial s_2} \end{bmatrix} = \mathbf{J^{-1}} \begin{bmatrix} \frac{\partial \mathbf{N}}{\partial \xi^1} \\ \frac{\partial \mathbf{N}}{\partial \xi^2} \end{bmatrix} \tag{3.17}$$

3.2 Principle of Virtual Work

The next approach in solving the membrane model will be to apply the *Principle of Virtual Work* [17] which applies to arbitrary bodies going under deformation. The Principle of Virtual Work states that the sum of the work done by external applied forces and internal forces done by virtual displacements is zero. Note that this assumption assumes there are no body forces present. This equilibrium can be shown as

$$\int_V \delta\vec{x}\,\rho_0\,\ddot{\vec{x}}\,dV + \int_A \delta w dA = \int_a p\,\hat{n}\;\delta\vec{x}da\,, \tag{3.18}$$

where V is the volume of the undeformed membrane; A and a represent the surface areas of the undeformed and deformed membrane states correspondingly; ρ_0 is the mass density; $\ddot{\vec{x}}$ is the acceleration vector; p is the external uniform pressure that acts on the membrane; and $\hat{n}$ is the unit normal vector indicating the direction of the acceleration vector.

3.2.1 External Forces

The external force applied refers to the right side of equation (3.18)

$$\int_a p\,\hat{n}\;\delta\vec{x}da \;=\; \int_a p\,(\delta\vec{x})^T\hat{n}\,da\,, \tag{3.19}$$

which is defined on the deformed state of the membrane. It will be useful to define our forces in terms of bases that we can easily solve. Hence, we define the outward unit normal vector, $\hat{n}$ as

$$\hat{n} = \frac{g_1 \text{ x } g_2}{|g_1 \text{ x } g_2|}. \tag{3.20}$$

We will also express the differential areas in both the undeformed and deformed states of the membrane as

$$dA = |G_1 \text{ x } G_2|\,d\xi^1 d\xi^2, \text{ and} \tag{3.21}$$

$$da = |g_1 \text{ x } g_2|\,d\xi^1 d\xi^2, \tag{3.22}$$

from which we can obtain

$$da = \frac{|g_1 \text{ x } g_2|}{|G_1 \text{ x } G_2|}\, dA. \tag{3.23}$$

Substituting equations (3.22) and (3.23), the external forces (right-hand side) of the Principle of Work becomes

$$\int_a p\, (\delta\vec{x})^{\mathrm{T}} \hat{n}\, da \; = (\delta\vec{x}^{\,e})^{\mathrm{T}} \int_a \mathbf{g}_p^e \; d\xi^1 d\xi^2, \tag{3.24}$$

where

$$\mathbf{g}_p^e = p\, \mathbf{N}^{\mathrm{T}} (g_1 \text{ x } g_2). \tag{3.25}$$

3.2.2 Internal Forces

Similarly we can now consider the internal force terms corresponding to the left-hand side of equation (3.18) and simplify the expression assuming only the quasi-static case in which we disregard the acceleration term

$$\int_A \delta w dA \; = \int_A \frac{\partial w}{\partial \tilde{C}} \delta\tilde{C} dA$$

$$= \int_A \left[\frac{\partial w}{\partial I_1}\frac{dI_1}{d\tilde{C}} + \frac{\partial w}{\partial I_2}\frac{\partial I_2}{\partial \tilde{C}}\right] \delta\tilde{C} dA \; . \tag{3.26}$$

Also, we can derive the component-wise derivative of the Cauchy-Green strain tensor such that we have

$$\delta C_{AB} = \left(\frac{\partial C_{AB}}{\partial \vec{x}^{\,e}}\right)^{\mathrm{T}} \delta\vec{x}^{\,e}. \tag{3.27}$$

Therefore we can simplify the internal forces in equation (3.26) even further to obtain

$$\int_A \delta w dA \; = (\delta \vec{x}^e)^{\mathrm{T}} \int_A \mathbf{g}_w^e \; d\xi^1 d\xi^2, \tag{3.28}$$

where,

$$\mathbf{g}_w^e = \left(\frac{\partial w}{\partial C_{AB}}\right)\left(\frac{\partial C_{AB}}{\partial \vec{x}^e}\right)|G_1 \text{ x } G_2|. \tag{3.29}$$

3.2.3 Equilibrium Solution

Now that we have expressed the external and internal forces, we can now express the Principle of Virtual Work at equilibrium for each finite element. We will only consider the quasi-static case in which we disregard the acceleration term in (3.18) which expands to

$$\int_A \delta w dA \; = \int_a p\,(\delta \vec{x})^{\mathrm{T}} \hat{n}\, da\,. \tag{3.30}$$

Substituting the reduced internal and external forces found from the Principle of Virtual Work equation, we have

$$(\delta \vec{x}^e)^{\mathrm{T}} \int_A \mathbf{g}_w^e \; d\xi^1 d\xi^2 = \; (\delta \vec{x}^e)^{\mathrm{T}} \int_a \mathbf{g}_p^e \; d\xi^1 d\xi^2, \tag{3.31}$$

which can be combined to get

$$(\delta \vec{x}^e)^{\mathrm{T}} \int_A (\mathbf{g}_w^e \; - \; \mathbf{g}_p^e)\, d\xi^1 d\xi^2 = 0. \tag{3.32}$$

One equation can be obtained for each element calculated during finite discretization. Assembling these equations with the use of a nodal-to-element connectivity matrix [14] forms a system of nonlinear equations

$$\mathbf{g}(\delta\vec{x}) = 0. \tag{3.33}$$

We now proceed to solve this system by employing the Newton-Raphson process [3] given by

$$\mathbf{K}\left(\delta\vec{x}^{(i)}\right)\left[\delta\vec{x}^{(i+1)} - \delta\vec{x}^{(i)}\right] = -\mathbf{g}\left(\delta\vec{x}^{(i)}\right), \tag{3.34}$$

where **g** is the residual vector after the i^{th} iteration and **K(**$\delta\vec{x}$) is the tangent stiffness matrix specified as

$$\mathbf{K} = \frac{\partial \mathbf{g}}{\partial(\delta\vec{x})}. \tag{3.35}$$

This leads us to solve the system of equations using Newton-Raphson method and lead us to the solution of the membrane model. The computational results of this method will be discussed in Chapter V.

CHAPTER IV

CONSTRUCTING THE COMPUTATIONAL ALGORITHM

Our next approach is to follow the methodology presented in the previous chapters and convert it into a functioning algorithm to solve for the membrane deformations automatically. One important aspect of numerical analysis is to increase accuracy by decreasing the number of computational operations. Another related objective is to optimize processing speeds of the simulation. The following chapter will go through the process of building a computational algorithm which will representatively model the membrane after it undergoes deformation.

4.1 Initial Implementation

Before building the model, we create a flowchart outlining the processes taking place in the simulation. We can divide our algorithm into three key subcomponents, the preprocessor, the processor, and the post-processor [9]. The overall structure of each process will be discussed in more detail in the following sections. A diagram showing the flow of all processes can be shown in Figure 4.1.

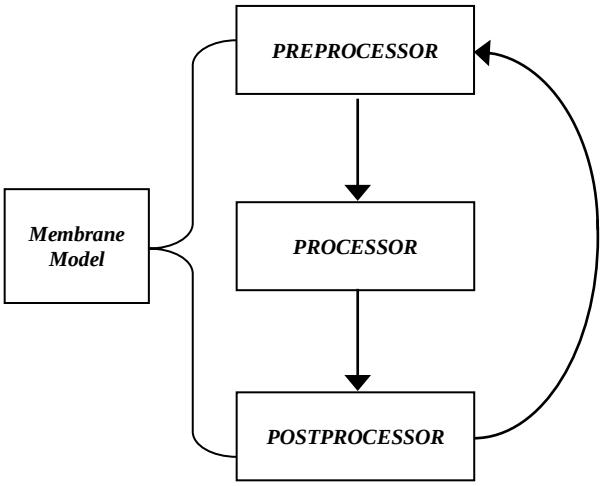


Figure 4.1: Main Algorithm for Membrane Model

4.2 Preprocessor

The preprocessor occurs first when running the code. In these preliminary steps, we will set-up our domain for the interface. Here, we will first declare set parameters to define the membrane's initial state, such as defining the number finite elements and specific constitutive model to run, as well as number of iterations and tolerance for running our algorithm.

We will continue to create a uniform rectilinear mesh on the domain and construct a pointer matrix which tracks the location and type of all basis functions, or piece-wise polynomials. Afterwards, we will find and impose the boundary conditions onto our clamped-clamped membrane problem. This will locate all fixed and moving nodes which will be important in calculating the deformation of the membrane. This will help us generate our initial guess for our membrane solution. Figure 4.2, shown below, details the flowchart of the preprocessor.

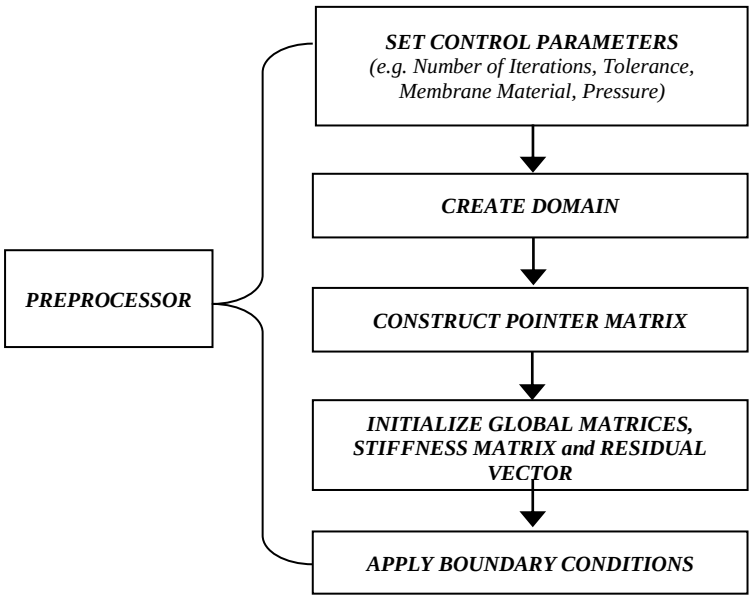


Figure 4.2: Preprocessor Flowchart

4.3 Processor and Postprocessor

The processor of the membrane model is where most of the methodology and computations are done. We will begin by implementing a *Newton-Raphson* recursion to find an equilibrium solution for the tangent stiffness matrix and residual vector, **K** and **g** respectively. We will start by using methods discussed in previous chapters for each finitely discretized elements (e.g. computing position vector from shape function matrix, solving for orthonormal and curvilinear bases, calculating the strain-energy function, applying Principle of Virtual Work). A walkthrough specifying the order of methods used in the processor stage is shown in Figure 4.3. After accumulating the global matrices **K** and **g** from each element, we apply boundary conditions and solve for the

updated position matrix. Note that in C++, we will implement LU decomposition and Gauss-Jordan elimination [10]. By writing the matrix **K** as the product of a lower and upper triangular matrix, we can significantly reduce the number of operations used in finding our solution when using Gauss-Jordan elimination. In Matlab, we use the built-in "back-slash" command from its Linear Algebra package, LAPACK. Finally we will find the maximum error between the current and new membrane positional matrix. The Newton-Raphson continues until we reach our maximum number of iterations or the maximum error is less than the tolerance set in the preprocessing step.

The postprocessor has a similarly structured flowchart which can be seen in Figure 4.4. This final stage of the membrane model is responsible for interpreting the final solution. In this method, we will output and store the concluding results to the user such as the center-node deflection, error estimate, and number of iterations needed to find the answer. We can then determine which variables declared in the pre-processing phase we may want to change to compare to our previous solution. Finally, we can load select solutions into our program to plot various figures to compare results when changing certain parameters.

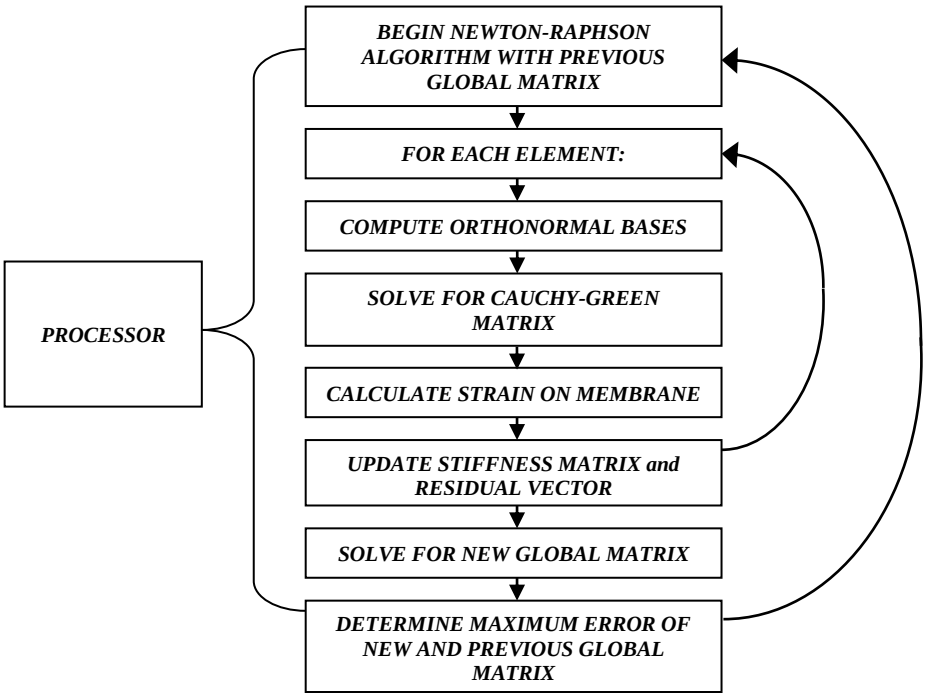


Figure 4.3: Processor Flowchart

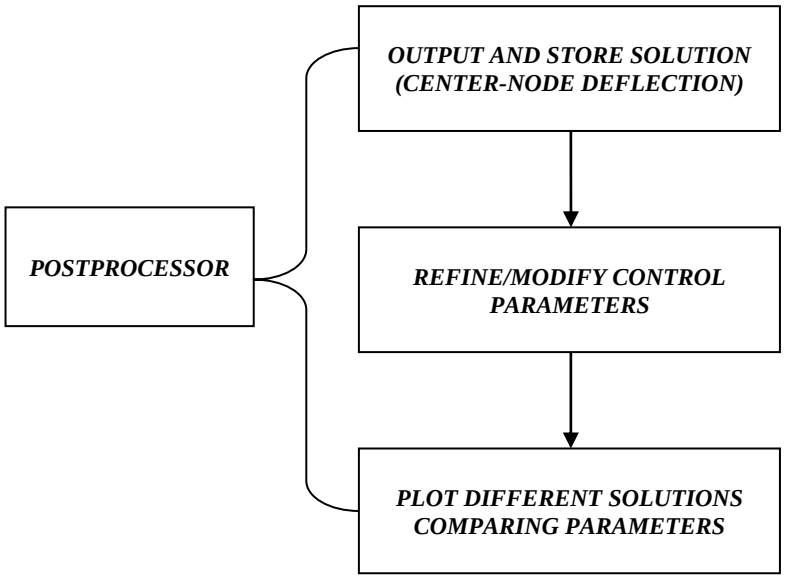


Figure 4.4: Postprocessor Flowchart

4.5 High Performance Programming

The original model for the membrane solution was programmed in Matlab. Matlab, which is short for Matrix Laboratory, is a numerical computing environment which is developed with main uses being matrix manipulations, plotting functions, and implementation of algorithms [8]. Unfortunately, the use of Matlab with large, complex programs such as the membrane model can lead to slower execution times.

To improve the speed and performance of the model, the membrane model has been converted into the C++ programming language as well. This language is built to support techniques focused on processing, and it can be used to optimize the model for efficiency [8]. Optimization of the model will be tested for both programming languages. Each has their own unique capabilities and limitations, which will ultimately result in different performances. Both the memory usage and number of calculations and functions play a critical role in determining the accuracy of the solution. For a complex membrane model, matrices can become arbitrarily large which uses numerous calculations and could lead to memory issues.

In the next chapter we will compare and discuss results running the model in Matlab and C++. For the case of programming our model, we will analyze the implementation in both programming languages as well as the final results. From there we will decide whether we can conclude if one language is more accurate for immediate future work using the membrane model as well as what algorithm improvements need to be made to make a model in either programming language more efficient.

CHAPTER V

NUMERICAL RESULTS AND DISCUSSION

In this chapter, we will present the numerical results for the implementation of the computational algorithm developed in this thesis. In particular, we apply the methodology against a model problem that was considered in [18] which evaluated the performance of a membrane element for the Neo-Hookean model. The approach used in this paper was very different and employed the three dimensional strain invariants. To compare the performance of the method described in this work, we considered the same benchmark problem considered in [18].

Our membrane model uses 25 finitely discretized elements all of equal size. Since our assumption is that the pressure is uniform and acting on the center of the membrane, we can use symmetry in the square membrane to only solve for one-quarter of the membrane. We will also assume that the edges are fixed. Finally, our model implements any of our defined strain-energy functions. Note that the original membrane model was written in Matlab [3], which has since been converted into C++ programming language with enhanced features.

The original test problem [18] entailed a flat, square Neo-Hookean material with fixed edges and dimensions of 2 x 2 x h units, such that h << 1 and consisted of fixed edges. The initial undeformed sheet is defined by the domain

$$\left\{(x,y,z)|-1\ \le x\ \le 1,-1\ \le y\ \le 1,-\frac{h}{2}\le z\ \le\ \frac{h}{2}\right\}. \tag{4.1}$$

The membrane is also subject to a uniform non-dimensionalized pressure p on the face $z = -\frac{h}{2}$. Comparing the deflection of the center-node which is located at the origin (0, 0, 0), Table 5.1 summarizes the results from the original membrane model built in Matlab and C++ compared to the reported results from the model used by Yang and Lu [18]. Note that the values in both programming languages are identical to the given tolerance.

Table 5.1: Center-Node Deflection of Neo-Hookean Model

Pressure	**Yang & Lu**	**Membrane Model (Matlab/C++)**
0.25	0.32	0.3150
0.5	0.42	0.4214
0.75	0.51	0.5129
1.0	0.60	0.6041
1.25	0.70	0.7059
1.5	0.84	0.8374
1.75	1.1	1.0937

Now we must verify our models integrity when implementing newer material entities. To do this, we will test models in both programming languages for the different membrane materials and compare results. To test that the transformation of the model

into C++ programming language we will compute the center-node deflection and membrane deformation at the uniform non-dimensionalized pressure of 0.25 in the updated model in both Matlab and C++. These results are shown in Table 5.2, below.

Table 5.2: Membrane Model Comparison between Programming Languages

Material	**Matlab Model**	**C++ Model**
Rivlin-Polynomial	0.2941	0.2941
Arruda-Boyce	0.2591	0.2591
Yeoh	0.2998	0.2998

The material parameters used for each strain-energy function are the following:

1) Neo-Hookean: C10 = 1, C01 = C11 = 0
2) Mooney-Rivlin: C10 = 1, C01 = 0.2, C11 = 0
3) Rivlin-Polynomial: C10 = 1, C01 = 0.2, C11 = .2
4) Arruda-Boyce: $\mu = 1$, N = 2
5) Yeoh: C1= .634, C2 = .066, C3 = 0.21

These values are gathered from parameter estimations from related research and can lead to different deformations if changed. It is likely that each constitutive model with the given parameters does not relate to an identical material, but they are still "good-fit" models. To measure the results of all constitutive models, we gather the results of center-

node deflections at increasing values of pressure from 0.25 to 1.75 and present the results in Table 5.3. Secondly we display visual graphs comparing each model at a pressure of .25. Finally we illustrate the quarter membrane undergoing deformation with the Neo-Hookean constitutive model at similar pressures. The numerical results validate the computational model developed in this work.

Table 5.3: Center-Node Deflection of Constitutive Models

Pressure	**Neo-Hookean**	**Mooney-Rivlin**	**Rivlin-Polynomial**	**Arruda-Boyce**	**Yeoh**
0.25	0.3150	0.2948	0.2941	0.3504	0.2998
0.5	0.4214	0.3892	0.3861	0.4725	0.3957
0.75	0.5129	0.4660	0.4582	0.5771	0.4722
1.0	0.6041	0.5366	0.5207	0.7742	0.5403
1.25	0.7059	0.6060	0.5774	0.8680	0.6037
1.5	0.8374	0.6775	0.6298	0.9551	0.6640
1.75	1.0937	0.7542	0.6788	1.0333	0.7215

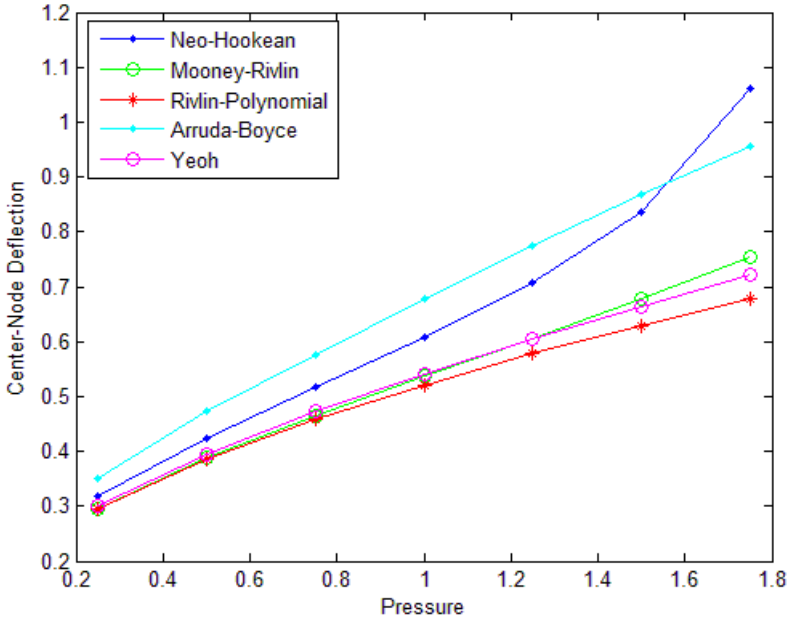


Figure 5.1: Comparison of Center-Node Deflections at Varying Pressures

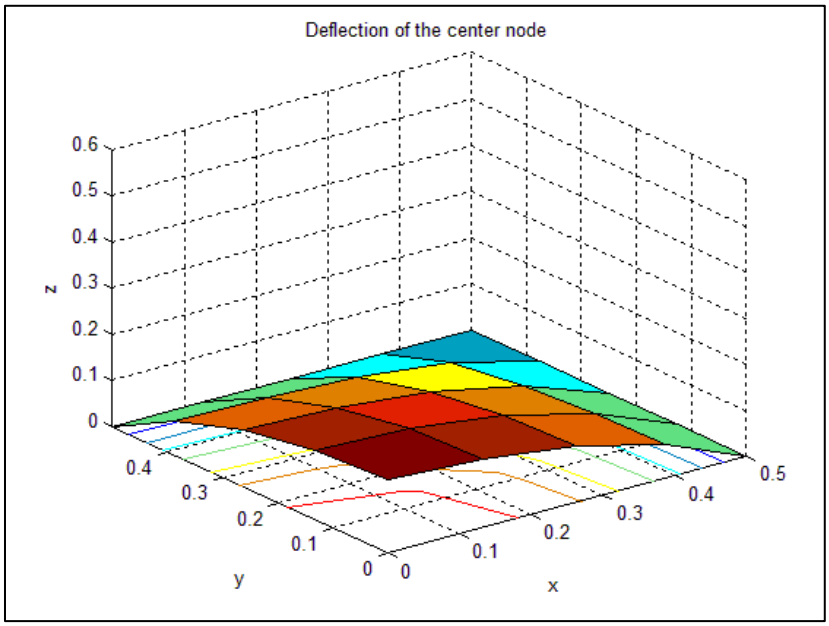


Figure 5.2: Membrane Deformation (Pressure = 0.25)

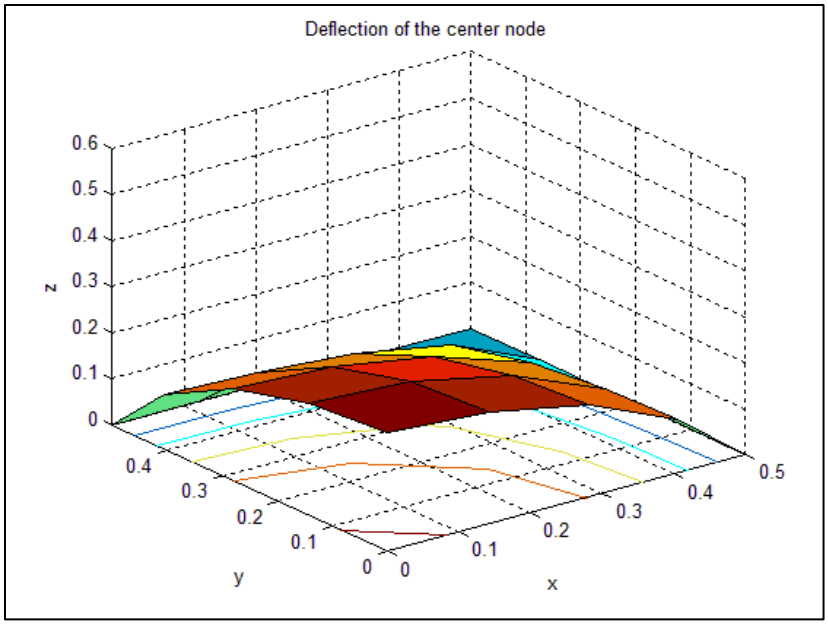


Figure 5.3: Membrane Deformation (Pressure = 0.75)

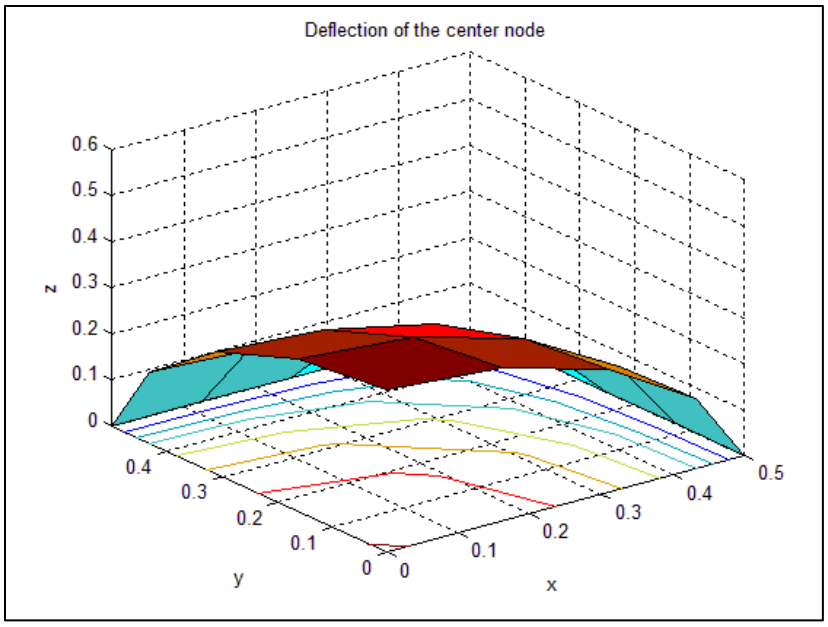


Figure 5.4: Membrane Deformation (Pressure = 1.25)

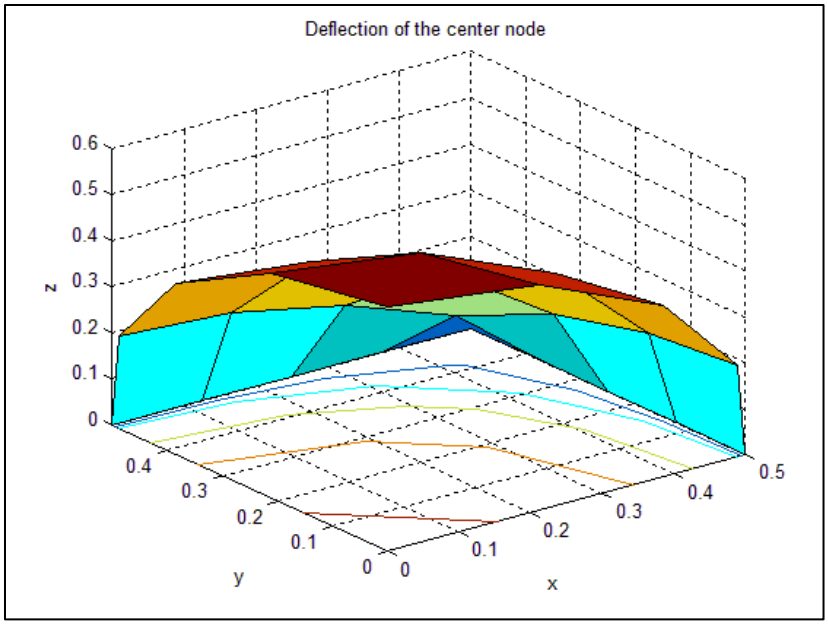


Figure 5.5: Membrane Deformation (Pressure = 1.75)

CHAPTER VI

CONCLUSION AND FUTURE RESEARCH

In this thesis, we have proposed to update an existing membrane model for the flexible-based wing structure of Micro Air Vehicles. As discussed in this paper, this model is intended for an isotropic hyperelastic membrane which uses finite element method, Principle of Virtual Work, as well as various constitutive models.

We have explained in depth the methodology for different constitutive models for the membrane materials and the comparison of their computational results. Each constitutive model had varying results with many models being consistent with each other. In relation to related projects [2], the most complex models produce consistent results, while the Neo-Hookean and Arruda-Boyce models do not fit the data trend of other models as well.

For the simple test cases using the Neo-Hookean constitutive model, we have received consistent results where deformations of the membrane converge comparable to studies of previous models. With all the constitutive models, we did not include the nonlinear terms which would make it possible to solve for complex deformations of the membrane body. We also used set material parameters for each model. For future work, we would like to be able to compare our model with experimental results. To do so, we

would need to estimate and test for the material parameters of our constitutive models that form the "best fit" data to the actual results.

Similarly, other assumptions were made to simplify the membrane model. For one, we assumed a quasi-static case when solving the Principle of Virtual Work. This assumption dropped the time-dependent acceleration term from our equation limiting the modeling aspect of the membrane deformation over time. Secondly, the model considered only the case of the squared-membrane followed rectangle geometry. For more accurate results, the membrane model would have to incorporate complex circular and complex geometries. Lastly, we assumed the membrane materials had an isotropic behavior. In contrast, research shows that many materials exhibit anisotropic properties which become more complex to model [13]. By implementing more complex materials, we could find the most precise constitutive model to use to best model the MAV wing.

We can also continue to optimize the results precision and execution speed by building the model in C++ programming language from Matlab (see Appendix for source code). Only in recent versions of Matlab, has the capability in object-oriented programming been available [8], but it is simple to incorporate this structure into C++. One disadvantage of our membrane model in C++ programming language is the number of computations and increasing precision. As matrices become arbitrarily large, the number of calculations performed and the accuracy of our solution may decrease with an error larger than our tolerance. This was not the case for testing performed in this paper. Another disadvantage of C++ is the lack of image processing and plotting functions without the use of third-party library.

For future work, we could incorporate additional libraries for enhanced precision and decreased runtime of numerous matrix calculations, including an optimized solver for solving a system Ax=b. Even without implementing a Linear Algebra library such as LAPACK or BLAAS, C++ still produced a result more quickly than Matlab. With a more accurate matrix class structure built in C++ programming language, not only could we optimize run-time, we can also incorporate the beam model to test the membrane-beam coupled approach in C++. Additionally, we would then be able to incorporate constitutive models for the beam model in a fully-coupled model of the Micro Air Vehicle's wing structure.

Finally, it is important to eliminate the need to rebuild the executable each time the program is run. Therefore one of the next steps needed to optimize our testing in C++ would be to create a configuration file to be used with the membrane. Here we could setup and read in all control variables before starting the code. Then we could simply change a value in this file before running the executable again.

APPENDIX

```
#include <iostream>
#include <vector>
#include <stdlib.h>
#include <math.h>

using namespace std;
#define _USE_MATH_DEFINES

void sort(vector<double>&,int);
void swap(int&, int&);

void gauss_quadrature(int, vector<double>&, vector<double>&);
void shape_function(double, double, vector<double>&);
void shape_function_der(double, double, vector<vector<double>>&);

void ludcmp(vector<vector<double>>&,int, vector<double>&);
void lubksb(vector<vector<double>>&,int,vector<double>&, vector<double>&);

typedef vector<double> Vec;
typedef vector<Vec> Mat;

int main ()
{
// Membrane Finite Element Method

int pressureLength = 1;      Vec PRESSURE (pressureLength);
PRESSURE[0] = 1; double INPUT_PRESSURE = PRESSURE[0];
int MATERIAL = 1;

double TOLERANCE = 1e-8; int NITER_MAX = 100; int ITER_COUNTER;
int MD=4; int NELEMS_SIDE_X = MD; int NELEMS_SIDE_Y=MD;
int LENGTH = 1;int WIDTH = 1;   double eucl_norm_res_vect;
int USING_STRETCH=1; int USING_SYMMETRY=1;int FLAG_CONV;

// CREATE GEOMETRY (Rectangular Domain)
// Creating Nodes
int NNODES_SIDE_X = NELEMS_SIDE_X + 1; int NNODES_SIDE_Y = NELEMS_SIDE_Y + 1;
int NNODES_TOTAL = NNODES_SIDE_X * NNODES_SIDE_Y; const int threeNODESTOTAL =
3*NNODES_TOTAL;

// Initialize X_GLOBAL
const int xglobalLength = NNODES_SIDE_X * NNODES_SIDE_Y;
Mat X_GLOBAL (xglobalLength,3);          Vec error_xg (3*NNODES_TOTAL); double max_error_xg;
```

```
for (int I=0; I<NNODES_SIDE_Y; I++){
for (int J=0; J<NNODES_SIDE_X; J++){
X_GLOBAL[(NNODES_SIDE_Y)*I+J][0] = (double) J/NELEMS_SIDE_X;
X_GLOBAL[(NNODES_SIDE_Y)*I+J][1] = (double) I/NELEMS_SIDE_Y;
X_GLOBAL[(NNODES_SIDE_Y)*I+J][2] = (double) 0;

if (USING_SYMMETRY==1){
X_GLOBAL[(NNODES_SIDE_Y)*I+J][0] = X_GLOBAL[(NNODES_SIDE_Y)*I+J][0]/2;
X_GLOBAL[(NNODES_SIDE_Y)*I+J][1] = X_GLOBAL[(NNODES_SIDE_Y)*I+J][1]/2;
X_GLOBAL[(NNODES_SIDE_Y)*I+J][2] = X_GLOBAL[(NNODES_SIDE_Y)*I+J][2]/2;
} } }

for (int I = 0; I<(NNODES_SIDE_X*NNODES_SIDE_Y); I++){
X_GLOBAL[I][0]=LENGTH * X_GLOBAL[I][0];
X_GLOBAL[I][1]=WIDTH * X_GLOBAL[I][1];
}

// Initialize POINTER_MATRIX
const int NELEMS_TOTAL = NELEMS_SIDE_X*NELEMS_SIDE_Y; int NSHAPE_ELEM = 4; int
NODE_INDEX;
int PM1; int PM2; int PM3; int PM4;          Mat POINTER_MATRIX (NELEMS_TOTAL, 4);

for (int J=0; J<NELEMS_SIDE_Y; J++){
for (int I=0; I<NELEMS_SIDE_X; I++){
NODE_INDEX = (J)*NELEMS_SIDE_X + (I+1);
PM1 = (J)*NNODES_SIDE_X + (I+1);
PM2 = PM1 + 1;
PM3 = (J+1)*NNODES_SIDE_X + (I+1) + 1;
PM4 = PM3 - 1;
POINTER_MATRIX[(NELEMS_SIDE_Y*J)+I][0] = PM1;
POINTER_MATRIX[(NELEMS_SIDE_Y*J)+I][1] = PM2;
POINTER_MATRIX[(NELEMS_SIDE_Y*J)+I][2] = PM3;
POINTER_MATRIX[(NELEMS_SIDE_Y*J)+I][3] = PM4;
} }

// DEFINING CENTER-NODE (Used to monitor deflection when post-processing
// Defining Center-node
Vec CENTER_NODE (MD+1);
if (USING_SYMMETRY==1){
for (int I=0; I<=MD; I++){
CENTER_NODE[I] = I;
}
}
else{
CENTER_NODE[0] = NNODES_SIDE_X*MD/2 + MD/2;
}

// IDENTIFY FIXED_NODES and MOVING_NODES (For Boundary Condition)
// Initialize FIXED NODES (that are constrained)
//(Clamped-Free)

int rightLength = NNODES_SIDE_X;
```

```
int topLength = ((NNODES_SIDE_X*NNODES_SIDE_X)-1)-(NELEMS_SIDE_Y*NNODES_SIDE_X +
1)+1;
int fixallLength = rightLength + topLength;
int xLength = NELEMS_SIDE_Y; int yLength = NELEMS_SIDE_Y;
int fixnodesLength = xLength + yLength + 3*fixallLength;
Vec FIX_ALL (fixallLength);         Vec FIX_NODES (fixnodesLength);

for (int I = 0; I<rightLength; I++){
FIX_ALL[I] = (I+1)*NNODES_SIDE_X;
}
for (int I = 0; I<topLength; I++){
FIX_ALL[I+rightLength] = (NELEMS_SIDE_Y*NNODES_SIDE_X+1) + I;
}
for (int I = 0; I<3*fixallLength; I++){
if (I<fixallLength){
FIX_NODES[I] = FIX_ALL[I];
}
else if (I<fixallLength*2){
FIX_NODES[I] = NNODES_TOTAL + FIX_ALL[I-fixallLength];
}
else if (I<fixallLength*3){
FIX_NODES[I] = 2*NNODES_TOTAL + FIX_ALL[I-2*fixallLength];
} }
for (int I=0; I<xLength; I++){
FIX_NODES[I+3*fixallLength] = I * NNODES_SIDE_X + 1;
}
for (int I=0; I<yLength; I++){
FIX_NODES[I+3*fixallLength+xLength] = NNODES_TOTAL + I+1;
}
sort(FIX_NODES,fixnodesLength);

// FREE NODES
int *MOVING_NODES_1 = new int[3*NNODES_TOTAL];   int movingnodesLength = 0;

for (int I = 0; I<(3*NNODES_TOTAL); I++){
MOVING_NODES_1[I] = I+1;
}
for (int I = 0; I<fixnodesLength; I++){
for (int J = 0; J<(3*NNODES_TOTAL); J++){
if (FIX_NODES[I] == MOVING_NODES_1[J]){
movingnodesLength++;
break;
} } }

movingnodesLength = 3*NNODES_TOTAL - movingnodesLength;
Vec MOVING_NODES (movingnodesLength); int skipnode; int movingnodeN = 0;

for (int J = 0; J<(3*NNODES_TOTAL); J++){
skipnode = 0;
for (int I = 0; I<fixnodesLength; I++){
if (FIX_NODES[I] == MOVING_NODES_1[J]){
skipnode = 1;
}
```

```
if (I == (fixnodesLength-1) && skipnode == 0){
MOVING_NODES[movingnodeN] = J+1;
movingnodeN++;
} } }
delete[] MOVING_NODES_1;

//Create Initial Guess
double STRETCH_FACTOR = 1.001; Mat x_GLOBAL (xglobalLength, 3);

for (int I=0; I<xglobalLength; I++){
for (int J=0; J<3; J++){
x_GLOBAL[I][J] = X_GLOBAL[I][J] * STRETCH_FACTOR;
} }
// CONSTRAINED NODES: incorporated in first guess
for (int I=0; I<fixnodesLength; I++){
if (FIX_NODES[I]-1 < NNODES_TOTAL){
x_GLOBAL[FIX_NODES[I]-1][0]=X_GLOBAL[FIX_NODES[I]-1][0];
}
else if (FIX_NODES[I]-1 < 2*NNODES_TOTAL){
x_GLOBAL[(FIX_NODES[I]-NNODES_TOTAL)-1][1]=X_GLOBAL[(FIX_NODES[I]-
NNODES_TOTAL)-1][1];
}
else{
x_GLOBAL[(FIX_NODES[I]-2*NNODES_TOTAL)-1][2]=X_GLOBAL[(FIX_NODES[I]-
2*NNODES_TOTAL)-1][2];
} }

// GETTING POINTS AND WEIGHTS FOR GAUSSIAN QUADRATURE
// Choosing number of gauss points for exact answer for linear approximation
const int NQUAD_POINTS_1D = 2; Vec POINTS (NQUAD_POINTS_1D); Vec WEIGHTS
(NQUAD_POINTS_1D);

// Getting gauss points and gauss weights in 1D
gauss_quadrature(NQUAD_POINTS_1D, POINTS, WEIGHTS);

// Getting gauss points and gauss weights in 2D
const int NQUAD_POINTS = NQUAD_POINTS_1D * NQUAD_POINTS_1D;
int COUNTER = 0; Vec QUAD_WEIGHTS (NQUAD_POINTS); Mat QUAD_POINTS (2,
NQUAD_POINTS);

for (int I = 0; I<NQUAD_POINTS_1D; I++){
for (int J = 0; J<NQUAD_POINTS_1D; J++){
QUAD_WEIGHTS[COUNTER] = WEIGHTS[I]*WEIGHTS[J];
QUAD_POINTS[0][COUNTER] = POINTS[J];
QUAD_POINTS[1][COUNTER] = POINTS[I];
COUNTER++;
} }

// PRINCIPLE OF VIRTUAL WORK
Mat x_GLOBAL_OLD (threeNODESTOTAL, NITER_MAX);
Mat K_GLOBAL (threeNODESTOTAL, threeNODESTOTAL); Vec g_GLOBAL (threeNODESTOTAL);
Vec ELEM_NODES (4); Mat x_ELEM (4,3); Mat X_ELEM (4,3); double XI_1; double XI_2;
Vec SHAPEFN (4); Vec DER_SHAPEFN_XI_1 (4); Vec DER_SHAPEFN_XI_2 (4);
```

```
Mat DER_SHAPEFN_XI (2,4); Mat DER_SHAPEFN_S (2,4); Mat DER_x_S (2,3); Mat DER_x_XI (2,3);
Vec X_VEC (3); Vec x_VEC (3); Vec G_1 (3); Vec G_2 (3);
Mat G_MAT (2,3); Mat CMAT (2,2); double norm_G_1; double norm_cross_G1_G2;
Vec cross_G1_G2 (3);Vec IHAT_1 (3); Vec IHAT_N (3); Vec IHAT_2 (3); Mat I_MAT (2,3);
Mat JACOBIAN (2,2);      double det_JACOBIAN; Mat INV_JACOBIAN (2,2); double I_1; double I_2;
Mat IMAT (2,2); Mat I1DER_CMAT (2,2); Mat I2DER_CMAT (2,2); Mat W_DER (2,2);
Mat DNa (3,12); Mat DNb (3,12); Mat DNc (3,12); Mat DNd (3,12); Mat SPK_STRESS (2,2);
Vec DCab (12); Vec DCcd (12); Vec GWE (12); Vec GPE (12); Vec g_ELEM (12);  Vec ZZ (4);
Mat DDCab (12, 12); Mat KM1E (12, 12); Mat KM2E (12, 12); Mat KGE (12, 12); Mat KPE (12,12);
Mat DN1 (3,12); Mat DN2 (3,12); Mat GME (12, 12); Mat K_ELEM (12, 12); Mat m_ELEM (12, 12);
Mat FULL_SHAPE (3,12); Mat DN (2,4); Mat DLN (2,4); Mat DI1 (2,2); Mat DI2 (2,2);
Vec cross_DER_x_XI1_XI2 (3); Vec DC11 (12); Vec DC22 (12); Vec traceDC (12);
Mat skew1 (3, 3); Mat skew2 (3, 3); Mat temp_skew (3,3); Mat temp_skew_t (3,3);
Mat DLN1 (3, 12); Mat DLN2 (3, 12); Mat skew1_DLN2 (3, 12); Mat skew2_DLN1(3, 12);

// Outside Pressure Loop
for (int PRESSURE_COUNTER = 0; PRESSURE_COUNTER<pressureLength;
PRESSURE_COUNTER++){
double CURRENT_PRESSURE = PRESSURE[PRESSURE_COUNTER]*INPUT_PRESSURE;

// Beginning Newton-Raphson
for (int I = 0; I < threeNODESTOTAL; I++){
for (int J = 0; J < NITER_MAX; J++){
x_GLOBAL_OLD[I][J] = 0;
} }

for (ITER_COUNTER = 0; ITER_COUNTER<NITER_MAX; ITER_COUNTER++){
FLAG_CONV = 0;  // Remains 0 until convergence is obtained

// For each element do the following
for (int I = 0; I < threeNODESTOTAL; I++){
g_GLOBAL[I] = 0;
for (int J = 0; J < threeNODESTOTAL; J++){
K_GLOBAL[I][J] = 0;
} }

for (int CURRENT_ELEM=0; CURRENT_ELEM<NELEMS_TOTAL; CURRENT_ELEM++){

for (int I=0; I<4; I++){
ELEM_NODES[I] = POINTER_MATRIX[CURRENT_ELEM][I];
for (int J=0; J<3; J++){
x_ELEM[I][J] = x_GLOBAL[ELEM_NODES[I]-1][J];
X_ELEM[I][J] = X_GLOBAL[ELEM_NODES[I]-1][J];
} }

for (int I= 0; I<12; I++){
g_ELEM[I] = 0;
for (int J=0; J<12; J++){
K_ELEM[I][J] = 0;
m_ELEM[I][J] = 0;
} }

for (int I=0; I<2; I++){
```

```
for (int J=0; J<2; J++){
SPK_STRESS[I][J] = 0;
} }

for (int QUAD_COUNTER=0; QUAD_COUNTER < NQUAD_POINTS; QUAD_COUNTER++){
// Getting elementwise convecting coordinates

XI_1 = QUAD_POINTS[0][QUAD_COUNTER];
XI_2 = QUAD_POINTS[1][QUAD_COUNTER];

// Shape function vector [N_1(XI_1,XI_2), N_2(XI_1,XI_2), N_3(XI_1,XI_2), N_4(XI_1,XI_2)]
shape_function(XI_1,XI_2, SHAPEFN);

// Derivative of Shape function matrix w.r.t XI_1, XI_2 (Size: 4 x 2)
shape_function_der(XI_1,XI_2, DER_SHAPEFN_XI);

// Bilinear approximation
for (int I=0; I<3; I++){
X_VEC[I] = 0; x_VEC[I]=0;
for (int J=0; J<4; J++){
X_VEC[I] = X_VEC[I] + (SHAPEFN[J] * X_ELEM[J][I]);
x_VEC[I] = x_VEC[I] + (SHAPEFN[J] * x_ELEM[J][I]);
} }

//Curvilinear basis (Undeformed)
for (int I=0; I<3; I++){
G_1[I] = 0; G_2[I] = 0;
for (int J=0; J<4; J++){
G_1[I] = G_1[I] + (DER_SHAPEFN_XI[0][J] * X_ELEM[J][I]);
G_2[I] = G_2[I] + (DER_SHAPEFN_XI[1][J] * X_ELEM[J][I]);
} }

for (int I=0; I<3; I++){
G_MAT[0][I] = G_1[I];
G_MAT[1][I] = G_2[I];
}

//Calculating norm(G_1)
norm_G_1 = sqrt(G_1[0]*G_1[0] + G_1[1]*G_1[1] + G_1[2]*G_1[2]);

//Calculating G_1 x G_2 (cross product)
cross_G1_G2[0] = (G_1[1]*G_2[2])-(G_1[2]*G_2[1]);
cross_G1_G2[1] = -1*((G_1[0]*G_2[2])-(G_1[2]*G_2[0]));
cross_G1_G2[2] = (G_1[0]*G_2[1])-(G_1[1]*G_2[0]);

//Calculating norm(G_1 x G_2)
norm_cross_G1_G2 = sqrt(cross_G1_G2[0]*cross_G1_G2[0] + cross_G1_G2[1]*cross_G1_G2[1] +
cross_G1_G2[2]*cross_G1_G2[2]);

for (int I=0; I<3; I++){
IHAT_1[I] = G_1[I] / norm_G_1;
}
```

```
for (int I=0; I<3; I++){
IHAT_N[I] = cross_G1_G2[I] / norm_cross_G1_G2;
}

//Calculating I_HAT2 (IHAT_N x IHAT_1) (cross product)
IHAT_2[0] = (IHAT_N[1]*IHAT_1[2])-(IHAT_N[2]*IHAT_1[1]);
IHAT_2[1] = -1*((IHAT_N[0]*IHAT_1[2])-(IHAT_N[2]*IHAT_1[0]));
IHAT_2[2] = (IHAT_N[0]*IHAT_1[1])-(IHAT_N[1]*IHAT_1[0]);

for (int I=0; I<3; I++){
I_MAT[0][I] = IHAT_1[I];
I_MAT[1][I] = IHAT_2[I];
}


for (int I = 0; I<2; I++){
JACOBIAN[I][0] = 0; JACOBIAN[I][1] = 0;
for (int J = 0; J<2; J++){
JACOBIAN[I][0] = JACOBIAN[I][0] + (G_MAT[I][J]*I_MAT[0][J]);
JACOBIAN[I][1] = JACOBIAN[I][1] + (G_MAT[I][J]*I_MAT[1][J]);
} }

det_JACOBIAN = (JACOBIAN[0][0]*JACOBIAN[1][1] - JACOBIAN[0][1]*JACOBIAN[1][0]);

INV_JACOBIAN[0][0] = (1/det_JACOBIAN) * JACOBIAN[1][1];
INV_JACOBIAN[0][1] = -1*(1/det_JACOBIAN) * JACOBIAN[0][1];
INV_JACOBIAN[1][0] = -1*(1/det_JACOBIAN) * JACOBIAN[1][0];
INV_JACOBIAN[1][1] = (1/det_JACOBIAN) * JACOBIAN[0][0];

for (int I=0; I<4; I++){
DER_SHAPEFN_S[0][I] = 0; DER_SHAPEFN_S[1][I] = 0;
for(int J=0; J<2; J++){
DER_SHAPEFN_S[0][I] = DER_SHAPEFN_S[0][I] +
INV_JACOBIAN[0][J]*DER_SHAPEFN_XI[J][I];
DER_SHAPEFN_S[1][I] = DER_SHAPEFN_S[1][I] +
INV_JACOBIAN[1][J]*DER_SHAPEFN_XI[J][I];
} }

for (int I=0; I<3; I++){
DER_x_S[0][I] = 0; DER_x_S[1][I] = 0;
for(int J=0; J<4; J++){
DER_x_S[0][I] = DER_x_S[0][I] + DER_SHAPEFN_S[0][J]*x_ELEM[J][I];
DER_x_S[1][I] = DER_x_S[1][I] + DER_SHAPEFN_S[1][J]*x_ELEM[J][I];
} }

for (int I=0; I<3; I++){
DER_x_XI[0][I] = 0; DER_x_XI[1][I] = 0;
for(int J=0; J<4; J++){
DER_x_XI[0][I] = DER_x_XI[0][I] + DER_SHAPEFN_XI[0][J]*x_ELEM[J][I];
DER_x_XI[1][I] = DER_x_XI[1][I] + DER_SHAPEFN_XI[1][J]*x_ELEM[J][I];
} }

for(int I=0; I<2; I++){
```

```
CMAT[0][I] = 0; CMAT[1][I] = 0;
for(int J=0; J<3; J++){
CMAT[0][I] = CMAT[0][I] + DER_x_S[0][J]*DER_x_S[I][J];
CMAT[1][I] = CMAT[1][I] + DER_x_S[1][J]*DER_x_S[I][J];
} }

I_1 = CMAT[0][0] + CMAT[1][1];
I_2 = CMAT[0][0]*CMAT[1][1] - CMAT[0][1]*CMAT[1][0];

double C00, C01, C10, C11;
double WDER_I1, WDER_I2, W2DER_I1_I1, W2DER_I1_I2, W2DER_I2_I2;
double IBAR_1 = I_1+1/I_2;

if (MATERIAL <=3){
C00 = 0;
if (MATERIAL==1){  //Neo-Hookean
C10 = 1; C01 = 0; C11 = 0;
}
else if (MATERIAL==2){     //Mooney-Rivlin
C10 = 1; C11 = 0; C01 = .2;
}
else if (MATERIAL==3){      //Rivlin Polynomial
C10 = 9.4; C01=.2; C11 = 82;
}

// First Derivatives for 2D Mooney-Rivlin Material
WDER_I1 = C10+ C01/I_2 + C11*(I_2 - 3 + 2*I_1*1/I_2 + 1/(I_2*I_2) - 3/I_2);

WDER_I2 = -C10 / (I_2*I_2) + C01 * (1- I_1 / (I_2 * I_2)) + C11*(I_1 - (I_1*I_1)/(I_2*I_2) -
(2*I_1)/(I_2*I_2*I_2) + 3*(1/(I_2*I_2) - 1 + I_1/(I_2*I_2)));

// Second Derivatives for 2D Mooney-Rivlin Material
W2DER_I1_I1 = 2*C11*(2/(I_2));
W2DER_I1_I2 = - C01 / (I_2 * I_2) + C11 * (1 -2*(I_1/(I_2*I_2) )+ 1/(I_2*I_2*I_2) + 3/(I_2*I_2));

W2DER_I2_I2 = 2*C10 / (I_2*I_2*I_2) + 2* C01 * I_1/(I_2*I_2*I_2) + C11*(2*I_1*I_1/(I_2*I_2*I_2)
+6*I_1/(I_2*I_2*I_2*I_2) - 6*(1/(I_2*I_2*I_2) + I_1/(I_2*I_2*I_2)));
}

double mu_ab, N;
if (MATERIAL==4){ //Arruda-Boyce

mu_ab = 1;
N = 2;

// First Derivatives for 2D Arruda-Boyce Material
WDER_I1 = mu_ab * (.5+ 2*(IBAR_1)/(20*N) + 33*(IBAR_1)*(IBAR_1)/(1050*N*N)+
76*(IBAR_1*IBAR_1*IBAR_1)/(7000*N*N*N) +
2595*(IBAR_1*IBAR_1*IBAR_1*IBAR_1)/(673750*N*N*N*N));

WDER_I2 = mu_ab * (-.5/(I_2*I_2) - 2/(I_2*I_2)*(IBAR_1)/(20*N) -
33/(I_2*I_2)*(IBAR_1*IBAR_1)/(1050*N*N)-
```

```
76/(I_2*I_2)*(IBAR_1*IBAR_1*IBAR_1)/(7000*N*N*N) -
2595/(I_2*I_2)*(IBAR_1*IBAR_1*IBAR_1*IBAR_1)/(673750*N*N*N*N*N));

// Second Derivatives for 2D Arruda-Boyce Material
W2DER_I1_I1 = mu_ab * (2/(20*N) + 66*(IBAR_1)/(1050*N*N) +
228*(IBAR_1*IBAR_1)/(7000*N*N*N) + 10380*(IBAR_1*IBAR_1*IBAR_1)/(673750*N*N*N*N*N));

W2DER_I1_I2 = mu_ab * (-2/(I_2*I_2)*(IBAR_1)/(20*N) - 66/(I_2*I_2)*(IBAR_1)/(1050*N*N) -
228/(I_2*I_2)*(IBAR_1*IBAR_1)/(7000*N*N*N) -
10380/(I_2*I_2)*(IBAR_1*IBAR_1*IBAR_1)/(673750*N*N*N*N*N));

W2DER_I2_I2 = mu_ab * (1/(I_2*I_2*I_2) + (4*I_1/(I_2*I_2*I_2) + 6/(I_2*I_2*I_2*I_2))/(20*N) +
(66/(I_2*I_2*I_2)*(IBAR_1*IBAR_1) + 66/(I_2*I_2*I_2*I_2)*(IBAR_1))/(1050*N*N) +
(152/(I_2*I_2)*(IBAR_1*IBAR_1*IBAR_1) +
228/(I_2*I_2*I_2*I_2*I_2)*(IBAR_1*IBAR_1))/(7000*N*N*N) +
(5190/(I_2*I_2*I_2)*(IBAR_1*IBAR_1*IBAR_1*IBAR_1) +
10380/(I_2*I_2*I_2*I_2*I_2)*(IBAR_1*IBAR_1*IBAR_1))/(673750*N*N*N*N*N));
}

double C1, C2, C3;
if (MATERIAL==5){ //Yeoh

C1 = .634; C2 = .066; C3 = .021;

// First Derivatives for 2D Yeoh Material
WDER_I1 = C1+ 2*C2*(IBAR_1-1) + 3*C3*(IBAR_1-1)*(IBAR_1-1);
WDER_I2 = -1*C1/(I_2*I_2) - 2*C2/(I_2*I_2)*(IBAR_1-1) - 3*C3/(I_2*I_2)*(IBAR_1-1)*(IBAR_1-1);
//
// Second Derivatives for 2D Yeoh Material
W2DER_I1_I1 = 2*C2 + 6*C3*(IBAR_1-1);
W2DER_I1_I2 = -2*C2/(I_2*I_2) -6*C3/(I_2*I_2)*(IBAR_1-1);
W2DER_I2_I2 = 2*C1/(I_2*I_2*I_2)+4*C2/(I_2*I_2)*(I_1 - 1) + 6*C2 * 1/(I_2*I_2*I_2*I_2)
+12*C3/(I_2*I_2*I_2)*(IBAR_1-1)*(IBAR_1-1) + 6*C3/(I_2*I_2*I_2*I_2)*(IBAR_1-1);
}
double w11=W2DER_I1_I1;          double w12=W2DER_I1_I2;          double w22=W2DER_I2_I2;

for (int I=0; I<2; I++){
for (int J=0; J<2; J++){
if (I==J){
IMAT[I][J] = 1;
}
else{
IMAT[I][J] = 0;
} } }

for (int I=0; I<2; I++){
for (int J=0; J<2; J++){
// Derivatives dI_i/dC
I1DER_CMAT[I][J] = IMAT[I][J];
I2DER_CMAT[I][J] = I_1 * IMAT[I][J] - CMAT[I][J];

//Derivative dw/dC = (dw/dI_i)(dI_i/dC)
W_DER[I][J] = WDER_I1 * I1DER_CMAT[I][J] + WDER_I2 * I2DER_CMAT[I][J];
```

```
SPK_STRESS[I][J] = SPK_STRESS[I][J] +2*W_DER[I][J];
} }

for (int I= 0; I<12; I++){
GWE[I] = 0;
GPE[I] = 0;
for (int J=0; J<12; J++){
KGE[I][J] = 0;
KPE[I][J] = 0;
GME[I][J] = 0;
KM1E[I][J] = 0;
KM2E[I][J] = 0;
} }

for (int I=0; I<4; I++){
ZZ[I] = 0;
}

int fullshapeLength = 0;
while (fullshapeLength < 12){
while (fullshapeLength < 8){
while (fullshapeLength < 4){

FULL_SHAPE[0][fullshapeLength] = SHAPEFN[fullshapeLength];
FULL_SHAPE[1][fullshapeLength] = ZZ[fullshapeLength];
FULL_SHAPE[2][fullshapeLength] = ZZ[fullshapeLength];
fullshapeLength++;
}

FULL_SHAPE[0][fullshapeLength] = ZZ[fullshapeLength-4];
FULL_SHAPE[1][fullshapeLength] = SHAPEFN[fullshapeLength-4];
FULL_SHAPE[2][fullshapeLength] = ZZ[fullshapeLength-4];
fullshapeLength++;
}

FULL_SHAPE[0][fullshapeLength] = ZZ[fullshapeLength-8];
FULL_SHAPE[1][fullshapeLength] = ZZ[fullshapeLength-8];
FULL_SHAPE[2][fullshapeLength] = SHAPEFN[fullshapeLength-8];
fullshapeLength++;
}

for (int I=0; I<2; I++){
for (int J=0; J<4; J++){
DN[I][J] = DER_SHAPEFN_S[I][J];     // Renaming dN/ds
DLN[I][J] = DER_SHAPEFN_XI[I][J];   // Renaming dN/dxi
} }

for (int I=0; I<2; I++){
for (int J=0; J<2; J++){
DI1[I][J] = I1DER_CMAT[I][J];
DI2[I][J] = I2DER_CMAT[I][J];
} }
```

```
//Calculating DER_x_XI(0,:) x DER_x_XI(1,:) (cross product)
cross_DER_x_XI1_XI2[0] = (DER_x_XI[0][1]*DER_x_XI[1][2])-(DER_x_XI[1][1]*DER_x_XI[0][2]);
cross_DER_x_XI1_XI2[1] = -1*((DER_x_XI[0][0]*DER_x_XI[1][2])-
(DER_x_XI[1][0]*DER_x_XI[0][2]));
cross_DER_x_XI1_XI2[2] = (DER_x_XI[0][0]*DER_x_XI[1][1])-(DER_x_XI[1][0]*DER_x_XI[0][1]);

for (int I=0; I<12; I++){
for(int J=0; J<3; J++){
GPE[I] = GPE[I] + (-1 * CURRENT_PRESSURE * FULL_SHAPE[J][I]*cross_DER_x_XI1_XI2[J] *
QUAD_WEIGHTS[QUAD_COUNTER]);
} }

int dlnLength = 0;
while (dlnLength < 12){
while (dlnLength < 8){
while (dlnLength < 4){

DN1[0][dlnLength] = DN[0][dlnLength];
DN1[1][dlnLength] = ZZ[dlnLength];
DN1[2][dlnLength] = ZZ[dlnLength];

DN2[0][dlnLength] = DN[1][dlnLength];
DN2[1][dlnLength] = ZZ[dlnLength];
DN2[2][dlnLength] = ZZ[dlnLength];

DLN1[0][dlnLength] = DLN[0][dlnLength];
DLN1[1][dlnLength] = ZZ[dlnLength];
DLN1[2][dlnLength] = ZZ[dlnLength];

DLN2[0][dlnLength] = DLN[1][dlnLength];
DLN2[1][dlnLength] = ZZ[dlnLength];
DLN2[2][dlnLength] = ZZ[dlnLength];
dlnLength++;
}

DN1[0][dlnLength] = ZZ[dlnLength-4];
DN1[1][dlnLength] = DN[0][dlnLength-4];
DN1[2][dlnLength] = ZZ[dlnLength-4];

DN2[0][dlnLength] = ZZ[dlnLength-4];
DN2[1][dlnLength] = DN[1][dlnLength-4];
DN2[2][dlnLength] = ZZ[dlnLength-4];

DLN1[0][dlnLength] = ZZ[dlnLength-4];
DLN1[1][dlnLength] = DLN[0][dlnLength-4];
DLN1[2][dlnLength] = ZZ[dlnLength-4];

DLN2[0][dlnLength] = ZZ[dlnLength-4];
DLN2[1][dlnLength] = DLN[1][dlnLength-4];
DLN2[2][dlnLength] = ZZ[dlnLength-4];
dlnLength++;
}
```

```
DN1[0][dlnLength] = ZZ[dlnLength-8];
DN1[1][dlnLength] = ZZ[dlnLength-8];
DN1[2][dlnLength] = DN[0][dlnLength-8];

DN2[0][dlnLength] = ZZ[dlnLength-8];
DN2[1][dlnLength] = ZZ[dlnLength-8];
DN2[2][dlnLength] = DN[1][dlnLength-8];

DLN1[0][dlnLength] = ZZ[dlnLength-8];
DLN1[1][dlnLength] = ZZ[dlnLength-8];
DLN1[2][dlnLength] = DLN[0][dlnLength-8];

DLN2[0][dlnLength] = ZZ[dlnLength-8];
DLN2[1][dlnLength] = ZZ[dlnLength-8];
DLN2[2][dlnLength] = DLN[1][dlnLength-8];
dlnLength++;
}

//create skew1 skew2 DLN1 DLN2
for (int ii = 0; ii<2; ii++){
temp_skew[0][0] = 0; temp_skew[0][1] = -1*DER_x_XI[ii][2]; temp_skew[0][2] = DER_x_XI[ii][1];
temp_skew[1][0] = 0; temp_skew[1][1] = 0; temp_skew[1][2] = -1*DER_x_XI[ii][0];
temp_skew[2][0] = 0; temp_skew[2][1] = 0; temp_skew[2][2] = 0;

for (int I = 0; I<3; I++){
temp_skew_t[0][I] = temp_skew[I][0];
temp_skew_t[1][I] = temp_skew[I][1];
temp_skew_t[2][I] = temp_skew[I][2];
}

for (int I = 0; I<3; I++){
for (int J = 0; J<3; J++){
if (ii==0){
skew1[I][J] = temp_skew[I][J] - temp_skew_t[I][J];
}
if (ii==1){
skew2[I][J] = temp_skew[I][J] - temp_skew_t[I][J];
} } } }

for (int I=0; I<3; I++){
for (int J=0; J<12; J++){
skew1_DLN2[I][J] = 0;
skew2_DLN1[I][J] = 0;
} }

for (int I=0; I<12; I++){
for (int J=0; J<3; J++){
skew1_DLN2[0][I] = skew1_DLN2[0][I] + skew1[0][J]*DLN2[J][I];
skew1_DLN2[1][I] = skew1_DLN2[1][I] + skew1[1][J]*DLN2[J][I];
skew1_DLN2[2][I] = skew1_DLN2[2][I] + skew1[2][J]*DLN2[J][I];

skew2_DLN1[0][I] = skew2_DLN1[0][I] + skew2[0][J]*DLN1[J][I];
skew2_DLN1[1][I] = skew2_DLN1[1][I] + skew2[1][J]*DLN1[J][I];
```

```
skew2_DLN1[2][I] = skew2_DLN1[2][I] + skew2[2][J]*DLN1[J][I];
} }

for (int I=0; I<12; I++){
for(int J=0; J<12; J++){
for (int K=0; K<3; K++){
KPE[I][J] = KPE[I][J] + (-1 * CURRENT_PRESSURE * FULL_SHAPE[K][I]*(skew1_DLN2[K][J]-
skew2_DLN1[K][J]) * QUAD_WEIGHTS[QUAD_COUNTER]);
} } }

for (int I=0; I<12; I++){
DC11[I] = 0; DC22[I] = 0;
for (int J=0; J<3; J++){
DC11[I] =  DC11[I] + 2*DER_x_S[0][J] * DN1[J][I];                    // 1x12 = 1x3 3x12
DC22[I] =  DC22[I] + 2*DER_x_S[1][J] * DN2[J][I];
}
traceDC[I] = DC11[I] + DC22[I];
}

int aaLength; int bbLength; int ccLength; int ddLength;
for (int aa =0; aa<2; aa++){
aaLength = 0;
while (aaLength < 12){
while (aaLength < 8){
while (aaLength < 4){
DNa[0][aaLength] = DN[aa][aaLength];
DNa[1][aaLength] = ZZ[aaLength];
DNa[2][aaLength] = ZZ[aaLength];
aaLength++;
}
DNa[0][aaLength] = ZZ[aaLength-4];
DNa[1][aaLength] = DN[aa][aaLength-4];
DNa[2][aaLength] = ZZ[aaLength-4];
aaLength++;
}
DNa[0][aaLength] = ZZ[aaLength-8];
DNa[1][aaLength] = ZZ[aaLength-8];
DNa[2][aaLength] = DN[aa][aaLength-8];
aaLength++;
}

for (int bb =0; bb<2; bb++){
bbLength = 0;
while (bbLength < 12){
while (bbLength < 8){
while (bbLength < 4){
DNb[0][bbLength] = DN[bb][bbLength];
DNb[1][bbLength] = ZZ[bbLength];
DNb[2][bbLength] = ZZ[bbLength];
bbLength++;
}
DNb[0][bbLength] = ZZ[bbLength-4];
DNb[1][bbLength] = DN[bb][bbLength-4];
```

```
DNb[2][bbLength] = ZZ[bbLength-4];
bbLength++;
}
DNb[0][bbLength] = ZZ[bbLength-8];
DNb[1][bbLength] = ZZ[bbLength-8];
DNb[2][bbLength] = DN[bb][bbLength-8];
bbLength++;
}

for (int I=0; I<12; I++){
DCab[I] = 0;
for (int J=0; J<3; J++){
DCab[I] = DCab[I] + (DER_x_S[aa][J] * DNb[J][I] + DER_x_S[bb][J] * DNa[J][I]);        //1x12
} }

for (int I=0; I<12; I++){
for(int J=0; J<12; J++){
DDCab[I][J] = 0;
for (int K=0; K<3; K++){
DDCab[I][J] = DDCab[I][J] + (DNa[K][I] * DNb[K][J] + DNa[K][I] * DNb[K][J]);
} } }

for (int I=0; I<12; I++){
GWE[I] = GWE[I] + W_DER[aa][bb] * DCab[I];
for (int J=0; J<12; J++){
KGE[I][J] = KGE[I][J] + W_DER[aa][bb] * DDCab[I][J];
KM2E[I][J]= KM2E[I][J] + (DCab[I] * WDER_I2 * (traceDC[J] * IMAT[aa][bb] - DCab[J]));
} }

for (int cc =0; cc<2; cc++){
ccLength = 0;
while (ccLength < 12){
while (ccLength < 8){
while (ccLength < 4){
DNc[0][ccLength] = DN[cc][ccLength];
DNc[1][ccLength] = ZZ[ccLength];
DNc[2][ccLength] = ZZ[ccLength];
ccLength++;
}
DNc[0][ccLength] = ZZ[ccLength-4];
DNc[1][ccLength] = DN[cc][ccLength-4];
DNc[2][ccLength] = ZZ[ccLength-4];
ccLength++;
}
DNc[0][ccLength] = ZZ[ccLength-8];
DNc[1][ccLength] = ZZ[ccLength-8];
DNc[2][ccLength] = DN[cc][ccLength-8];
ccLength++;
}
for (int dd =0; dd<2; dd++){
ddLength = 0;
while (ddLength < 12){
while (ddLength < 8){
```

```
while (ddLength < 4){
DNd[0][ddLength] = DN[dd][ddLength];
DNd[1][ddLength] = ZZ[ddLength];
DNd[2][ddLength] = ZZ[ddLength];
ddLength++;
}
DNd[0][ddLength] = ZZ[ddLength-4];
DNd[1][ddLength] = DN[dd][ddLength-4];
DNd[2][ddLength] = ZZ[ddLength-4];
ddLength++;
}
DNd[0][ddLength] = ZZ[ddLength-8];
DNd[1][ddLength] = ZZ[ddLength-8];
DNd[2][ddLength] = DN[dd][ddLength-8];
ddLength++;
}

for (int I=0; I<12; I++){
DCcd[I] = 0;
for (int J=0; J<3; J++){
DCcd[I] = DCcd[I] + (DER_x_S[cc][J] * DNd[J][I] + DER_x_S[dd][J] * DNc[J][I]);
} }

double temp1 = w11 * DI1[cc][dd] + w12 * DI2[cc][dd];
double temp2 = w12 * DI1[cc][dd] + w22 * DI2[cc][dd];
double temp = DI1[aa][bb]* temp1 +  DI2[aa][bb]* temp2;

for (int I=0; I<12; I++){
for (int J=0; J<12; J++){
KM1E[I][J]= KM1E[I][J] + (DCab[I] * temp * DCcd[J]);
} } } } } }

for (int I=0; I<12; I++){
for (int J=0; J<12; J++){
GME[I][J] = 0;
for (int K=0; K<3; K++){
GME[I][J] = GME[I][J] + FULL_SHAPE[K][I] * FULL_SHAPE[K][J] * norm_cross_G1_G2;
} } }

for (int I=0; I<12; I++){
g_ELEM[I] = g_ELEM[I] + GWE[I] * norm_cross_G1_G2 * QUAD_WEIGHTS[QUAD_COUNTER] +
GPE[I];
for (int J=0; J<12; J++){
K_ELEM[I][J] = K_ELEM[I][J] + (KM1E[I][J]+KM2E[I][J]+KGE[I][J]) * norm_cross_G1_G2 *
QUAD_WEIGHTS[QUAD_COUNTER] + KPE[I][J];
m_ELEM[I][J] = m_ELEM[I][J] + GME[I][J] * QUAD_WEIGHTS[QUAD_COUNTER];
} } }  // QUAD_COUNTER

// ASSEMBLY
Vec POSITION (12);

for (int I=0; I<4; I++){
POSITION[I] = ELEM_NODES[I];
```

```
POSITION[I+4] = ELEM_NODES[I] + NNODES_TOTAL;
POSITION[I+8] = ELEM_NODES[I] + 2*NNODES_TOTAL;
}

for (int I = 0; I<12; I++){
g_GLOBAL[POSITION[I]-1] = g_GLOBAL[POSITION[I]-1] + g_ELEM[I];
for (int J=0; J<12; J++){
K_GLOBAL[POSITION[I]-1][POSITION[J]-1] = K_GLOBAL[POSITION[I]-1][POSITION[J]-1] +
K_ELEM[I][J];
} } } // for ELEM_COUNT

// APPLY BOUNDARY CONDITIONS
Mat K1_GLOBAL (movingnodesLength, 3*NNODES_TOTAL);
Mat K2_GLOBAL (movingnodesLength, movingnodesLength);
Mat K2_GLOBAL_LU (movingnodesLength, movingnodesLength);
Vec g1_GLOBAL (movingnodesLength); Vec K2_div_g1 (movingnodesLength);

for (int I=0; I<movingnodesLength; I++){
g1_GLOBAL[I] = g_GLOBAL[MOVING_NODES[I]-1];
for (int K=0; K<3*NNODES_TOTAL; K++){
K1_GLOBAL[I][K] = K_GLOBAL[MOVING_NODES[I]-1][K];
} }

for (int K=0; K<movingnodesLength; K++){
for (int I=0; I<movingnodesLength; I++){
K2_GLOBAL[I][K] = K1_GLOBAL[I][MOVING_NODES[K]-1];
} }

// COMPUTING SOLUTION
Vec indx (movingnodesLength);

for (int I=0; I<movingnodesLength; I++){
K2_div_g1[I] = g1_GLOBAL[I];
for (int J=0; J<movingnodesLength; J++){
K2_GLOBAL_LU[I][J] = K2_GLOBAL[I][J];
} }

ludcmp(K2_GLOBAL_LU,movingnodesLength, indx);
lubksb(K2_GLOBAL_LU,movingnodesLength,K2_div_g1, indx);

for (int I=0; I<NNODES_TOTAL; I++){
x_GLOBAL_OLD[I][ITER_COUNTER] = x_GLOBAL[I][0];
x_GLOBAL_OLD[I+NNODES_TOTAL][ITER_COUNTER] = x_GLOBAL[I][1];
x_GLOBAL_OLD[I+2*NNODES_TOTAL][ITER_COUNTER] = x_GLOBAL[I][2];
}

for (int I=0; I<movingnodesLength; I++){
if (MOVING_NODES[I]-1 < NNODES_TOTAL-1){
x_GLOBAL[MOVING_NODES[I]-1][0] = x_GLOBAL[MOVING_NODES[I]-1][0] - K2_div_g1[I];
}
else if (MOVING_NODES[I]-1 < 2*NNODES_TOTAL-1){
x_GLOBAL[(MOVING_NODES[I]-NNODES_TOTAL)-1][1] = x_GLOBAL[(MOVING_NODES[I]-
NNODES_TOTAL)-1][1] - K2_div_g1[I];
```

```
}
else{
x_GLOBAL[(MOVING_NODES[I]-2*NNODES_TOTAL)-1][2] = x_GLOBAL[(MOVING_NODES[I]-
2*NNODES_TOTAL)-1][2] - K2_div_g1[I];
} }

cout << "z_GLOBAL(CENTER_NODE)= "<< x_GLOBAL[CENTER_NODE[0]][2] << endl;

// Find the Euclidean norm of the _previous_ residual vector

eucl_norm_res_vect = 0;
for (int I=0; I<movingnodesLength; I++){
eucl_norm_res_vect = eucl_norm_res_vect + g1_GLOBAL[I] * g1_GLOBAL[I];
}
eucl_norm_res_vect = sqrt(eucl_norm_res_vect);

cout << "eucl_norm_res_vect= " << eucl_norm_res_vect << endl;

for (int I = 0; I<3; I++){
for (int J = 0; J< NNODES_TOTAL; J++){
error_xg[I*NNODES_TOTAL+J] = abs(x_GLOBAL[J][I] -
x_GLOBAL_OLD[I*NNODES_TOTAL+J][ITER_COUNTER]);
} }

max_error_xg = error_xg[0];
for (int I=1; I<3*NNODES_TOTAL; I++){
if (error_xg[I] > max_error_xg){
max_error_xg = error_xg[I];
} }

cout << "Iteration number = " << ITER_COUNTER+1 << endl;
cout << "max_error_xg = " << max_error_xg << endl;

if (max_error_xg < TOLERANCE){
cout << "IT HAS CONVERGED" << endl;
FLAG_CONV=1;
}

if (FLAG_CONV==1){
break;
} }                    // for ITER_COUNTER

cout << "Pressure= " << CURRENT_PRESSURE << endl;
cout << "Center-node (x,y)= (" << x_GLOBAL[CENTER_NODE[0]][0] << ", " <<
x_GLOBAL[CENTER_NODE[0]][1] << ")" << endl;
cout << "Center-node Deflection= " << x_GLOBAL[CENTER_NODE[0]][2] << endl;
cout << "Number of iterations= " << ITER_COUNTER << endl;
cout << "Error= " << max_error_xg << endl;

}          // for PRESSURE_COUNTER
}

void gauss_quadrature(int N, vector<double> &A, vector<double> &W){
```

```
if (N==1){
A[0] = 0.0; W[0] = 2.0;
}
else if (N==2){
A[0] = 0.577350269189626;  A[1] = -A[0];  W[0] = 1.;  W[1] = 1.;
}
else if (N==3){
A[0] = 0.774596669241483;  A[1] = 0.;  A[2] = -A[0];
W[0] = 0.555555555555556;  W[1] = 0.888888888888889;  W[2] = W[0];
}
else if (N==4){
A[0] = 0.861136311594053;  A[1] = 0.339981043584856;  A[2] = -A[1];
A[3] = -A[0];  W[0] = 0.347854845137454;  W[1] = 0.652145154862546;
W[2] = W[1];  W[3] = W[0];
}
else if (N==5){
A[0] = 0.906179845938664;  A[1] = 0.538469310105683;  A[2] = 0.0;
A[3] = -A[1];  A[4] = -A[0];  W[0] = 0.236926885056189;
W[1] = 0.478628670499366;  W[2] = 0.568888888888889;  W[3] = W[1];
W[4] = W[0];
}
else if (N==6){
A[0] = 0.932469514203152;  A[1] = 0.661209386466265;
A[2] = 0.238619186083197;
A[3] = -A[2];  A[4] = -A[1];  A[5] = -A[0];  W[0] = 0.171324492379170;
W[1] = 0.360761573048139;  W[2] = 0.467913934572691;  W[3] = W[2];
W[4] = W[1];  W[5] = W[0];
} }

void shape_function(double r, double s, vector<double> &shp){
double rp = 1. + r; double rm = 1. - r;
double sp = 1. + s; double sm = 1. - s;
shp[0] = 0.25*rm*sm; shp[1] = 0.25*rp*sm;
shp[2] = 0.25*rp*sp; shp[3] = 0.25*rm*sp;
}

void shape_function_der(double r, double s, vector<vector<double>> &d){

double rp = 1. + r; double rm = 1. - r; double sp = 1. + s; double sm = 1. - s;
d[0][0] = -0.25*sm; d[0][1]=  0.25*sm; d[0][2] =  0.25*sp; d[0][3] = -0.25*sp;
d[1][0] = -0.25*rm; d[1][1] = -0.25*rp; d[1][2] =  0.25*rp; d[1][3] =  0.25*rm;
}

void ludcmp(vector<vector<double>>& a, int n, vector<double>& indx){
int I; int J; int K; int d=1; int NMAX = n; int imax;
double TINY = 1e-20; double aamax, dum, sum; Vec vv (NMAX);

for (I=0; I<n; I++){
aamax=0;
for (J=0;J<n;J++){
if (abs(a[I][J])>aamax){
aamax=abs(a[I][J]);
```

```
}
}
if (aamax == 0){
cout << "ERROR: Singular Matrix" << endl;
return;
}
vv[I] = 1/aamax;
}

for (K=0; K<n; K++){
aamax = 0;
for (I=K; I<n; I++){
dum=vv[I]*abs(a[I][K]);
if (dum>=aamax){
imax=I;
aamax=dum;
}
}

if (K != imax){
for (J=0; J<n; J++){
dum = a[imax][J];
a[imax][J] = a[K][J];
a[K][J] = dum;
}
d=-d;
vv[imax] = vv[K];
}
indx[K] = imax;

if(a[K][K] == 0){
a[K][K] == TINY;
}
for (I = K+1; I<n; I++){
dum=a[I][K] /= a[K][K];
for (J=K+1; J<n; J++){
a[I][J] -= dum*a[K][J];
} } } }

void lubksb(vector<vector<double>>& a, int n, vector<double>& b, vector<double>& indx){

int i,ii=0,ip,j; double sum;

for (i=0;i<n;i++) {
ip=indx[i];
sum=b[ip];
b[ip]=b[i];
if (ii){
for (j=ii-1;j<=i-1;j++){
sum -= a[i][j]*b[j];
} }
else if (sum){
ii=i+1;
```

```
}

b[i]=sum;
}
for (i=n-1;i>=0;i--) {
sum=b[i];
for (j=i+1;j<n;j++){
sum -= a[i][j]*b[j];
}
b[i]=sum/a[i][i];
} }

void sort (vector<double>& var, int size) {

int index = size - 2; int changeFlag = 1;

while (index >= 0 && changeFlag){
changeFlag = 0;
for (int I = 0; I <= index; I++) {
if (var[I] > var[I+1])
{
swap(var[I], var[I+1]) ;
changeFlag = 1;
} }
index--;
} }

// swap function for integers
void swap(int& x, int& y){
int temp = x;
x = y;
y = temp;
}
```

REFERENCES

REFERENCES

[1] Bower, A. F., *Applied Mechanics of Solids*, CRC Press, Florida, (2010).

[2] Duncan, B. C., Crocker, L. E., Urquhart, J. M. *Evaluation of Hyperelastic Finite Element Models for Flexible Adhesive Joints*, National Physical Laboratory, NPL Report CMMT(A)285, (2000).

[3] Ferguson, Lauren A., *A Computational Model for Flexible Wing Based Micro Air Vehicles*, M. S. Thesis, Department of Mathematics, Texas Tech University, Lubbock, TX, USA, (2006).

[4] Humphrey, J. D., *Cardiovascular Solid Mechanics: Cells, Tissues, and Organs*, Springer-Verlag, New York. (2002).

[5] Hung, A., Mithraratne, K., Sagar, M., Hunter, P., *Multilayer Soft Tissue Continuum Model: Towards Realistic Simulation of Facial Expression,*World Academy of Science, Engineering and Technology 54, (2009)

[6] Ifju, P., Jenkins, D., Ettinger, S., Lian, Y., Shyy, W. and Waszak, R.M., *Flexible-Wing-Based Micro Air Vehicles*, 40th AIAA Aerospace Sciences Meeting and Exhibit, AIAA Paper 2002-0705, (2002).

[7] Kundomal, C. A., *Computational Modeling of Membrane Mechanics*, M. S. Thesis, Department of Mathematics, Texas Tech University, Lubbock, TX, USA, (2006).

[8] Matlab – The Language of Technical Computing, The MathWorks, Inc., Available at http://www.mathworks.com/products/matlab/, Last updated: April 2011.

[9] McGee, W. M., *Three-Dimesional Mortar Finite Element Method for Convection-Diffusion Equation with Non-Conforming Meshes*, Masters' Thesis, Applied Mathematics, Texas Tech University, Lubbock, TX, USA, (2003).

[10] *Numerical Recipes in Fortran 77: The Art of Scientific Computing*, Cambridge University Press, 1992,

[11] Peiró J., Sherwin, S.,*Finite Difference, Finite Element and Finite Volume Methods for Partial Differential Equations*, Handbook of Materials Modeling. Volume I: Methods and Models, 1-32 (2005).

[12] Ramamurti, R., Sandberg, W., Vaiana, P., Kellogg, J., Cylinder, D., *Computational Fluid Dynamics Study of Unconventional Air Vehicle Configurations*, 19th Bristol International Conference on Unmanned Air Vehicle Systems (2004).

[13] Regaieg, A., Mars, J., Dammak, F., Dhieb, A., *Finite element analysis of nonlinear orthotropic hyperelastic membrane shell*, Unit of mechanical, modeling and manufacturing, (2007).

[14] Seshaiyer, P. *Computational Structural Methodolgy for Flexible Wing based Micro Air Vehicles* AFRL Internal Report. 2005.

[15] Stanford, B., Viieru, D., Albertani, R., Shyy, W., Ifju, P., *A Numerical and Experimental Investigation of Flexible Micro Air Vehicle Wing Deformation*, 44th AIAA Aerospace Sciences Meeting and Exhibit, Paper AIAA 2006-440, (2006).

[16] Stumpf, F.,Marczak, R., *Optimization of Constitutive Parameters for Hyperelastic Models Satisfying the Baker-Ericksen Inequalities*, Mecanica Computational, 29: 2901-2916 (2010).

[17] Virtual Work. Wikipedia, The Free Encyclopedia.Available at http://en.wikipedia.org/wiki/Virtual_work/. Last updated: 20 May 2011.

[18] Yang W. H., Lu, C. H., *General deformations of Neo-Hookean membranes*, Journal of Applied Mechanics, 40: 7-12. (1973).

CURRICULUM VITAE

Michael J. Garrity graduated from Mount Vernon High School, Alexandria, Virginia, in 2006. He received his Bachelor of Science in Mathematics from George Mason University in 2010. He is employed at Modern Technology Solutions Inc. (MTSI) as a Modeling and Simulation Engineer for over a year and an half and is expecting to receive his Master of Science in Mathematics from George Mason University in 2011.