The Adoption of a Finite Element Model of Material Deformation Relevant to Studying Corneal Biomechanics

A Thesis Submitted to the Committee on Graduate Studies
in Partial Fulfillment of the Requirements for the Degree of Master of Science
in the Faculty of Arts and Science

Trent University

Peterborough, Ontario, Canada

Materials Science M.Sc. Graduate Program

September 2020

ABSTRACT

The Adoption of a Finite Element Model of Material Deformation Relevant to Studying Corneal Biomechanics

By Yuchen Song

The human cornea is required to exhibit specific material properties to maintain its regular shape under typical intraocular pressures which then allow for its correct optical functionality. In this thesis, the basis of continuum solid mechanics and the finite element method are introduced. We use finite element modelling to simulate the extension of an effective-1d, linear-elastic bar, a cornea-like body governed by Poisson's equation, and the deformation of a loaded, linear-elastic, cube. Preliminary results for the deformation of a simulated, linear-elastic, cornea have also been achieved using the finite element approach.

Keywords: finite element method, corneal biomechanics, intraocular pressure, continuum solid mechanics

Acknowledgements

I would like to express my deepest gratitude to my supervisor, Professor Rayf Shiell, for his unwavering support, guidance, and words of encouragement throughout my M.Sc. Especially, I would like to thank him for his personal support during the challenges associated with the coronavirus pandemic.

I would also like to thank my supervisor committee members, Prof. Bill Atkinson and Prof. Hendrick de Haan, for providing their valuable feedback.

I would like to thank my parents, this thesis would not have been possible without their support.

Contents

List of Figures

List of Tables

Chapter 1

Literature review and motivation for studying corneal biomechanics

The cornea, the spherical layer that covers the front of the eyeball, has a remarkable transparent property that enables us to observe the universe around us. Nearly 2/3 of the human eye's refractive power stems from the cornea, and corneal geometry plays an important role in this refraction. For example, in the eye disease keratoconus a deformed cornea derived from thinning causes blurred vision [1]. A normal *intraocular pressure* (IOP), typically 10-20 mmHg above the pressure external to the body, and reasonable mechanical stiffness, are required to maintain the cornea in its regular shape.

Human corneal mechanical properties and dimensions vary between subjects, e.g. their ages, health, and hydration [2-5]. The mechanical properties of the cornea are of great interest and have been experimentally determined through different techniques, including strip extensions [6-8], elastography [9], ultrasound induced optical coherence tomography [10], air-puff methods [4] and inflation studies [11]. Under small strain, human corneal tissue behaves like a linear-elastic, nearly-incompressible, homogeneous material [12].

Corneal properties have been quantified *in vivo* from fitting material models with various material parameters to the experimental data. Montanino's group [13] combined an analytical, nonlinear-elastic, isotropic membrane model for the cornea with a meshfree (i.e. fluid) weakly compressible approach for the aqueous humor, and developed a

dynamic, isotropic, cylindrically-symmetric model to simulate the mechanical response of the cornea-aqueous humor system to air-puff tests. The simulation adopted a set of values for the elastic modulus, from 0.3 MPa to 1.5 MPa, and was compared with an air-puff test example. This study performed simulation similar to the actual experimental data and confirmed the importance of the aqueous humor in the air-puff tests. Shih's group [4] employed a static simulation to model the corneal extreme deformation due to an air-puff with the cornea treated as a homogenous, isotropic, nonlinear-elastic material in an initial half-spherical shell. The resulting elastic modulus was determined by fitting the experimental data from 536 right eyes and was determined to have a skewed distribution from 0.10-0.68 MPa with a mean value of 0.206 MPa. Kling's group [14] simulated corneal deformation during air-puff tests by using a viscoelastic model associated with a 2d cylindrically-symmetric bilayer geometry comprised of two isotropic materials to assess the elastic moduli of nine subjects' eyes, and resulted in a mean value of 0.45 MPa for the anterior layer and 0.40 MPa for the posterior layer, in addition to determining the viscoelastic parameters.

Corneal mechanical properties are more often assessed through *in vitro* studies, for example, through strip extension tests [7, 8, 15] and inflation tests [11, 15]. Under large strains the cornea exhibits a nonlinear deformation response, in which the elastic modulus increases as the stress increases [16, 17]. Elsheikh's group [11] performed inflation tests of an excised cornea with an inflation pressure from 0-75 mmHg, and by assuming an isotropic homogenous material, related the internal pressure and specimen dimensions to stress and strain, from which they determined that the resulting elastic modulus across this range increased from 0.15 to 0.43 MPa. Jayasuriya's group [8] investigated the

drying time-dependence on the elastic moduli of human cornea strip samples, in extension tests from fourteen corneas, with four cut horizontally (in the nasal-temporal direction), three vertically (in the superior-inferior direction), and seven diagonally. A dynamic stress, corresponding to a maximum force of 5 N and a frequency of 104 Hz, was applied to each strip as the sample dried for roughly forty minutes. The average elastic modulus (along each direction) was initially determined to be around 3 MPa for the horizontally sectioned samples, 1 MPa vertically, and 0.3 MPa diagonally.

Ex vivo experiments have also been performed to establish corneal biomechanical responses in many other species, most commonly including pig [7], bovine [7, 10, 11], and rabbit [6]. The porcine cornea exhibits a similar stress-strain behavior to the human cornea in strip extension tests and has comparable biomechanical properties [7].

The diameter of the human cornea is about 11 mm measured horizontally and 10 mm vertically [5]. It has a center thickness of about 550 μm [18] and is essentially a convex-concave negative aspheric lens, with an approximate radius of curvature of 7.7mm at the anterior surface and 6.8 mm at the posterior surface [19]. An air-cornea-aqueous system typically has a refractive power of 40 diopters, and the local refractive power can be mapped based on its curvatures and refraction indexes [1]. The cornea assumes the shape of a part-sphere, with approximate cylindrical symmetry, and a polar angle of $53°$.

The human cornea is made up of five basic layers (from anterior to posterior): the epithelium, Bowman's layer, stroma, Descemet's membrane and the endothelium. The outmost layer of the cornea, the epithelium, is made up of epithelial cells and acts as a barrier to protect the interior of the cornea. A thin layer, Bowman's layer, lies beneath the corneal epithelium and consists of randomly arranged interwoven collagen fibrils that

helps in maintaining the corneal shape [20]. The collagen-rich central layer, the stroma, comprises nearly 90% of the thickness of the cornea and is the major contributor to corneal biomechanics [21]. Descemet's membrane lies between the stroma and the posterior layer, called the endothelium, which contains endothelial cells that are structurally supported by Descemet's membrane. In this thesis, we are primarily interested in modelling the stroma and so restrict our modelling to single layer simulations.

As shown in Figure 1.1, three helical alpha amino acid chains form a collagen molecule. Bundles of five of those molecules then self-assemble to form *microfibrils*. About 70 of these, each one twisted and tilted by about 15° to the resulting fibril long axis [22], together form fibrils which exist in layers called *lamellae*. There are approximately 250 stacked lamellae in the center corneal stroma and each adjacent lamella is arranged at different angles, roughly similar to a plywood structure [19]. As described above, significant in-plane anisotropic mechanical responses have been observed, where the determined corneal elastic modulus is found to be significantly larger in horizontal (nasal-temporal) and vertical (superior-inferior) strips compared to diagonal ones [8]. X-ray scattering confirms the stroma lamellae have two more frequently orientated directions near the center of the cornea; typically 60% of the lamellae lie along superior-inferior and nasal-temporal directions [23-25]. These orientation directions play an important role in corneal mechanical properties [21]. The corneal shape could also be affected by an uneven distribution of the lamellar in those two preferred directions [24].

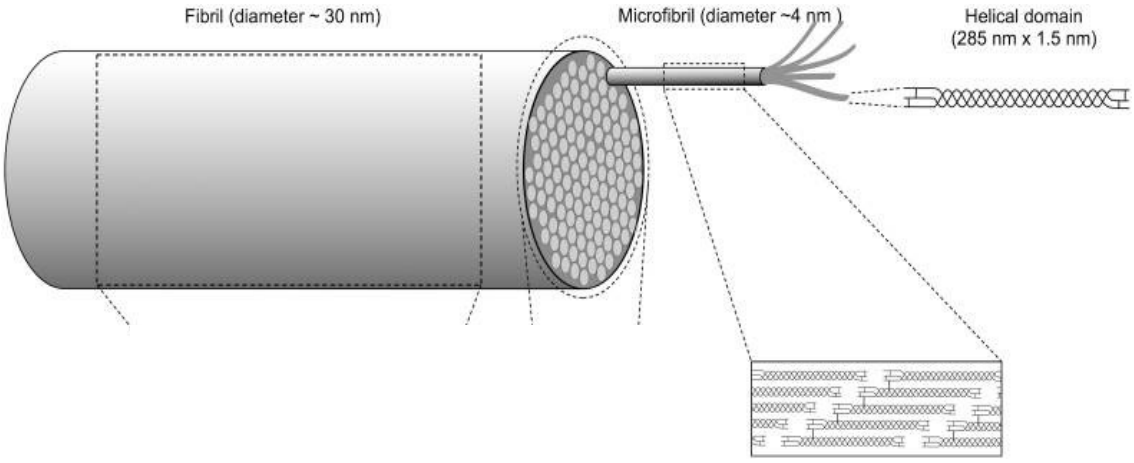

Figure 1.1: Structural hierarchy of collagen within the cornea (not to scale). The top right image shows a collagen molecule that forms part of a microfibril. Each fibril is about 1 mm in length. Figure (https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4655862/figure/fig2/) by Meek MM, and Knupp C [21] is licensed under CC BY 4.0 (https://creativecommons.org/licenses/by/4.0/).

The collagen lamellae structure is continuous with the rest of the globe, i.e., the sclera, and both the cornea and the sclera are primarily composed of the same type of collagen fibrils, although the sclera has an irregular collagen orientation and is completely opaque [19]. The fibril distribution is also crucial for the cornea's optical properties [26, 21]. The cornea transmits over 75% of incident light from 350-1400 nm with absorption of UVB and UVC at short wavelength, and absorption due to water at long wavelengths [27]. The basis of the broad transparency can be explained by light scattering from the collagen fibrils with their regular spacing and the uniform fibril diameters [21]. Birefringence is also observed from the cornea due to its well-ordered fibril arrangement, which enables the stress distribution to be accessed using photoelasticity and verifies the fibril orientation described above [9].

A normal intraocular pressure (IOP) and reasonable corneal mechanical stiffness is required to maintain the cornea in its regular shape. IOP is a key component for characterizing eye disease and monitoring eye health; for example, glaucoma is principally attributed to a high IOP that causes damage to the optical nerve at the rear of the eye. Applanation, i.e. applying external pressure to flatten the cornea, is commonly used to measure *in vivo* corneal material properties and also the IOP. This can be achieved by mechanically pushing the cornea with a prism until the cornea is applanate. Goldmman's applanation tonometry (GAT) is considered to be the gold standard for IOP measurement. Air puff methods, also known as noncontact method, use a pulse of air sent towards the cornea instead of applanating prism, to commonly determine corneal mechanical properties and also IOP. In 2005, the *ocular response analyzer* (ORA) was introduced [28], and this enabled corneal biomechanical properties to be measured *in vivo*, some of which are described above. However, the IOP is often overestimated by non-contact methods [29] which may be due to the cornea's mechanical properties. The cornea also exhibits viscoelastic properties, offered by the stromal ground substance and collagen fibers [30], and during deformation, the applied pressure difference between when the cornea applanates and re-applanates is often reported by clinicians as the *corneal hysteresis* (CH) [4]. CH has been indicated in several studies as a measurable biomarker which strongly relates to the incidence of glaucoma, however a definite correlation still remains as an open question [31]. During air-puff tests, much of the cornea undergoes approximately a linear-elastic deformation, in which each small portion has a strain typically less than ~ 6%, that together result in a large corneal deformation. Nevertheless, the strain at the corneal apex is typically 10%, and goes beyond the linear-elastic region [14].

In this thesis, I adopt a continuum mechanics model and finite element to simulate the mechanical behavior of corneal models under different pressures. Here we set a solid foundation of these techniques to study a complicated and realistic cornea, and the ultimate goal is to supplement calculations of corneal biomechanics such as references [13], [14], and [32]. As described above, a good understanding of the biomechanical properties of the cornea is essential for predicting corneal behavior under large- and small-scale deformation, accessing the biomechanical properties, and achieving an accurate measurement of IOP.

In this context, a linear-elastic static mechanical model is employed to study corneal deformation in the linear region, and further we introduce two nonlinear, hyperelastic mechanical models, Neo-Hookean and Rivlin-Mooney, to simulate the corneal nonlinear response undergoing a large deformation. The scientific computing approach adopts finite element modelling as the key technique for numerical simulation, which is an efficient way to model the deformation of a physical body under various conditions and helps with the analysis of biomechanical properties. Our work concentrates on the *elastic behavior* of the cornea; its viscoelastic response may be addressed in future studies, and also an anisotropic inhomogeneous material may be examined by also taking into account. These resulting finite element simulations could then be compared with experimental data and used to more precisely characterize corneal health.

Chapter 2

Continuum solid mechanics and finite element modelling

In order for the cornea to focus light on the retina properly, it is important to maintain the correct geometric structure with appropriate material properties. A theoretical mechanical model with the aid of numerical techniques is capable of determining the material properties and explaining the observed deformation of physical bodies over both large- and small-scales. In this chapter, we briefly review the mechanical concepts that describe material deformation, as well as a suitable numerical method for simulating this, called *finite element modelling*.

2.1 Introduction to the mechanics of solid deformable materials

Solid mechanics is the study of the behavior of a solid material, here we specifically focus on the deformation resulting from an undeformed body experiencing external forces. Within a continuum approach, the physical body is modeled as a continuous mass that fills the space, Ω, that it occupies. The dynamics and kinematics of material deformation at a point is characterized by a few fundamental physical descriptors, such as *stress* and *strain*. In this section, the physical phenomena of various stresses and the

resulting deformation due to external loading is reviewed [33-35].

2.1.1 Defining stress and strain tensors

There are two distinct ways to apply forces to a solid body and thus to cause material deformation. First, the material can experience forces that originate at a distance, for example gravity, leading to what is called a "*body force"* (with units $\mathrm{N/m^3}$), which acts on volume elements within the physical body. Second, deformation can also be caused by a force applied through contact with a boundary surface (an external surface) of the body, such as an air pressure pulse applied to the front surface of the cornea, which is called an *external contact force* or *surface force* (with units of newtons). After a solid is deformed into its equilibrium position, resulting atom-atom interactions may then exist inside the solid in order to maintain the equilibrium state, which occur between internal surfaces, and are called *internal contact forces*.

The ratio of a force, $\Delta\vec{F}$, distributed across, and applied to, a surface (whether boundary or internal) to the surface area itself, ΔA , as this area becomes infinitesimally small, is called the *traction*. The traction thus relates to a specified surface with a normal vector $\hat{n}$, at position $\vec{x}$:

$$\vec{t}\left(\hat{n},\vec{x}\right) \equiv \lim_{\Delta A \to 0} \frac{\Delta \vec{F}}{\Delta A}, \qquad [2.1]$$

and can be considered to be the force density or force distribution acting on the specified surface, and thus an infinite number of tractions exist at any position. For example, consider a cuboid composed of uniform material subject to only two opposing uniaxial

(perpendicular to each face) extension forces with magnitude *F*, acting on the faces of area *A*, called surface '1' and '2' respectively, as shown in Figure 2.1.a. If these forces are distributed uniformly across these surfaces then in equilibrium the resulting tractions relating to the surfaces '1' and '2' as shown, at position $\vec{x}_{surf\,1}$ and $\vec{x}_{surf\,2}$ can be written as

$$\vec{t}(+\hat{e}_2, \vec{x}_{surf\,1}) = \frac{F}{A}\hat{e}_2 \quad , \quad \vec{t}(-\hat{e}_2, \vec{x}_{surf\,2}) = -\frac{F}{A}\hat{e}_2 \quad . \qquad [2.2a,b]$$

Here, these are external tractions, as they directly arise from external contact forces, and they are relatively easy to picture. In contrast, it can be difficult to picture *internal* tractions within a body. We now introduce the concept of the *stress tensor* which helps with determining such tractions.

Consider instead the separation plane 'B' which separates the cuboid into two identifiable parts as shown in Figure 2.1 (b). Importantly, the two parts are still presumed in contact. Each part has to be in equilibrium as the whole cuboid is in equilibrium. For the head-side body along $\hat{e}_2$, the integral of the internal tractions over the internal surface '3' has to balance the external force on surface '1',

$$\int_{surf\,3} \vec{t}(-\hat{e}_2, \vec{x}_{surf\,3}) \bullet (-\hat{e}_2) dA = \int_{surf\,1} \vec{t}(\hat{e}_2, \vec{x}_{surf\,1}) \bullet (\hat{e}_2) dA \; . \qquad [2.3]$$

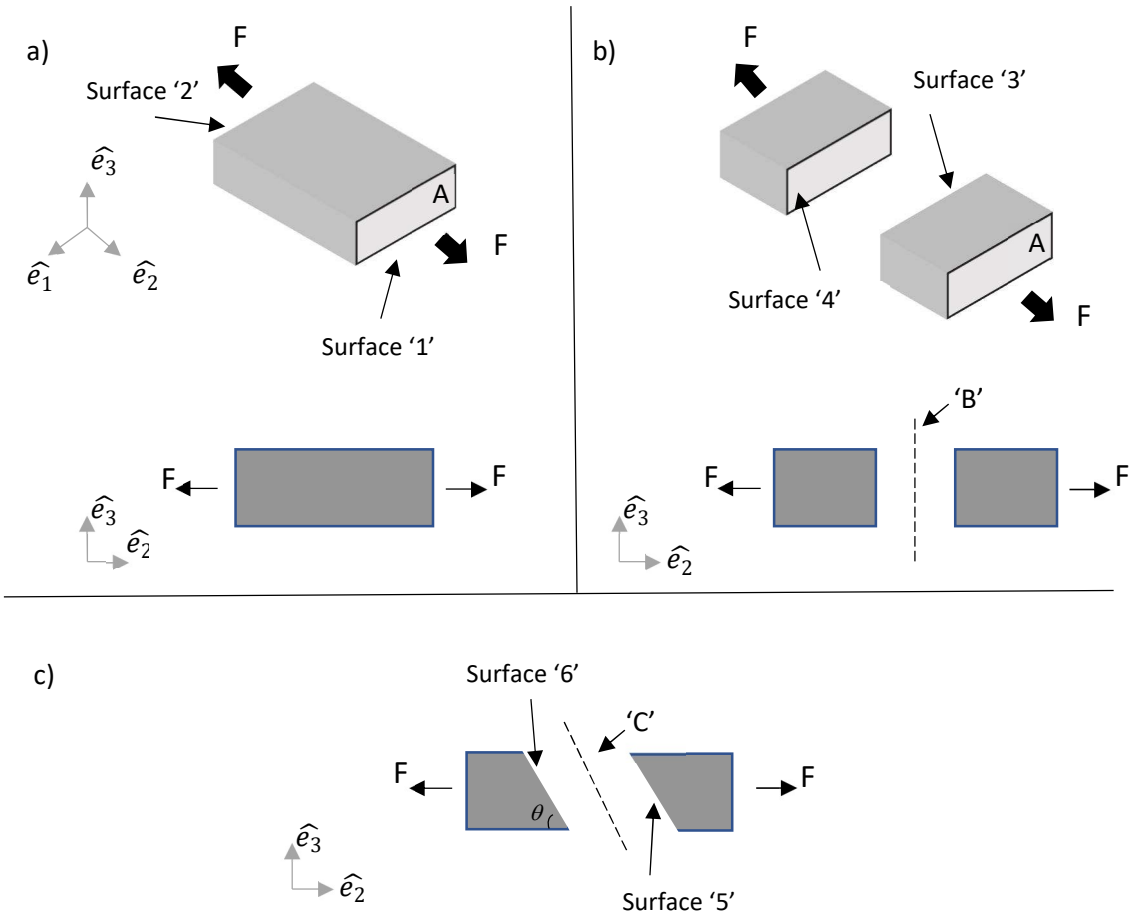

Figure 2.1: A uniform cuboid in equilibrium subjected to two forces with magnitude F on opposite faces give rise to the tractions acting on a) boundary surfaces, '1' and '2', b) the internal surfaces, '3'and '4', with normal vector $\pm\hat{e}_2$, respectively, and c) the internal surfaces, '5' and '6', with normal vector $\pm\hat{n}_5 = \mp(\sin\theta\hat{e}_2 + \cos\theta\hat{e}_3)$, respectively.

As here the cuboid is assumed to be uniform, the traction acting on surface '3' at position $\vec{x}_{surf\,3}$ can then be obtained, as can similarly, the traction acting on surface '4':

$$\vec{t}(-\hat{e}_2, \vec{x}_{surf\,3}) = -\frac{F}{A}\hat{e}_2 \quad , \quad \vec{t}(+\hat{e}_2, \vec{x}_{surf\,4}) = \frac{F}{A}\hat{e}_2 \quad . \qquad [2.4a,b]$$

For this choice of separation plane, each internal traction is equal and opposite to the external traction. Note this validates *Cauchy's fundamental lemma*, that in equilibrium the traction acting on the opposite sides of any surface element, whether it is an internal or boundary surface element, are equal in magnitude and opposite in direction.

Now consider an alternate choice of separation plane, 'C', which separates the cuboid into two, as shown in Figure 2.1(c). Here one equilibrium relation now gives

$$\int_{surf\,5} \vec{t}(\hat{n}_5, \vec{x}_{surf\,5}) \bullet \hat{n}_5 dA = \int_{surf\,1} \vec{t}(\hat{e}_2, \vec{x}_{surf\,1}) \bullet (\hat{e}_2) dA \quad . \qquad [2.5]$$

The traction acting on surface '5' at position $\vec{x}_{surf\,5}$ is thus

$$\vec{t}(\hat{n}_5, \vec{x}_{surf\,5}) = -\frac{F}{A} \cos\theta \, \hat{e}_2 \quad . \qquad [2.6]$$

For *this* choice of separation plane, the internal traction is **not** equal and opposite to the external traction. However, as expected, this result still satisfies a physical, atomic model where the force per unit area acting on surfaces '5' and '6' has reduced compared to the case of a normal separation plane as the atomic density (atoms per unit area) has decreased, while the area itself has increased, resulting in the same physical internal contact forces between adjacent pairs of atoms within the body.

As we have just seen, tractions at any particular position inside a body clearly depend on the surface normals considered, and so the s*tress tensor* is introduced to enable the determination of these tractions for any surface normal at any position. Each element of this tensor is a component of the traction vector that acts on one of three specified surfaces with normal vectors that are the stress tensor's basis vectors, as we now see.

The *Cauchy stress tensor*, $\underline{\underline{\sigma}}(\vec{x})$, also known as the *true stress tensor*, is a second order symmetric tensor and is defined at deformed position $\vec{x}$ using a particular basis as

$$\vec{t}(\hat{n},\vec{x}) = \underline{\underline{\sigma}}(\vec{x})\hat{n} \quad , \qquad [2.7]$$

where the $\hat{n}$ is a unit vector of any surface at point $\vec{x}$. If $\{\hat{n}_i\}$ are the basis vectors of $\underline{\underline{\sigma}}$, then the traction acting on each of the three surfaces with these normal vectors are simply the row (also column) vectors of this stress tensor. The traction acting on any surface at each position within a deformed body can thus be found by applying Equation 2.7. As second useful stress tensor when using an undeformed (labelled by $\vec{X}$) configuration is the transpose of the *first Piola-Kirchhoff* stress tensor (also called the *PK1 stress* or *engineering stress*): $\underline{\underline{P}} = J\underline{\underline{\sigma}}\underline{\underline{F}}^{-T}$, where $\underline{\underline{F}}$ is the *deformation gradient tensor* (see Equation 2.10 below), and *J*, the *volume ratio*, is the determinant of this tensor. The Cauchy stress tensor is thus composed of nine components, and we generally write it in matrix form,

$$\underline{\underline{\sigma}}\left(\vec{x}\right) = \begin{bmatrix} \sigma_{11} & \sigma_{12} & \sigma_{13} \\ \sigma_{21} & \sigma_{22} & \sigma_{23} \\ \sigma_{31} & \sigma_{32} & \sigma_{33} \end{bmatrix} \quad . \qquad [2.8]$$

Element σ_{ij} is the component of the traction vector on an internal or boundary surface of the body at position $\vec{x}$ with normal vector $\hat{e}_i$ along direction $\hat{e}_j$, where $\hat{e}_i$ and $\hat{e}_j$ are two of the three basis vectors with which the Cauchy stress tensor was defined. The

symmetry of the stress tensor ($\sigma_{ij} = \sigma_{ji}$) can be proved using the conservation of linear momentum and angular momentum.

Hence, there are only six independent components and the Cauchy stress tensor is frequently reduced to a vector form,

$$\vec{\sigma}\left(\vec{x}\right) = \begin{bmatrix} \sigma_{11} \\ \sigma_{22} \\ \sigma_{33} \\ \sigma_{12} \\ \sigma_{23} \\ \sigma_{31} \end{bmatrix} \quad . \qquad [2.9]$$

As we now show, this $\{\sigma_{ij}\}$ set of elements can be partitioned into those describing normal and tangential tractions acting on each of the basis surfaces. Each *normal* stress component, σ_{ii}, denotes the force per unit area along the normal direction to the *i*-th surface, and each *tangential* component, σ_{ij} , denotes the force per unit area along the two orthogonal *j*-th directions that are parallel to the *i*-th surface. The $\{\sigma_{ij}\}$ set are known as the *shear stress* components and sometimes denoted by $\{\tau_{ij}\}$ in the literature.

Consider three of the six faces of an infinitesimal cube located at position $\vec{x}$, as shown in Figure 2.2, and a Cauchy stress tensor that invokes these same vectors as its basis vectors. The three normal and six shear stress components on those three faces can then be seen in Figure 2.2.

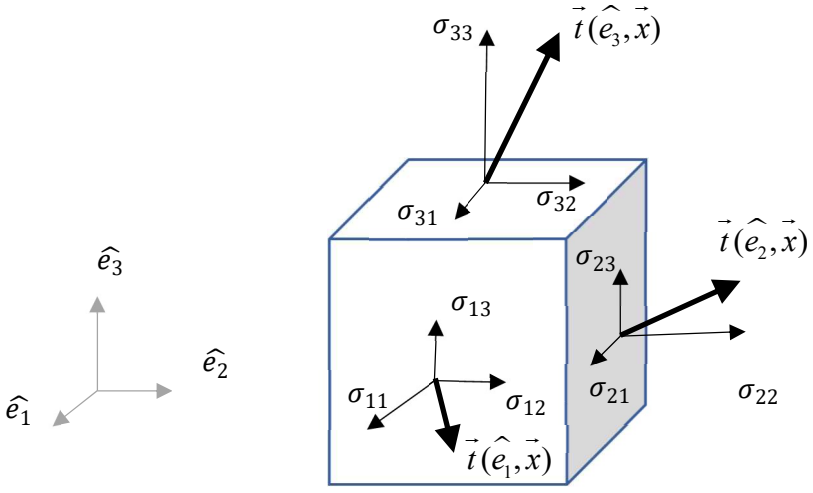

Figure 2.2: Components of the tractions acting on three faces with normal vectors $\hat{e}_1$, $\hat{e}_2$ and $\hat{e}_3$ of an infinitesimal cube centered at position $\vec{x}$ as related to the stress tensor. Note that the opposite surfaces with normal vectors $-\hat{e}_1$, $-\hat{e}_2$ and $-\hat{e}_3$ experience respectively equal and opposite tractions.

Meanwhile *kinematics* deals with the motion of the body undergoing deformation, regardless of an investigation of the origin of any forces. The motion of a generic material point ending up at location $\vec{x}$ after deformation can be characterized by the second order, two-point[1], tensor called the *deformation gradient tensor*, $\underline{\underline{F}}$, at this position:

$$\underline{\underline{F}} = \text{Grad}\ \vec{x}(\vec{X}) \quad ; \quad F(\vec{X}) = \frac{\partial \vec{x}(\vec{X})}{\partial \vec{X}} \quad ; \quad F_{ij}(\vec{X}) = \frac{\partial x_i(\vec{X})}{\partial X_j} \quad . \qquad [2.10]$$

Here we have specifically written the gradient using three equivalent notations: the traditional 'Grad' form, a widely-used shorthand vector form, and the explicit coordinate

[1] The term *two-point* indicates that this tensor depends on the actual coordinate unit vectors chosen in two coordinate frames, i.e. the reference configuration and the deformed configuration. This applies to both $\underline{\underline{F}}$ and $\underline{\underline{P}}$.

form. In all cases $\vec{X}$ is the position of the material point in the reference (undeformed) configuration; see Figure 2.3. The displacement $\vec{u}$ is $\vec{u} = \vec{x} - \vec{X}(\vec{x})$, and much of the work within this thesis will involve finding this displacement vector $\vec{u}$ for a variety of physical bodies.

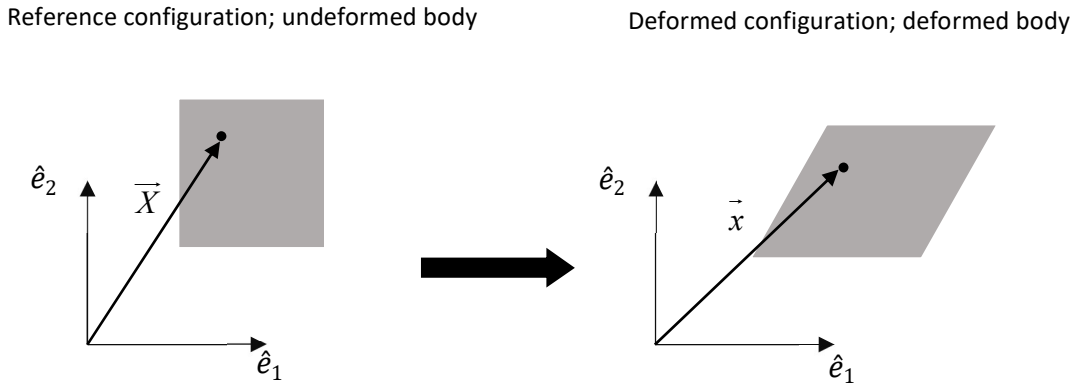

Figure 2.3: Demonstration of the reference configuration with coordinate $\vec{X}$ and the deformed configuration with coordinate $\vec{x}$; both systems adopt the same basis vectors.

The deformation gradient tensor serves to carry an undeformed infinitesimal line element $d\vec{X}$ into its deformed element $d\vec{x}$:

$$d\vec{x} = \underline{\underline{F}}(\vec{x})\, d\vec{X} \quad . \qquad [2.11]$$

However, the deformation gradient does not clearly quantify the change of angle between pairs of line elements after deformation. From here on, we use the generic term *strain* to represent the kinematics of the material deformation. The response of an arbitrary material point within the body due to a deformation can then be described in terms of one of several (dimensionless) *strain tensors*. These are a convenient measure of

the deformation and can be useful to link to the dynamics. Here, we introduce some commonly used strain tensors.

The *right Cauchy-Green strain tensor* $\underline{\underline{C}}$ gives a measure of the dot product of a pair of line elements, denoted by $d\vec{x}_1$ and $d\vec{x}_2$, after deformation,

$$d\vec{x}_1 \bullet d\vec{x}_2 = d\vec{X}_1 \underline{\underline{F}}^T \bullet \underline{\underline{F}} d\vec{X}_2 = d\vec{X}_1 \bullet \underline{\underline{C}} d\vec{X}_2 \quad , \tag{2.12}$$

and is therefore defined as

$$\underline{\underline{C}} \equiv \underline{\underline{F}}^T \underline{\underline{F}} \quad . \tag{2.13}$$

Here we can see that the right Cauchy-Green strain tensor is a symmetric tensor and there exists three eigenvalues, ${\lambda_1}^2$, ${\lambda_2}^2$ and ${\lambda_3}^2$, where $\{\lambda_a\}$ are called the *principal stretches*. These are the stretch ratios of line elements lying along the three principal axes (the eigenvectors of $\underline{\underline{C}}$). Three physically-useful *invariants* of the right Cauchy-Green strain tensor can be expressed in terms of these principal stretches:

$$\begin{aligned} I_1 &= {\lambda_1}^2 + {\lambda_2}^2 + {\lambda_3}^2 \\ I_2 &= {\lambda_1}^2{\lambda_2}^2 + {\lambda_2}^2{\lambda_3}^2 + {\lambda_3}^2{\lambda_1}^2 \\ I_3 &= {\lambda_1}^2{\lambda_2}^2{\lambda_3}^2 \end{aligned} \tag{2.14}$$

Similarly, a deformation tensor in the reference configuration, called the *left Cauchy-Green strain tensor*, $\underline{\underline{B}}$, is defined as

$$\underline{\underline{B}} \equiv \underline{\underline{F}}\,\underline{\underline{F}}^T \quad . \tag{2.15}$$

The *Green-Lagrange strain tensor* $\underline{\underline{E}}$ provides the change in the squared lengths of a line element,

$$\frac{\left|d\vec{x}\right|^2-\left|d\vec{X}\right|^2}{2}=d\vec{X}\bullet\underline{\underline{E}}\,d\vec{X} \qquad , \tag{2.16}$$

where

$$\underline{\underline{E}}\equiv\frac{1}{2}(\underline{\underline{C}}-\underline{\underline{I}}) \qquad ; \qquad E_{ij}=\frac{1}{2}\left(\frac{\partial u_i}{\partial x_j}+\frac{\partial u_j}{\partial x_i}+\frac{\partial u_k}{\partial x_i}\frac{\partial u_k}{\partial x_j}\right) \quad . \tag{2.17}$$

The *Cauchy strain tensor*, $\underline{\underline{\varepsilon}}$, also known as the *engineering strain tensor* or the *infinitesimal strain tensor*, defines the kinematic relation between the strain and displacement, $\vec{u}$, when the deformation of the body satisfies small deformation theory,

$$\underline{\underline{\varepsilon}}=\frac{1}{2}\left(\underline{\underline{F}}^T+\underline{\underline{F}}\right)-\underline{\underline{I}} \qquad ; \qquad \varepsilon_{ij}=\frac{1}{2}\left(\frac{\partial u_i}{\partial x_j}+\frac{\partial u_j}{\partial x_i}\right) \quad . \tag{2.18}$$

Finally, for completeness we mention the *Henckey strain*, also known as the *true strain*, $\underline{\underline{\varepsilon}}_{true}$, which is an incremental strain measure, where the deformation is considered to be a series of steps, and this strain compares the deformation between the current configuration and the previous deformed configuration.

In this section, we have introduced two fundamental concepts of solid mechanics. The stress tensors determine the tractions within a body and the strain tensors describe the deformation of a body. The stress and strain tensors introduced in this section are summarized in Table 2.1.

Table 2.1: Summary of the stress and strain tensors introduced in Section 2.1.1.

Tensor name	Notation	Expression
Stress tensors		
Cauchy stress tensor	$\underline{\underline{\sigma}}$	$\vec{t}(\hat{n},\vec{x}) = \underline{\underline{\sigma}}(\vec{x})\,\hat{n}$
First Piola-Kirchhoff stress tensor	$\underline{\underline{P}}$	$\underline{\underline{P}} = J\underline{\underline{\sigma}}\,\underline{\underline{F}}^{-T}$
Strain tensors		
Right Cauchy-Green strain tensor	$\underline{\underline{C}}$	$\underline{\underline{C}} = \underline{\underline{F}}^{T}\,\underline{\underline{F}}$
Left Cauchy-Green strain tensor	$\underline{\underline{B}}$	$\underline{\underline{B}} = \underline{\underline{F}}\,\underline{\underline{F}}^{T}$
Green-Lagrange strain tensor	$\underline{\underline{E}}$	$\underline{\underline{E}} = \frac{1}{2}(\underline{\underline{C}} - \underline{\underline{I}})$
Cauchy strain tensor	$\underline{\underline{\varepsilon}}$	$\underline{\underline{\varepsilon}} = \frac{1}{2}\left(\underline{\underline{F}}^{T} + \underline{\underline{F}}\right) - \underline{\underline{I}}$

2.1.2 Including the balance principle and the constitutive relation to derive the strong form representation

The behavior of a body experiencing deformation can be predicted using a model where the steps involved in these formulation, governing equation within the model is illustrated in Figure 2.4. The *balance principles* of both linear and angular momentum are first employed to analyze all internal and external forces. Then, the appropriate material model is chosen to present the correct constitutive equation which approximates the relation between stress and strain. Finally, material point displacements are determined from the appropriate strain tensor using kinematics. Putting these together results in the *governing equation*, normally a partial differential equation.

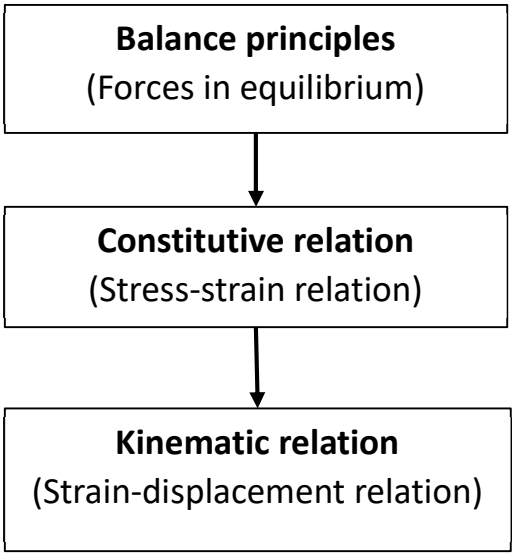

Figure 2.4: Steps involved to arrive at final equation representing material deformation, called the governing equation.

The balance principles of mass, linear momentum, angular momentum etc., which are applicable to any solid material, have to be satisfied. Here, with the assumption that mass is conserved, and rigid body rotation is not involved during deformation, i.e. the total angular momentum of the body is unchanged, we focus on the application of the balance principle of linear momentum.

The *resultant force* $\vec{F}$ on an arbitrary volume element within the deformed body, Ω_e, is generally composed from two types of force: the "body force", $\vec{b}$, acting on each constituent *infinitesimal* volume within the body, and the tractions, $\vec{t}$, which act on each boundary surface of the element.

$$\vec{F} = \int_{\Omega_e} \vec{b}\, dV + \int_{\partial\Omega_e} \vec{t}\, ds = \frac{d}{dt} \int_{\Omega_e} \rho \vec{v} dV \quad . \qquad [2.19]$$

The second equality is known as the balance principle of linear momentum, where ρ is the mass density of the body. Applying the divergence theorem and assuming the acceleration to be zero for a static case, Equation 2.19 can be extended to this quasi-static equilibrium case, resulting in

$$\int_{\Omega_e} \vec{b}\, dV + \int_{\Omega_e} \mathrm{div}\underline{\underline{\sigma}}\, dV = 0 \qquad [2.20]$$

This leads to the following equation, which applies to the *deformed* configuration,

$$\vec{b} + \vec{\nabla} \cdot \underline{\underline{\sigma}} = \vec{0} \qquad ; \qquad b_i + \sum_{j=1}^{3} \frac{\partial \sigma_{ij}}{\partial x_j} = 0 \quad i = \{1,2,3\}. \qquad [2.21]$$

While a purely-rigid body is undeformable, points within a deformable body change their relative position during deformation. In general, a relationship between two physical quantities (e.g. stress and strain) subject to the specified material properties is called a *constitutive relation*. In 1678, the law of elasticity describing the well-known linear stress-strain relationship for uniaxial 1d extension/compression was proposed by physicist Robert Hooke [36], which is known as *Hooke's law.* If a deformed material returns to its initial state once the external forces are removed, this material in this case is called an *elastic material*. During unloading, an elastic material thus traces the same loading path, and both loading and unloading are rate-independent of applied forces, with no net energy dissipated. This can be compared with either *viscoelastic* deformation (often called *elastomeric* deformation for easily-stretched rubber-like polymers), where the material returns to its initial state upon the removal of the external force, but with some time-dependence, or *plastic* deformation, which is not fully reversible upon the removal of external forces. The relationship between stress and strain varies for different materials,

as can be see in Figure 2.5. We can see that materials that exhibit elastic behaviour can demonstrate either or both of *linear-* or *nonlinear-elastic* behavior.

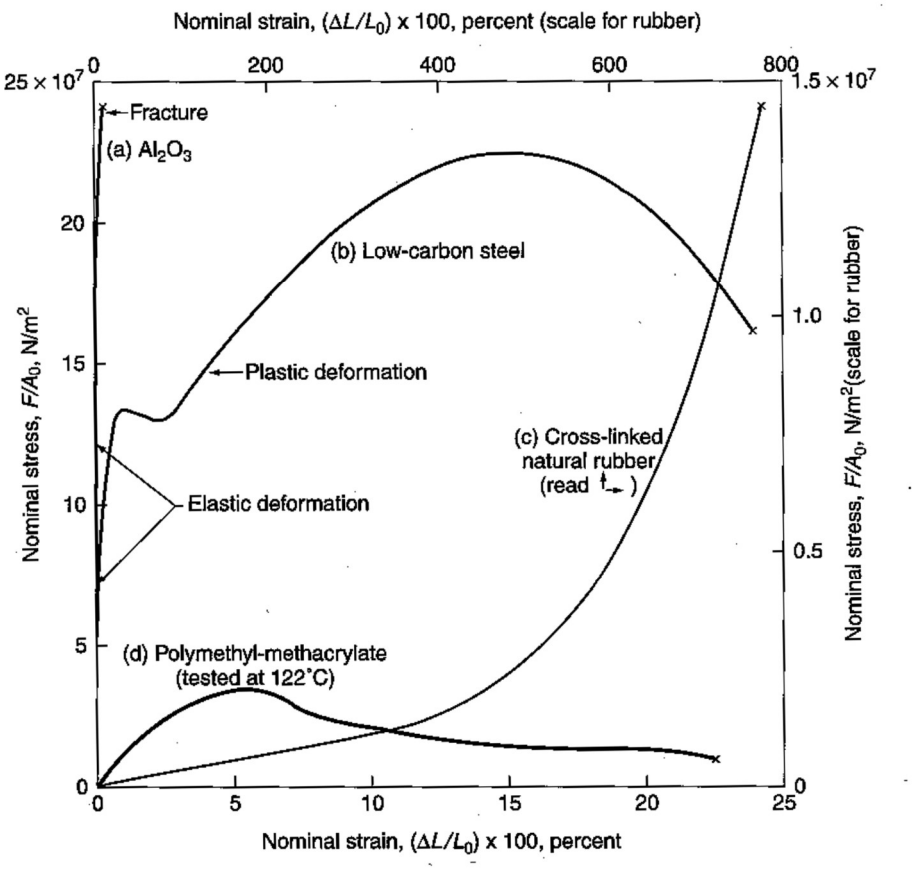

Figure 2.5: Curves of nominal stress vs nominal strain for (a) aluminum oxide, (b) low-carbon steel, (c) cross-linked natural rubber, and (d) polymethyl-methacrylate. It transpires that these four examples exhibit mechanical responses that include elastic, plastic, elastomeric and viscoelastic behaviour, respectively. (Figure taken from A.G. Guy 1972 [38] with permission, Appendix B.)

For linear-elastic deformation, where the stress is directly dependent upon to the strain, the stress-strain relationship can be characterized by the form,

$$\underline{\underline{\sigma}} = \underline{\underline{\underline{\underline{\mathbb{C}}}}}\,\underline{\underline{\varepsilon}} \qquad ; \qquad \sigma_{ij} = \sum_{k=1,l=1}^{3,3} \mathbb{C}_{ijkl}\varepsilon_{kl} \qquad i,j = \{1,2,3\} \qquad [2.22]$$

where $\underset{\equiv\equiv}{\mathbb{C}}$ is the fourth-order *stiffness tensor*. The symmetry property of the Cauchy stress tensor and the Cauchy strain tensor enables us to reduce the stiffness tensor with 81 elements to a matrix with only 36 independent elements for the most general case:

$$\begin{bmatrix} \sigma_{11} \\ \sigma_{22} \\ \sigma_{33} \\ \sigma_{12} \\ \sigma_{23} \\ \sigma_{31} \end{bmatrix} = \begin{bmatrix} \mathbb{C}_{1111} & \mathbb{C}_{1122} & \mathbb{C}_{1133} & \mathbb{C}_{1112} & \mathbb{C}_{1123} & \mathbb{C}_{1131} \\ \mathbb{C}_{2211} & \mathbb{C}_{2222} & \mathbb{C}_{2233} & \mathbb{C}_{2212} & \mathbb{C}_{2223} & \mathbb{C}_{2231} \\ \mathbb{C}_{3311} & \mathbb{C}_{3322} & \mathbb{C}_{3333} & \mathbb{C}_{3312} & \mathbb{C}_{3323} & \mathbb{C}_{3331} \\ \mathbb{C}_{1211} & \mathbb{C}_{1222} & \mathbb{C}_{1233} & \mathbb{C}_{1212} & \mathbb{C}_{1223} & \mathbb{C}_{1231} \\ \mathbb{C}_{2311} & \mathbb{C}_{2322} & \mathbb{C}_{2333} & \mathbb{C}_{2312} & \mathbb{C}_{2323} & \mathbb{C}_{2331} \\ \mathbb{C}_{3111} & \mathbb{C}_{3122} & \mathbb{C}_{3133} & \mathbb{C}_{3112} & \mathbb{C}_{3123} & \mathbb{C}_{3131} \end{bmatrix} \begin{bmatrix} \varepsilon_{11} \\ \varepsilon_{22} \\ \varepsilon_{33} \\ \varepsilon_{12} \\ \varepsilon_{23} \\ \varepsilon_{31} \end{bmatrix}, \qquad [2.23]$$

where $\mathbb{C}_{ijkl}$ is the *elastic modulus* that relates the strain component ε_{ij} with stress component σ_{kl}.

The number of independent elements in a stiffness tensor can be further reduced depending upon the existence of material symmetry. When the material is isotropic, this constitutive relation can be conveniently written as

$$\begin{bmatrix} \sigma_{11} \\ \sigma_{22} \\ \sigma_{33} \\ \sigma_{12} \\ \sigma_{23} \\ \sigma_{31} \end{bmatrix} = \begin{bmatrix} \lambda+2\mu & \lambda & \lambda & 0 & 0 & 0 \\ \lambda & \lambda+2\mu & \lambda & 0 & 0 & 0 \\ \lambda & \lambda & \lambda+2\mu & 0 & 0 & 0 \\ 0 & 0 & 0 & \mu & 0 & 0 \\ 0 & 0 & 0 & 0 & \mu & 0 \\ 0 & 0 & 0 & 0 & 0 & \mu \end{bmatrix} \begin{bmatrix} \varepsilon_{11} \\ \varepsilon_{22} \\ \varepsilon_{33} \\ 2\varepsilon_{12} \\ 2\varepsilon_{23} \\ 2\varepsilon_{31} \end{bmatrix}. \qquad [2.24]$$

Here λ and μ are the *first and second Lamé constants* respectively, which can be written as

$$\lambda = \frac{E\nu}{(1+\nu)(1-2\nu)}, \qquad \mu = \frac{E}{2(1+\nu)}. \qquad [2.25a,b]$$

Here $E = \frac{\sigma_{ii}}{\varepsilon_{ii}}$ and is called the *Young's modulus* and is independent of direction *i*. The material parameter ν is the dimensionless *Poisson's ratio*, which varies between -1 and +0.5 and reflects how much an extension in one direction results in a compression in another. In this context, a material is considered to be *incompressible* when its volume remains the same under a uniaxial deformation, and such a material has $\nu = 0.5$ and satisfies a volume ratio $J \equiv \det\left(\underline{\underline{F}}\right) = 1$. However, such purely-incompressible materials do not physically exist and a material that approaches such behavior is denoted as *nearly-incompressible*.

With an assumed isotropy and an assumed value of $\nu = 0$ for the materials shown in figure 2.5, we can ignore the other uniaxial and shear strains and thus we determine *E* at low strains to be $240\ \text{GPa}$, $150\,\text{GPa}$, $1.9\ \text{MPa}$, and $1.2\ \text{GPa}$, for aluminum oxide, low-carbon steel, cross-linked natural rubber, and PMMA, respectively. Without this assumption of $\nu = 0$, the precise response of a body under uniaxial strain depends on the dimensions of the body; we shall simulate such a case using a finite element approach in section 3.3.3.

The equations deriving from balance principle (e.g. Equation 2.21), the constitutive relation (e.g. Equation 2.22), and the stress definition (e.g. Equation 2.18), can be combined into a single governing pde, with the original boundary conditions experienced by the body, and this is said to constitute the *strong form* of the mathematical problem to be solved for the physical system. We now give, as a demonstration, a relatively simple example of this process for an effective-1d system that can be solved analytically.

2.1.3 A simple analytical example: a uniaxially-loaded, linear-elastic, bar

Here we examine the analytical solution of a uniaxially-loaded, isotropic, homogeneous, linear-elastic bar lying along $\hat{e}_1$, with a uniform cross section, uniform elastic modulus, E, and a *deformed* length *L*. Points along the bar are thus labelled by either coordinate X_1 (in the undeformed configuration) or coordinate x_1 (in the deformed configuration). The bar may also experience a body force, $\vec{b}(\vec{x}) = b_0 \times x_1 \hat{e}_1$, e.g. a gravitational or electromagnetic force, along the axial direction. Let one end of the bar be fixed with $u(X_1 = 0) = 0$, and the deformed bar is assumed to be at a steady state. The boundary condition at the other end may then correspond to either a known, fixed displacement: a *Dirichlet boundary condition*, or a known fixed traction: a *Neumann boundary condition*. The task is to determine at equilibrium the displacement of material points along the bar.

At steady state, the deformed bar must satisfy Equation 2.21,

$$\vec{b}(\vec{x}) + \vec{\nabla} \bullet \underline{\underline{\sigma}}(\vec{x}) = \vec{0} \quad . \qquad [2.26]$$

Substituting the linear constitutive relation into these,

$$\vec{b} + \vec{\nabla} \bullet (\underline{\underline{\underline{\underline{\mathbb{C}}}}} : \underline{\underline{\varepsilon}}) = \vec{0} \quad . \qquad [2.27a]$$

Rewriting this in coordinate form in terms of the Lamé constants

$$
\begin{aligned}
&b_1+\frac{\partial}{\partial x_1}[(\lambda+2\mu)\varepsilon_{11}+\lambda\varepsilon_{22}+\lambda\varepsilon_{33}]+2\frac{\partial}{\partial x_2}[\mu\varepsilon_{12}]+2\frac{\partial}{\partial x_3}[\mu\varepsilon_{13}]=0\\
&2\frac{\partial}{\partial x_1}[\mu\varepsilon_{12}]+\frac{\partial}{\partial x_2}[(\lambda+2\mu)\varepsilon_{22}+\lambda\varepsilon_{11}+\lambda\varepsilon_{33}]+2\frac{\partial}{\partial x_3}[\mu\varepsilon_{23}]=0\\
&2\frac{\partial}{\partial x_1}[\mu\varepsilon_{13}]+2\frac{\partial}{\partial x_2}[\mu\varepsilon_{23}]+\frac{\partial}{\partial x_3}[(\lambda+2\mu)\varepsilon_{33}+\lambda\varepsilon_{22}+\lambda\varepsilon_{11}]=0
\end{aligned}
\qquad [2.27b]
$$

Here, we model this uniform body assuming no interactions between different directions, and also no shear stresses applied to any boundary surface. The former means that the uniaxial strain along one direction is independent of that along any other (so Poisson's ratio is set to be to zero: $\nu=0$), and the latter that all shear strain components are zero throughout the body. The mechanical response of a more general 3d case will be solved using a finite element approach in Section 3.3.3. Consequently, Equation 2.27b can here be reduced to

$$
b_1+\frac{d}{dx_1}\left(E\varepsilon_{11}\right)=0 \quad . \qquad [2.28]
$$

Substituting into this equation the definition of the Cauchy strain tensor from Equation 2.18 results in

$$
b_1+\frac{d}{dx_1}\left(E\frac{du_1}{dx_1}\right)=0 \quad . \qquad [2.29]
$$

We now determine the displacement of the bar from its deformed state along $\hat{e}_1$, where for simplicity u_1 is labelled u below. We adopt two alternate sets of boundary conditions: a double-Dirichlet boundary condition or a Dirichlet-Neumann boundary condition. For the first of these, we set:

$$u(x_1 = 0) = 0 \text{ m} \quad ; \quad u(x_1 = L) = u_0 \ . \tag{2.30}$$

For the second of these, we combine Equations 2.7 and 2.22 to then set

$$u(x_1 = 0) = 0 \text{ m} \quad ; \quad t(x_1 = L) = E\varepsilon_{11} = t_0 \ . \tag{2.31}$$

Analytically solving Equation 2.29 for each set of boundary conditions results in the following two relationships, respectively

$$u_{DD} = -\frac{b_0}{6E} x_1^{\,3} + \left(\frac{u_0}{L} + \frac{b_0 L^2}{6E} \right) x_1 \tag{2.32}$$

$$u_{DN} = -\frac{b_0}{6E} x_1^{\,3} + \left(\frac{t_0}{E} + \frac{b_0 L^2}{2E} \right) x_1 \tag{2.33}$$

To visualize these solutions, we choose a material with Young's modulus, $E = 3 \text{ MPa}$; a deformed length, $L = 0.3 \text{ m}$; a body force, $b_1(\vec{x}) = 2x_1 \times 10^5 \text{ Nm}^{-3}$; and either a Dirichlet displacement, $u_0 = 0.002 \text{ m}$, or a Neumann traction that corresponds to the same boundary displacement: $t_0 = 14000 \text{ Nm}^{-2}$. The displacement $u(\vec{x})$ from the undeformed configuration is shown in Figure 2.6 for both the double-Dirichlet boundary condition, u_{DD}, and the Dirichlet-Neumann boundary condition, u_{DN}. Because we chose corresponding boundary conditions, Figures 2.6(a), and (b) are identical.

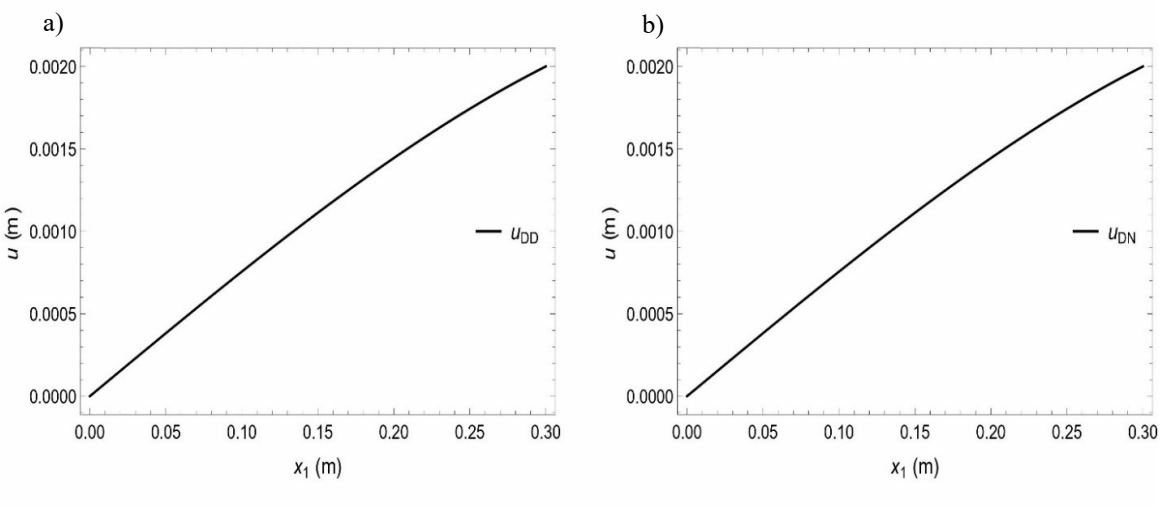

Figure 2.6: Plots of the displacement from the undeformed configuration for: a) a double-Dirichlet boundary condition, u_{DD}, and b) the corresponding Dirichlet-Neumann boundary condition, u_{ND} for a bar with Young's modulus, $E = 3$ MPa; a deformed length, $L = 0.3$ m; body force, $b_1(\vec{x}) = 2x_1 \times 10^5$ Nm^{-3} and neglecting any interactions between different directions.

The Dirichlet-Neumann solution resulted in the bar having an undeformed length $L_0 = 0.298$ m, as expected, and both solutions provide the displacement, $u(x_1)$, throughout the body.

In addition, in Figure 2.7 we can see how the displacement of the bar from its undeformed state varies with the body force, $b_1(\vec{x}) = b_0 \times x_1$ Nm^{-3}. When the bar deforms with no body force (as shown in green, solid line), it obeys a linear displacement function along its length as we expect from Hooke's law for a simple extension. When the bar is subject to a significant body force with this functional dependence (as shown in blue, coarse-dashed line), the bar experiences a compression (adjacent points move closer together along $\hat{e}_1$) for $x_1 > 0.28$ m. The traction along $\hat{e}_1$ at any point along the bar can

be found from Equations 2.7, 2.18 and 2.22, which for this effective-1d isotropic bar is given by $E\frac{du}{dx_1}$; thus the traction can be seen to vary with b_0, and for the highest b_0 considered, is zero at $x_1 \cong 0.28\,\mathrm{m}$.

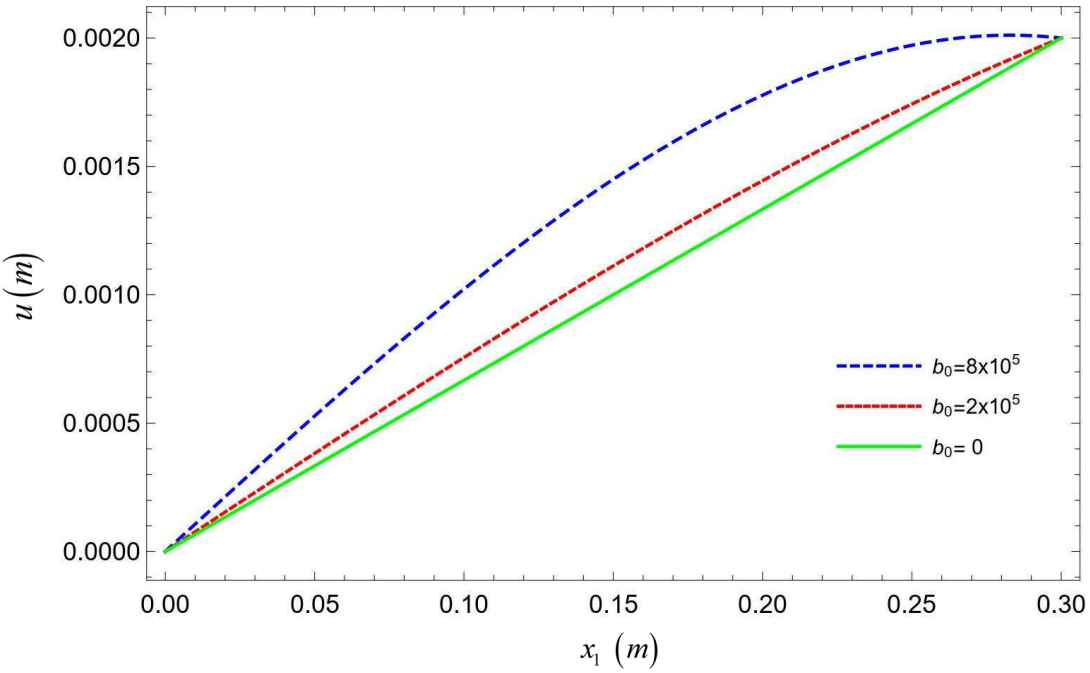

Figure 2.7: Plots of the displacement from the undeformed configuration for a bar experiencing a zero (green, solid), small (red, fine-dashed), and large (blue, coarse-dashed) body force, $b_1 = b_0 \times x_1\,\mathrm{Nm}^{-3}$. Parameter b_0 is $\{0, 2\times10^5, 8\times10^5\}$ respectively.

We have here analytically determined the displacement of an effective-1d material undergoing deformation. This case is very limited, it has little to compare with reality since a fully compressible material with no interactions between different directions was assumed. A more realistic 3d system is difficult and often impossible to solve analytically. In order to address more physically-realistic systems, we shall adopt a finite element approach with numerical solutions.

2.1.4 Beyond linear elasticity: introducing hyperelastic models

If the deformation is nonlinear (and so $\underline{\underline{\underline{\underline{\mathbb{C}}}}}$ is not constant), as often happens at large strain, we can introduce a nonlinear (but still elastic) model. A *hyperelastic* material model is one where the constitutive relation at each strain is derived from a *strain-energy function*, W, which can be represented in terms of the deformation gradient: $W\left(\underline{\underline{F}}\right)$, or in terms of the strain tensors. Typically, the deformation of rubber-like materials (e.g. biological tissues) can be represented by a strain-energy function and thus they can be modeled as a hyperelastic material. In the hyperelastic models we consider here the material is restricted to be homogeneous, and hence the material parameters of the model are constrained to be constant over the body.

The strain-energy function is the energy stored per unit undeformed volume (at a material point $\overrightarrow{X}$) within the body due to the deformation, which is equal to the mechanical work done per volume by all stresses in moving from the undeformed state at time t_0 to the deformed state at time t',

$$W = \int_{t_0}^{t'} \underline{\underline{P}} : \dot{\underline{\underline{F}}}\, dt \quad . \qquad [2.34]$$

By applying the chain rule, the rate of change of the strain-energy function can be expressed as

$$\frac{\partial W}{\partial t} = \frac{\partial W}{\partial \underline{\underline{F}}} : \dot{\underline{\underline{F}}} = \underline{\underline{P}} : \dot{\underline{\underline{F}}} \quad , \qquad [2.35]$$

where, as seen in Section 2.1.1, the shorthand vector notation for a derivative with respect to elements of a vector or matrix is used. However, since this applies for arbitrary $\dot{\underline{\underline{F}}}$, the strain-energy function connects at each material point the deformation gradient tensor $\underline{\underline{F}}$ to the stress tensors $\underline{\underline{P}}$ and $\underline{\underline{\sigma}}$:

$$\underline{\underline{P}} = \frac{\partial W}{\partial \underline{\underline{F}}} \quad ; \quad P_{ij} = \frac{\partial W}{\partial F_{ij}} \qquad i, j = \{1,2,3\} \qquad [2.36]$$

$$\underline{\underline{\sigma}} = J^{-1} \frac{\partial W}{\partial \underline{\underline{F}}} \underline{\underline{F}}^{T} \quad ; \quad \sigma_{ij} = J^{-1} \frac{\partial W}{\partial F_{ij}} F_{ji} \qquad i, j = \{1,2,3\} \qquad [2.37]$$

The strain-energy function can be also defined in terms of other stress and strain tensors [33],

$$\underline{\underline{S}} = 2\frac{\partial W}{\partial \underline{\underline{C}}} = \frac{\partial W}{\partial \underline{\underline{E}}} \quad , \qquad [2.38]$$

where $\underline{\underline{S}}$ is the symmetric *second Piola-Kirchhoff stress tensor*, or the *PK2 stress*, defined as $\underline{\underline{S}} = J\underline{\underline{F}}\,\underline{\underline{\sigma}}\,\underline{\underline{F}}^{-T}$, and unlike $\underline{\underline{P}}$, which consists of normal and shear stresses, $\underline{\underline{S}}$ does not permit a similar physical interpretation.

The stiffness tensor can then be obtained from

$$\underline{\underline{\underline{\underline{\mathbb{C}}}}} = 4\frac{\partial^2 W}{\partial \underline{\underline{C}}\,\partial \underline{\underline{C}}} \quad ; \quad \mathbb{C}_{ijkl} = 4\frac{\partial^2 W}{\partial C_{ij}\,\partial C_{kl}} . \qquad [2.39]$$

The strain-energy function can be decomposed into isochoric and non-isochoric part. If we postulate that a material is isotropic and incompressible, then we can specify two increasingly-accurate forms of the strain-energy function: a *Neo-Hookean* model and a

Mooney-Rivlin model, which are able to predict this materials' deformation in the nonlinear region. These models do not contain a component that models the effects of a material's volume change.

A *Neo-Hookean material model* is a hyperelastic model that can be used to study the nonlinear stress-strain relation of an isotropic, incompressible material, by using the strain-energy function.

$$W_{NH} = \frac{\mu}{2}(I_1 - 3) \qquad [2.40]$$

where μ is the second Lamé constant for the material and the invariant I_1 is defined in Equation 2.14. This mathematically simple model is generally accurate for describing materials under small deformation [37].

A Mooney-Rivlin model can supersede the Neo-Hookean model for large deformation. The strain-energy function is a linear combination of two of the invariants of the right Cauchy-Green deformation tensor given in Equation 2.14,

$$W_{MR} = c_1(I_1 - 3) + c_2(I_2 - 3) \ , \qquad [2.41]$$

where c_1 and c_2 are the Mooney-Rivlin coefficients which can be empirically established.

In reality, the cornea is an *anisotropic*, *inhomogeneous* material mainly due to its preferred collagen orientation, and exhibits nonlinear viscoelastic properties during deformation. Nevertheless, in what follows we shall first consider the cornea to be isotropic and homogeneous in this thesis for convenience. Infinitesimal deformation

theory will be employed to model a linear-elastic, nearly-incompressible cornea with a Poisson's ratio of 0.49999.

2.2 Introduction to finite element modelling

The finite element method (FEM) is a numerical technique for computing approximate solutions to partial differential equations (PDE) given particular boundary conditions [39]. There are two other numerical methods commonly used to solve partial differential equations, which are the finite difference method (FDM) and the finite value method (FVM) [40].

FDM is one of the simplest and oldest techniques for numerically solving differential equations. The principle is to apply a Taylor expansion to approximate the discretized differential equation [41]. An accurate approximation using FDM requires a simple geometry. Conversely, more flexibility for complex geometries are illustrated in FVM and FEM. FVM is a natural choice for computational fluid dynamics as it evaluates flux across element boundaries. FEM meanwhile is an efficient and precise numerical technique for modelling mechanical problems involving complex geometries. Here we select FEM as an appropriate method to simulate the biomechanical behavior of the cornea.

Generally, FEM involves starting from the *strong form* representation, which is a single "*governing*" PDE that derives from applying the balance principle, the constitutive relation, and the definition of stress; together with the boundary conditions. As shown in

Figure 2.8, this strong form is then converted into an integro-differential equation, known as the *weak form*. To perform an FEM calculation, the physical body is divided into a topologically-equivalent system composed of discrete subdomains, known as *elements* or *cells*, each containing a number of *nodes* (i.e. knots in a mesh), and the desired solution is then represented in a finite-dimensional function space (composed of *basis functions*; here we use Lagrange polynomials[2]). We then consider the desired solution calculated only at these *finite* set of nodes, these solution values being known as *degrees of freedom* (dofs), at these locations. The solution anywhere can then be approximately obtained by interpolation using the same Lagrange polynomials. With such a discretization into elements, and with finite-dimensional functions we can then arrive at the *matrix-vector weak form*. This final equation can then be solved using standard linear algebra computer-aided techniques.

The steps involved in the strong-to-weak conversion are given in the following sections, using as an example the effective-1d system introduced in Section 2.1.3, and this approach can be relatively easily extended to 2d or 3d scalar problems, as seen in section 3.3.2. We further extend this to solve vector problems in Section 3.3.3.

---

[2] The *k*-th order 3d Lagrange polynomial associated with a nodal point labelled with index *j* is defined by: $L_j^{(k)}(x,y,z) \equiv \prod_{\substack{0\le m\le k \\ m\neq j}} \frac{(x-x_m)}{(x_j-x_m)} \frac{(y-y_m)}{(y_j-y_m)} \frac{(z-z_m)}{(z_j-z_m)}$ and has the Kronecker delta property such that

$L_j^{(k)}(x_i,y_i,z_i) = \delta_{ij}$

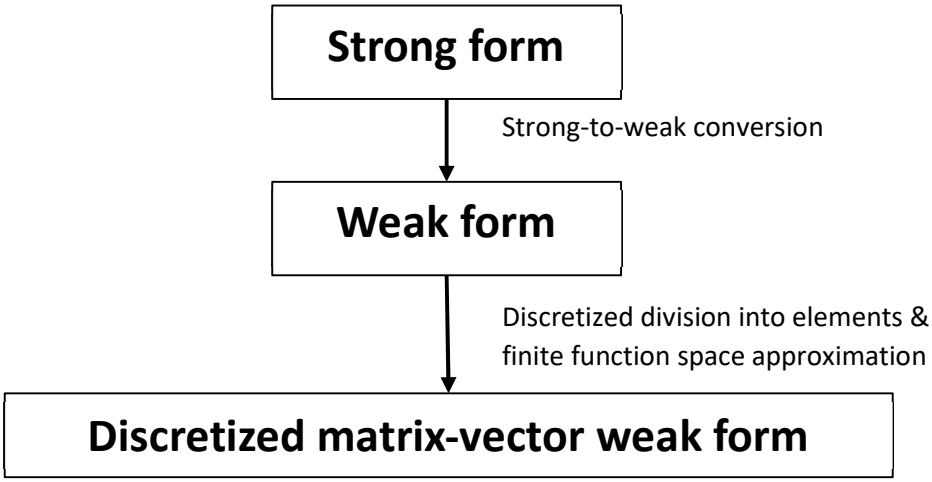

Figure 2.8: Flow chart outlining the steps from the strong form to the discretized matrix-vector weak form used within FEM.

2.2.1 Obtaining the weak form representation from the strong form representation

In finite element analysis, a physical system is first modelled by a set of PDEs and/or ODEs along with the boundary conditions, which is referred to as the strong form representation. This term "strong" is used because it imposes quite demanding differentiability and continuity requirements on the mathematical solution to the problem. For example, an *n*th-order differential equation requires a solution to be continuous in its *n*-th derivative. Conversely, the weak form only requires two integrals taken over (*n*-1)-th order terms to be equal. The strong form and weak form equivalently describe the physical system; both result in the same solution. The accuracy of the resulting is principally dependent on the number of elements and the order of the basis functions chosen.

In this section, we formulate the weak form for the physical example from Section 2.1.3 to demonstrate the basic steps of strong-to-weak conversion, we used the same approach used in reference [39]. The equation derived from the balance principle which applies at every material point $\vec{x}$ is:

$$\vec{b} + \vec{\nabla} \bullet (\underset{\equiv\equiv\equiv}{\mathbb{C}} : \underset{\equiv}{\varepsilon}) = 0 \qquad [2.42]$$

Here, we again model this body assuming no interactions between different directions, and no shear stresses applied to any boundary surface. The former means that the uniaxial strain along one direction is independent of that along any other (so Poisson's ratio is set to be to zero: $\nu = 0$), and the latter that all shear strain components are zero throughout the body. This also means the bar maintains its uniform cross section, *A*. The mechanical response of a more general case will be solved using a finite element approach in Section 3.3.3. Consequently, Equation 2.27b can again be reduced to

$$\frac{d}{dx_1}\left(E \frac{du_1}{dx_1} \right) + b_1(x_1) = 0 \qquad [2.43]$$

and we shall adopt the Dirichlet-Neumann boundary condition,

$$u(x_1 = 0) = u_0 = 0\,\text{m} \qquad ; \qquad t(x_1 = L) = E\varepsilon_{11} = t_0 \qquad [2.44]$$

We now determine the displacement of the bar in its deformed state along $\hat{e}_1$, where u_1 is labelled u and x_1 is labelled x below for convenience. The fixed Dirichlet displacement at the end of the bar is u_0; The fixed Neumann traction applied to other end of the bar is t_0. E is the Young's modulus. Equation 2.43 and 2.44 together represent the

strong form, we now convert this to the weak form for computing the finite element solution. To do this, we multiply Equation 2.43 by an arbitrary *weight function* (also known as a *test function*), $w(x)$, and integrate over the domain $\Omega \equiv [0, L]$

$$\int_0^L w(x) \bullet \frac{d}{dx}\left(E \frac{du(x)}{dx} \right) dx + \int_0^L w(x) \bullet b(x) dx = 0 \qquad [2.45]$$

Meanwhile, the weight function must be zero at every Dirichlet boundary, where the displacement solution is specified. Here, it manifests $w(0) = 0$.

Applying integration by parts to the first integral, we can write

$$\int_0^L w(x) \frac{d}{dx}\left(E \frac{du(x)}{dx} \right) dx = \left[w(x) E \frac{du(x)}{dx} \right]_0^L - \int_0^L \frac{dw(x)}{dx} E \frac{du(x)}{dx} dx \qquad [2.46]$$

Using the boundary condition, Equation 2.44, we have

$$\begin{aligned} \int_0^L w(x) \frac{d}{dx}\left(E \frac{du(x)}{dx} \right) dx &= w(L) E \frac{du(L)}{dx} - w(0) E \frac{du(0)}{dx} - \int_0^L \frac{dw(x)}{dx} E \frac{du(x)}{dx} dx \\ &= w(L)\, t_0 - \int_0^L \frac{dw(x)}{dx} E \frac{du(x)}{dx} dx \end{aligned} \qquad [2.47]$$

Inserting Equation 2.47 into Equation 2.45, we arrive at the final weak form:

$$\int_0^L \frac{dw(x)}{dx} E \frac{du(x)}{dx} dx = w(L)\, t_0 + \int_0^L w(x) b(x)\, dx \qquad \forall w(x) \text{ with } w(0) = 0 \qquad [2.48]$$

Note this is not restricted to a *homogeneous* linear-elastic bar; the value of E can here vary with x. We can see the strong form, Equation 2.43, involves a second derivative of $u(x)$, requiring that $u(x)$ need be continuously differentiable until at least its second

derivative. The weak form reduces its differential order by one and has a lower requirement on the continuity.

The final weak form is equivalent to the strong form; the strong form is implied by the weak form and can be constructed from its weak form by reversing the steps. That equivalence allows us to obtain the solution for $u(x)$ from the weak form. To establish a finite element approach numerically, the physical body needs to be divided into elements, and functions $u(x)$ and $w(x)$ need to be represented in a finite dimensional space.

2.2.2 Meshes, elements, nodes, and mapping between the physical space and reference space

The process of discretization of the physical body into a finite number of elements is called *meshing* , and for each element, Ω_e , a local solution, u_e , is desired. These solutions are then assembled together and result in the global solution for the entire body. A properly generated mesh and suitable type of element together ensure more accurate finite element solutions.

A *mesh* (sometime referred to as a *grid*) is the collection and connection of elements that together construct the physical body, which represents the geometrical and topological information, respectively. In a continuum system, mass completely fills the physical space it occupies and there are an infinite number of points which makes the system not solvable using numerical methods. Meshing simplifies the continuous body by treating it as a set of discrete elements, and this allows us to apply FEM to obtain a numerical solution.

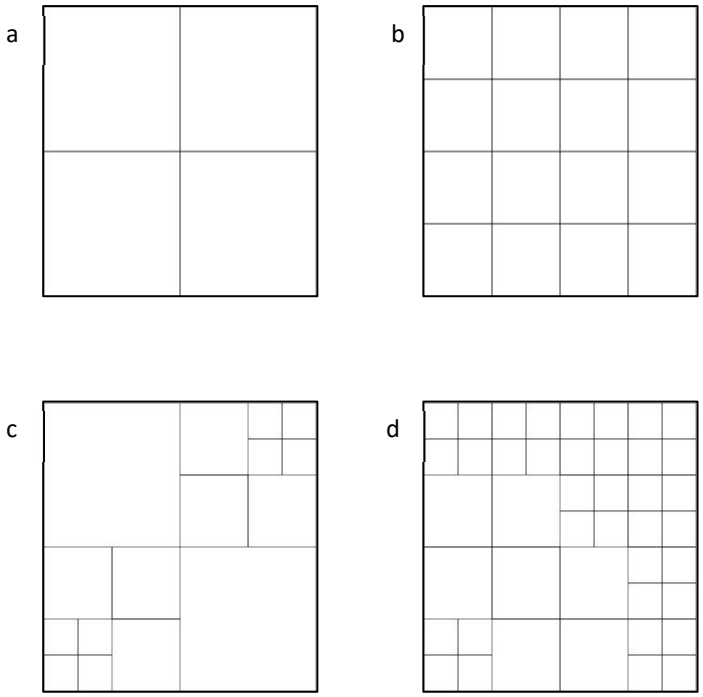

Figure 2.9: Meshes generated on a 2d square object using quadrilateral elements in deal.II. (a) structured 4 element mesh, (b) structured 16 element mesh, (c) unstructured 16 element mesh, and (d) unstructured 43 element mesh.

As one of the major contributors to FEM accuracy, a mesh refined with a proper strategy achieves a good simulation. There are two types of mesh: structured and unstructured. The structured mesh, historically related to the finite difference method, is uniformly generated over the physical body, see Figures 2.9(a) and (b). The unstructured mesh is more general and constructed based on the relative physical importance, see Figures 2.9(c) and (d). An unstructured mesh is locally refined to be suitable for cases with complex internal geometries; and concentrates the simulation at the regions of interest. For instance, in Figures 2.9(b) and (c), a 2d square body is divided into 16 quadrilateral elements and generated through a structured or an unstructured mesh, respectively. For a diagonally-loaded problem, the unstructured mesh shown in (c) can

provide a more precise simulation. Higher accuracy can automatically be produced by generating a high mesh density (the number of elements per area or volume), but that usually takes a longer time to simulate. Constructing different mesh densities within different areas can help by saving processing time and a more physically realistic mesh leads to a more accurate finite element solution.

Figure 2.10 shows examples of different types of element in 1d, 2d, and 3d. In 3d cases, complex geometries are better fitted using tetrahedral elements; but a basic hexahedral element comprises more nodes which requires fewer elements to obtain the same degrees of freedom since a hexahedron can be made up from numerous tetrahedrons. Thus while a tetrahedral mesh suits arbitrarily-shaped geometries, for a given mesh density and a uniform geometry, a hexahedral mesh is more accurate.

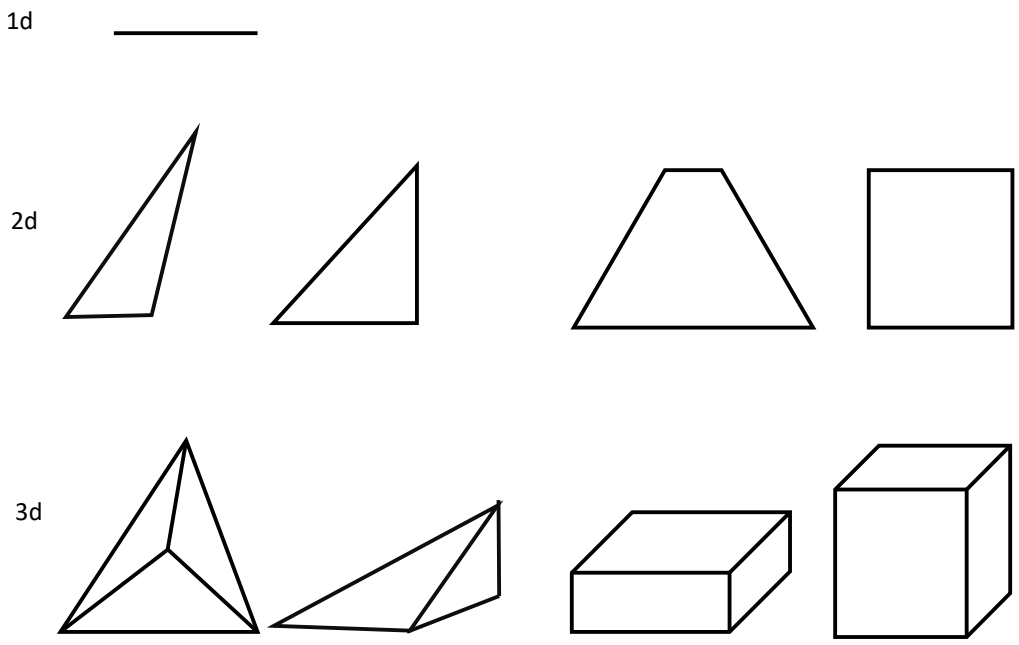

Figure 2.10: Elements in 1d, 2d, and 3d. Top row: a 1d element. Middle row: 2d triangular and quadrilateral elements with irregular and regular shapes, respectively. Bottom row: 3d tetrahedral and hexahedral elements with irregular and regular shapes, respectively.

Nodes specify the locations of the degrees of freedom where the finite element solution u^h, is to be calculated and the number of nodes within the element is related to the order of the basis functions adopted. Moreover, several scalar degrees of freedom (dofs) can be defined at a single node, for instance, a thermal simulation of a deformable body could involve four dofs per node which represent the (*x,y,z*) displacements and the temperature at each nodes.

Regarding the terminology of nodes, *external nodes* lie on the vertices, and may be also lie on the edges and/or faces of an element, and additionally there may be *internal nodes* located inside each element. The approximate solutions (represented by dofs) are only computed at nodes. A hexahedral element with eight nodes is usually called a *Hex8* element and its interpolation of dofs between nodes are done by first-order basis functions. Nodes are located on the vertices of this element and they are external nodes commonly shared with neighboring elements. The values of the approximate solution at the common nodes are consistent with the solutions for each of its connecting elements and this continuous relationship ensures the correct assembly of the global solution. Nodes are placed only at element vertices when using *linear* (i.e. 1d space), *bilinear* (i.e. 2d space), or *trilinear* (i.e. 3d space) basis functions. When higher-order basis functions are introduced, additional nodes can be placed along the edges of an element, on the surfaces, or inside the element. Hexahedral elements that require a total number of 20 nodes to be modeled with second order basis functions are called *Hex20* elements. These nodes are then located at eight vertices and each midpoint of the 12 edges in a Hex20 element. Typically, a higher-order function helps in capturing more complex geometries and achieving more accurate computations. There can also exist some surface and/or

internal nodes that correspond to even higher-order basis functions. In this study, we investigate the mechanics of an object with a non-complex geometry, such as a half-spherical shell, and so a hexahedral mesh with first-order basis functions promises both accuracy and efficiency of simulation. In addition to elements with straight edges, elements with curved edges can be constructed if nodes exist on the edges, and such elements are known as *isoperimetric elements*. This allows us to better describe bodies with curved boundaries.

Nodes can be identified by the elemental coordinate within a specific element, in which case this is called a *local node representation*, or by using a global coordinate, called a *global node representation*. For a 1d body divided into *n* elements as shown in Figure 2.11, node 1 in all elements Ω_i, except i =1, is a common node shared with its neighbor Ω_{i-1}, and can also be read as the *i*-th node of the physical body using a global node representation. The local node representation helps with calculations at a local level, particularly those that involve local solutions for a specific element, u_e^h.

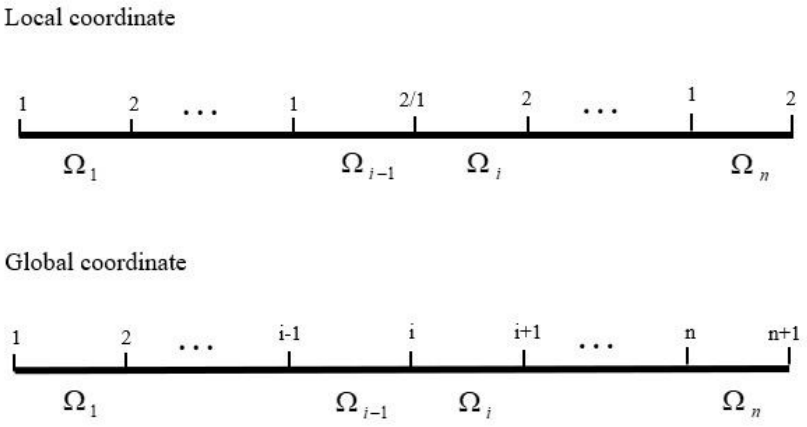

Figure 2.11: A depiction of the local and global node representations, and their numbering for a 1d body.

The elements that comprise the physical body typically exhibit an irregular shape for which required integrals are difficult to solve. *Mapping* is the transformation which maps points between the physical space and a *unit* (specifically between 0 and 1) or *bi-unit* (specifically between -1 and $+1$) reference space in order to perform the calculations on a regular shape. Figure.2.12 shows such a mapping for a general quadrilateral element, Ω_e. Such a transformation of functions between the physical space and the reference space can be mathematically described by using the *Jacobian matrix*[3], $\underline{\underline{J}}$. This square matrix is defined as the gradient of the physical space position vector with respect to its reference space coordinate which can be used to calculate the local approximation of points in a mapping. The volume change of the element in the transformation is given by the determinant of the Jacobian matrix.

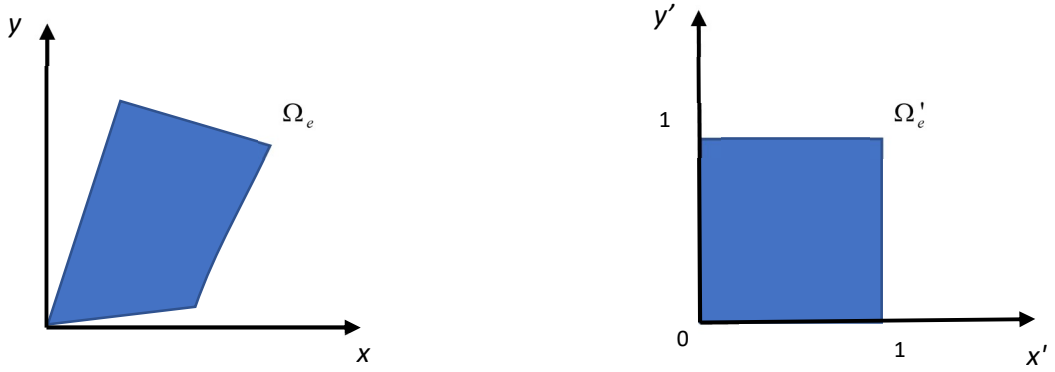

Figure.2.12: The mapping of a quadrilateral element in physical space, Ω_e, and in unit reference space, Ω_e'.

[3] In a 2d space, for example, the Jacobian matrix for the mapping transformation can be written as

$$\underline{\underline{J}} = \begin{bmatrix} \frac{\partial x}{\partial x'} & \frac{\partial x}{\partial y'} \\ \frac{\partial y}{\partial x'} & \frac{\partial y}{\partial y'} \end{bmatrix}$$

Note that this Jacobian matrix $\underline{\underline{J}}$ is not the same as the scalar volume ratio introduced in Section 2.1.1

For a solution to be obtained, the boundary conditions need to be included. For both scalar and vector problems, there are three common types of boundary conditions: Dirichlet, Neumann and Robin. A Dirichlet boundary condition specifies the scalar or vector value that the solution takes on a surface, for example, a point on a body that is deformed to a known, fixed, displacement. A Neumann boundary condition specifies the derivative of the solution along a particular direction, for example, a point on a surface experiencing an applied traction. The Robin boundary condition is a weighted combination of Dirichlet and Neumann. FEM solves boundary value problems and a unique, complete, solution requires at least one Dirichlet boundary condition to be applied to the system.

2.2.3 Basis functions and degrees of freedom

After discretizing the physical body into elements, $\{\Omega_e\}$, the approximate solution for each element, u_e^h, is taken to span a finite function space, ϕ_i, containing the *basis functions*. Discretizing the body with a reasonable mesh refinement enables us to use low-order basis functions to obtain an accurate solution.

The degrees of freedom at each node for each element represent the solutions $\{u^h{}_e\}$ and are the approximate physical response at these nodes. In contrast to external nodes, internal nodes are not shared between different elements. In some of the literature, the term ‘degree of freedom’, is used for the number of independent coordinates in phase space that describes an element [42, 43]. In our study here, it is convenient to instead use

this term to describe the scalar components of the solution at each node, and this is consistent with the notation used in the deal.II finite element library [42]. Thus, a purely-thermal simulation needs one dof at each node to represent the temperature, while a deformation simulation typically needs three dofs per node to represent the nodal translations in three coordinate directions.

In order to solve the weak form for our effective-1d example, Equation 2.48, we interpolate between nodes using the predefined basis functions that cover each entire element. The basis functions can take many forms, including polynomials and trigonometric forms. Of these the polynomial form is the most appropriate for interpolation in many applications [45]. It is comparatively easy to interpolate points using polynomials; also, the derivative and integral of polynomials can be readily found. Herein, Lagrange polynomials are used as the basis functions.

Continuing with our example, each finite element solution sought for the *e*-th element can be written as a linear combination of degrees of freedom and basis functions as follows:

$$u_e^h(x) = \sum_{i=1}^{n_{\text{nodes},e}} d_i^e \phi_i(x) \qquad [2.49]$$

where $u_e^h(x)$ is the approximate *e*-th element solution, $\{\phi_i(x)\}$ are the chosen basis functions, $n_{\text{nodes},e}$ is the number of nodes in the *e*-th element, and d_i^e is the degree of freedom at the *i*-th node for this solution. The *global* finite element solution, $u^h(x)$, will be summed over the individual contributions from each element. The total number of basis functions for each dimension of the desired solution equals the number of global

nodes that comprise the body, n_{nodes}. Should higher-order basis functions be adopted, then the total number of nodes, n_{nodes}, increases accordingly.

As a concrete, first-order, 1d example, the Lagrange basis function associated with the *i*-th internal node takes the form:

$$\phi_i(x) = \begin{cases} \dfrac{(x - x_{i+1})}{(x_i - x_{i+1})} & x_i < x \le x_{i+1} \\ \dfrac{(x - x_{i-1})}{(x_i - x_{i-1})} & x_{i-1} \le x \le x_i \\ 0 & \text{elsewhere} \end{cases} , \qquad [2.50]$$

where for an external node we would only adopt either the first or second line as appropriate, and the last line. Basis functions are often referred to as a *shape functions*, but a subtle difference between the two exists as a shape function only encompasses a single element, Ω_e, and adopts local node numbering so the corresponding shape function here is:

$$N_1^e(x) = \frac{(x - x_2)}{(x_1 - x_2)} \qquad x_1 \le x \le x_2 \qquad . \qquad [2.51]$$

Figure 2.13 illustrates these functions and difference between them.

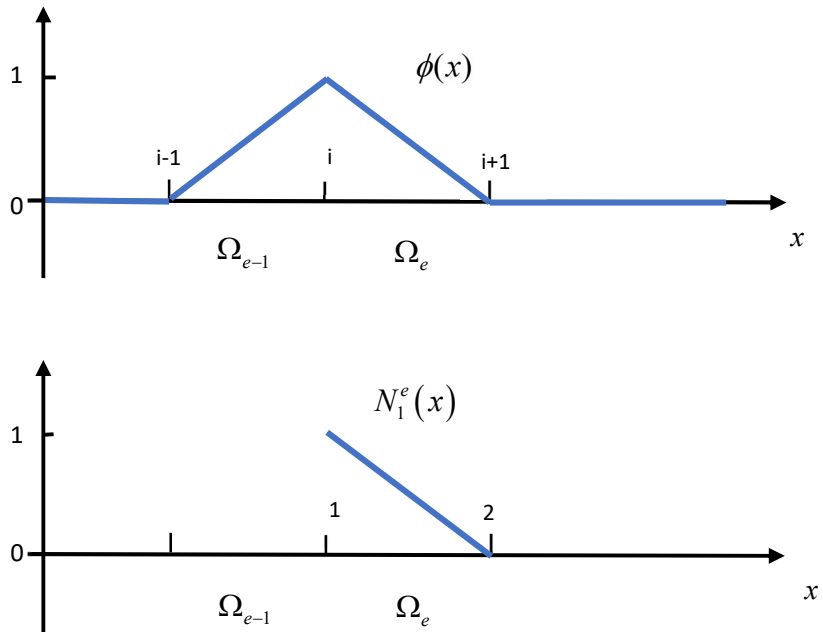

Figure 2.13: A first-order basis function (top) and shape function (bottom) associated with the *i*-th global node for a 1d body.

Elements in physical space are often irregular, rather than regular tetrahedrals or hexahedrals, etc. It is convenient to compute values in the reference space and later transfer the computation through mapping. To illustrate this concept, as shown in Figure 2.14, we consider the following first-order shape function for element *e* using local node numbering in a unit reference space $x' \in [0,1]$

$$N_1^e(x') = 1 - x' \qquad [2.52]$$

The second shape function, associated with the node labelled 2 at $x' = 1$, can be written as

$$N_2^e(x') = x' \qquad . \qquad [2.53]$$

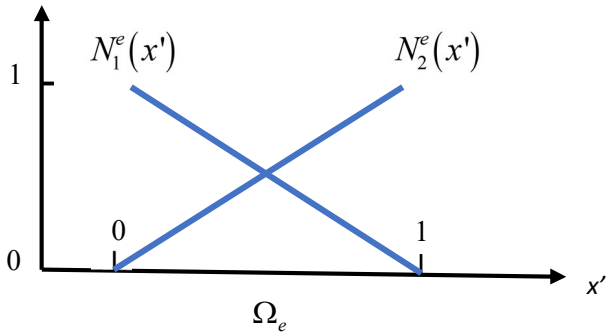

Figure 2.14: First order shape functions for element Ω_e in a 1d unit reference space, $x' \in [0,1]$.

Basis functions satisfy the following properties which make them particularly suitable for finite element calculations. The Kronecker delta property requires the basis function to equal unity at its associated node and zero at all other nodes, $\phi_i(x_j) = \delta_{ij}$. Further, as seen from Equation 2.50, the sum of basis functions at any position x or x' within an element Ω_e is unity, $\sum_i^{n_{nodes}} \phi_i(x) = 1$.

The global finite element solution is formed by summing the dofs and basis functions across all elements

$$u^h(x) = \sum_{e=1}^{n_{el}} u^h{}_e(x) = \sum_{i=1}^{n_{nodes}} d_i \phi_i(x) \quad , \qquad [2.54]$$

where n_{el} is the number of the elements, and n_{nodes} is the number of global nodes. This will be imposed on the weak form to perform the numerical calculation.

2.2.4 Deriving the matrix-vector equations

Here we present the weak form in matrix-vector form to then be solved using the tools of linear algebra.

Continuing our effective-1d example from Section 2.2.1, we partition the entire region of interest, Ω, into a set of elements, $\{\Omega_e\}$, and equation 2.48 evolves from integrating over the entire body to sums of integrals over individual elemental volumes, Ω_e,

$$\sum_e \int_{\Omega_e} \frac{dw^h(x)}{dx} E \frac{du^h(x)}{dx} dx = w^h(L)\, t_0 + \sum_e \int_{\Omega_e} w^h(x) \bullet b(x)\, dx \qquad \forall w(x) \text{ with } w(0) = 0 \qquad [2.55]$$

Meanwhile, the *e*-th elemental approximate solution, $u_e^h(x)$, is presented as Equation 2.49, where, as described above, we choose to use linear Lagrange polynomials, Equation 2.50, for the simplest, first-order case. The corresponding shape functions in the unit reference space read

$$N_1(x') = 1 - x' \quad , \quad N_2(x') = x' \qquad [2.56a,b]$$

Similarly, the weight functions are also approximated in the same function space

$$w_e^h(x) = \sum_{j=1}^{n_{\text{nodes},e}} c_j{}^e \phi_j(x) \quad , \qquad [2.57]$$

where $\{c_j{}^e\}$ is the set of the arbitrary coefficients for the weight function, while the weight function coefficient, $c_{x=0}$, relating to the node at $x = 0$ has to be set to zero in order to satisfy that the weight function $w(x = 0) = 0$.

After substitution, we arrive at the discretized weak form in a finite function space with the constraint for all weight functions subject to Dirichlet, $c_{\text{node at }0} = 0$ at $x = 0$

$$\sum_e \int_{\Omega_e} \sum_{i,j}^{n_{\text{nodes},e}} c_j \frac{d\phi_j}{dx} E \frac{d\phi_i}{dx} d_i^e \, dx = c_{\text{node at } L} t_0 + \sum_e \int_{\Omega_e} \sum_j^{n_{\text{nodes},e}} c_j \phi_j \, b(x) \, dx \qquad [2.58]$$

This can be represented in a matrix form

$$\underline{c}^T \underline{\underline{A}} \, \underline{d} = \underline{c}^T \underline{f} \qquad [2.59]$$

The arbitrariness of the weight function coefficient vector, $\underline{c}$, enables Equation 2.59 to be simplified into the form

$$\underline{\underline{A}} \, \underline{d} = \underline{f} \quad , \qquad [2.60]$$

where $\underline{\underline{A}}$ is generically called the *system matrix*, $\underline{d}$ is the dof (or *solution*) *vector*, and $\underline{f}$ is the *forcing vector*, with global node numbering

$$A_{ij} = \sum_e \int_{\Omega_e} \frac{d\phi_j}{dx} E \frac{d\phi_i}{dx} dx \quad ; \quad \underline{d} = \begin{pmatrix} u^h(0) \\ u^h(x_1) \\ \vdots \\ u^h(L) \end{pmatrix} \quad ; \quad \underline{f} = \begin{pmatrix} f_1 \\ f_2 \\ \vdots \\ f_{n_{nodes}} + t_0 \end{pmatrix} \quad ; \quad f_j = \sum_e \int_{\Omega_e} \phi_j b(x) dx$$

[2.61]

In finite element analysis, these matrix and vector components are first established at the nodes for each individual element. This is, split Equation 2.61 into elemental form

$$A_{ij}^e = \int_{\Omega_e} \frac{d\phi_j}{dx} E \frac{d\phi_i}{dx} dx \quad ; \quad f_j^e = \int_{\Omega_e} \phi_j b(x) dx \qquad [2.62]$$

A complete calculation then requires solving the *global matrix-vector equation* (2.60) which comes from an assembly of the elemental integrations and represents the entire body. For a scalar problem in 1d, 2d, or 3d, vector $\underline{d}$ contains as many elements as there are nodes, n_{nodes}, with each component being either the desired degree of freedom or a Dirichlet boundary condition. The forcing vector $\underline{f}$ represents the external force consisting of the 'body force', $b(x)$, and any Neumann boundary conditions. The global system matrix $\underline{\underline{A}}$ is derived from the constitutive relation that describes the material properties, which in general stems from the stiffness tensor, $\underline{\underline{\underline{\underline{\mathbb{C}}}}}$, quantifying the stress-strain relationship, and specifically for this assumed-isotropic bar with only effective-1d deformation encompasses only the Young's modulus.

Equations 2.60 and 2.61 directly apply to an effective-1d system as introduced above. To find a scalar solution instead for a 3d body with n_{nodes} nodes, $\underline{\underline{A}}$, $\underline{d}$, and $\underline{f}$ keep the same form, with the elements of $\underline{d}$ still being the *scalar* solutions at each node, and the elements of $\underline{f}$ and $\underline{\underline{A}}$ being integrals over elemental volumes. To find instead a *vector* solution for a 2d or 3d body, the strong form starts from a vector governing pde and still leads to Equation 2.60 where the components are themselves either vectors ('the vector form') or $\underline{\underline{A}}$, $\underline{d}$, and $\underline{f}$ are each expanded in size to encompass all vector component ('the sum of components form'). The mathematics presented in Section 3.3.3 and Chapter 4 adopts the first of these approaches, while the actual C++ code that executes the computations migrates this into the second approach. As seen from Equation 2.58, for the

example of a scalar problem, we here choose to label the nodes within the weak form by *i* and *j*, while for vector problems below we shall label the nodes by *n* and *m*, with *i,j,k,l* reserved for labelling the spatial orientations.

We shall next manipulate the global matrix-vector equation (2.60) such that the constrained linear system satisfies the boundary conditions. The boundary conditions from equation 2.44 have to be imposed on the associated elements, which in this case are the first and the last elements of the body. The end nodes in the global representation, $i=1$ and $i = n_{\text{last}}$, must therefore satisfy the Dirichlet and Neumann boundary condition, respectively,

$$u^h(0) = d_1 = 0 \quad , \quad \frac{d}{dx}\left(Eu^h\right)_{x=L} = t_0 \quad . \qquad [2.63a,b]$$

The numerical calculation for this effective-1d system will be done in Section 3.3.1.

This matrix-vector approach is remarkably broad and can represent widely-different physical systems as described in Table 2.2.

Table 2.2: Physical implementations of the finite element matrix-vector weak form.

	System matrix, $\underline{\underline{A}}$	**Solution vector,** $\underline{d}$	**Forcing vector,** $\underline{f}$
Material Deformation	Stiffness	Displacement	Force
Electrostatic	Dielectric permittivity	Electric potential	Charge density
Thermal	Conductivity	Temperature	Heat source

Chapter 3

Finite Element Modelling Using the deal.II Library

3.1 Introduction to scientific computing, and C++ libraries

3.1.1 The general approach to scientific computing

Real-life problems are often impossible to solve analytically. The impact of computer science is continuously growing in many fields of research and computers are commonly employed to investigate those real-world questions. As shown in Figure 3.1, there are basically three steps involved in solving problems by scientific computing. First, the physical system is described using fundamental concepts, laws, etc. Second, a mathematical model is established which is suitable for the complex problem. Last, computer simulation is invoked for processing the mathematical model and generating the results for the physical system. Often, reliable and meaningful solutions can not be achieved before repeatably adjusting and validating the model.

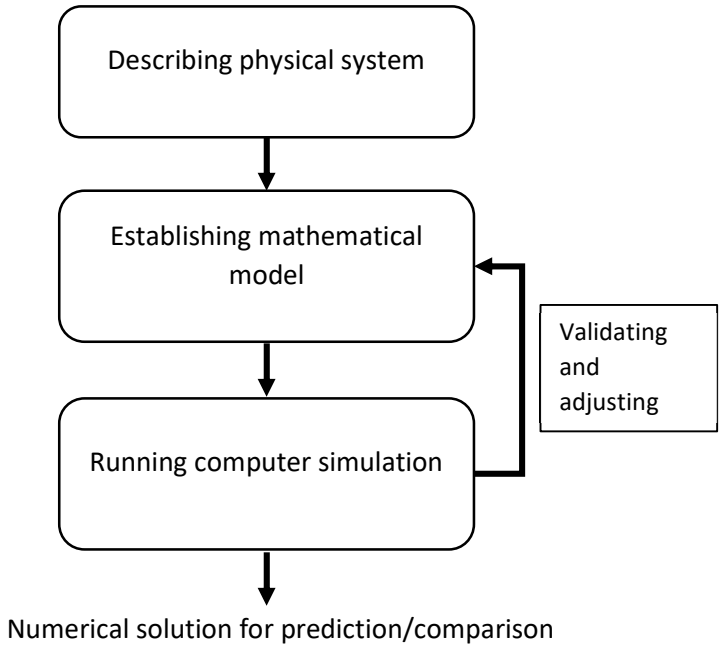

Figure 3.1: Scientific computing process

An important simple example in continuum solid mechanics is the effective-1d system first introduced in Section 2.1.3: namely, the uniaxial deformation of a uniform-area, linear-elastic bar with given material properties and boundary conditions. Different portions of the bar will typically have varying physical responses during the deformation. The analytical solution was shown in Section 2.1.3. However, problems involving higher dimensions, and bodies with either a geometry and/or a material property that is nonuniform, become more difficult or impossible to solve analytically. Computer-aided simulation can then be applied to predict the deformation of such a body.

In this thesis, we wish to investigate the biomechanical properties of the human cornea during its deformation due to varying IOPs and external pressures. We adopt deformation theory and assume particular constitutive relations to model the mechanical response. Hyperelastic models , e.g. Neo-Hookean or Mooney-Rivlin [33], are commonly used to

represent the nonlinear corneal material properties. We use the Finite Element Method (FEM) to solve the resulting governing partial differential equation with its associated boundary conditions. The C++ finite element library *deal.II*, version 9.0.1, has been used for the calculations in this thesis. The accuracy of the FEM solution depends upon the type of elements and the total number of elements adopted. A precise simulation result requires a large number of elements and suitable element shapes, but an over-refined mesh can lead to increased computational expense.

3.1.2 Developing a project in C++

As an extension of the C programming language, C++ is an object-oriented programming (OOP) language and is usually viewed as a medium-level language. Within C++, a *classes* is a set or collection of member functions and data, for example a simple class might be: {cosine, sine…; π, e,…}. An *object* is the particular application of members within a class, thus much C++ programming involves invoking the objects required to access a class and thus conduct the desired task.

A basic C++ program is derived from a file consisting of *source code*, typically with .cc extension, and various header files, with .h extensions. Most C++ programs require also specific C/C++ *libraries* in order to produce the functional executable, with .exe extension. Libraries are generally meant to be accessed by many users. They usually comprise two parts: first, a set of text-based header files (for example the general C++ library *math.h*, *iostream.h* files and other library-specific .h files) which defines the interface/access to the corresponding library and enables compilation of the source code,

and second, a set of precompiled object files (with .o extension) containing the machine code of the library, which enables an increase in overall efficiency when compiling, updating, etc., and also maintains the library's integrity. There are two types of library: static and dynamic. A *static library* is loaded into the program at the time of linking and becomes part of the executable. A *dynamic library* is not loaded into the program until the time of running, and so it exists as a separate file. Using a dynamic library helps in reducing the size of an executable, and the dynamic library can be easily updated without recompiling the entire source code.

Figure 3.2 illustrates the steps in performing a calculation using C++ libraries. The C++ source code and header files are written for the desired calculation and together sent to the compiler for conversion into machine code which is stored as an object file. The linker then combines this with other *global object files* and *static library files* ('.a' extension denoting 'archive' in Linux) to form a single executable. When running, this executable is coupled with dynamic library files (with '.so' extension, denoting 'shared object' in Linux) and loaded into the computer hardware to produce the desired output.

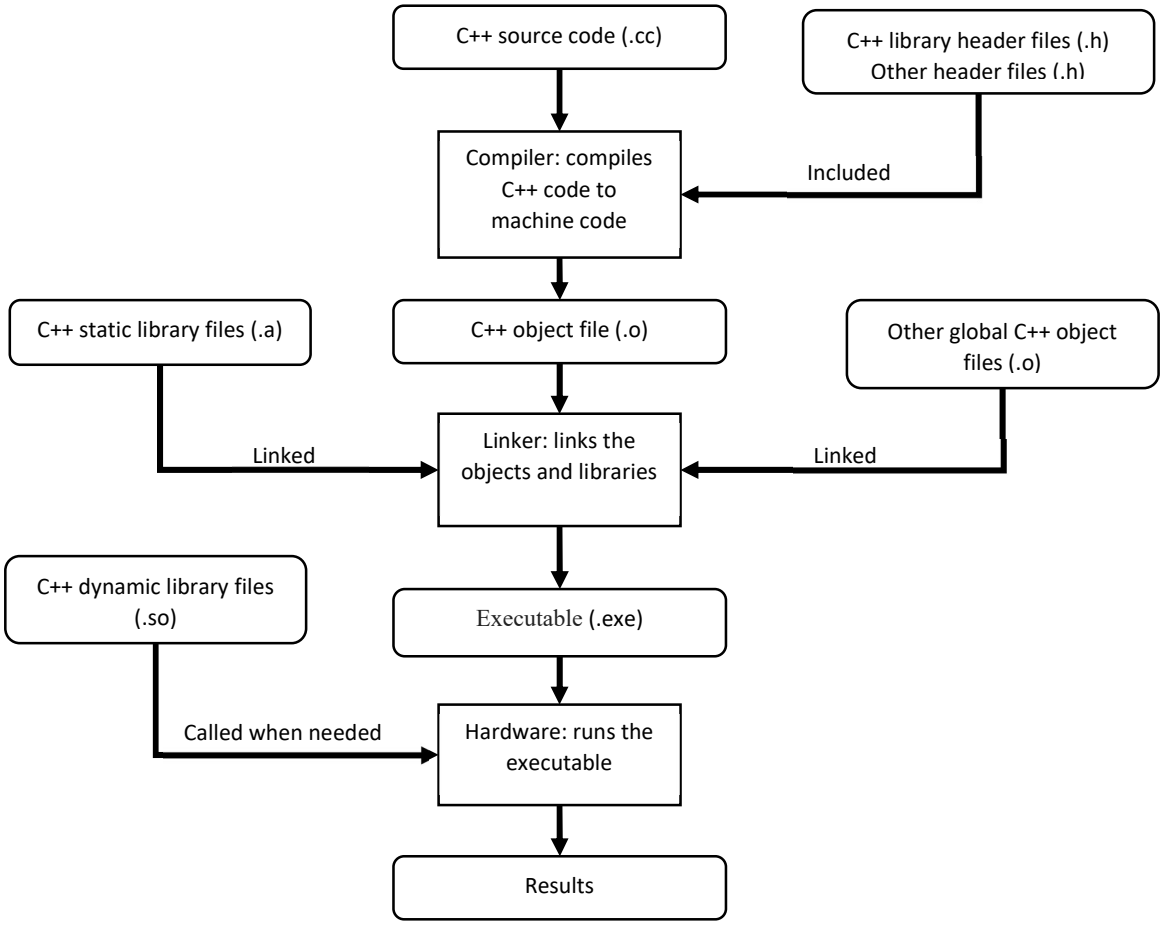

Figure 3.2: Flow chart depicting a C++ project, from source code to an executable. A program contains the elements in the middle column, i.e. the source code and all required libraries.

3.2 Introduction to deal.II

3.2.1 An overview of deal.II

The C++ library *deal.II*, also known as the Differential Equations Analysis Library, is a finite element method library written in C++ for solving partial differential equations [42]. It evolved from a FEM library called *DEAL* for initially defining meshes, and the

release used in this thesis is version 9.0.1 from October 2018. The deal.II library is supported by a variety of operating systems, including GNU/Linux, MacOS X and Windows. However, deal.II is mainly developed within a Unix-like environment using the GNU Compiler Collection (GCC) and so it is easier to install and run deal.II projects on a Linux system. This library is a precompiled dynamic library which can be upgraded without modifying any existing executables in a program.

A basic user-built deal.II project is composed of C++ source code and a managing script file called *CMakeList.txt* , which identifies the project's source file and target files for the program *Cmake* that manages the compiling. The GNU *make* program compiles the source code and header files generated by *Cmake*, and links these with the library files to then form a single executable. Like other dynamic libraries, parts of the deal.II object files are declared within a number of the library header files, where each header file calls the respective portion of the library .so object file.

Useful documentation is provided on the deal.II website, *www.dealii.org*, in three different levels: the *tutorial* package which contains a collection of tutorials with full source code (eg. step-1 tutorial, step-2 tutorial, etc.); the *manual* which provides descriptions of all classes and their member functions; and the *modules* which describe particular groupings of classes and functions that are used together to perform discrete tasks such as allocating dofs to elements and element mapping between the physical space and reference space.

All files that comprise deal.II are available from its website, and those files are stored in a series of directories with the top-level names shown in Figure 3.3. The library configuration files for installation can be found in the first website directory named '*deal.II'*. The second website directory, also named '*deal.II'*, contains the library header files. When specific header files are required to be included within the source code, that contain classes that are needed, they are organized within the deal.II library on the user's computer according to the directories. Examples of such directories include *grid*, *dofs*, *fe*, *numerics* and *lac*: the first of these contains header files relevant to geometry definition and mesh generation, the second relevant to dof management, the third relevant to the description of elements that comprise the body, the fourth relevant to data output and the last required for linear algebra procedures. Two website directories named '*doc'* contain extensive documentation such as tutorials, example code etc. The deal.II library source code is listed in the website directory named *'source'*. In 2018, the library in total exceeded 1.1 million lines of code [45].

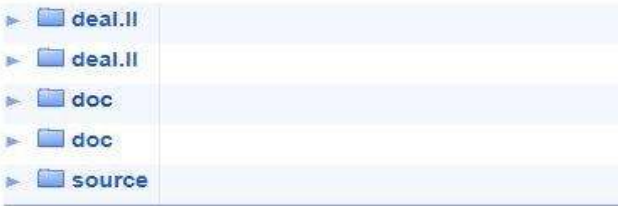

Figure 3.3: the deal.II directory listing from http://www.dealii.org

As seen in Figure 3.4, applying FEM to solve a physical system using deal.II can be divided into the following seven steps. For *preparation* outside of deal.II, we

1. physically model the system of interest as a set of partial differential equations and imposed boundary conditions;

2. define the geometry of the body in anticipation of mesh generation;

3. derive the discretized matrix-vector weak form.

For *processing* within deal.II, we

4. create a mesh based on the geometry of the body and select element properties such as element type and number of nodes per element;

5. conduct the required elemental calculations and assemble the elemental matrices into a global one;

6. solve the global matrix-vector equation with the linear algebra solver and then output the solution.

For *postprocessing* of the finite element solution outside deal.II, we

7. import the solution file into visualization software and analyze the result.

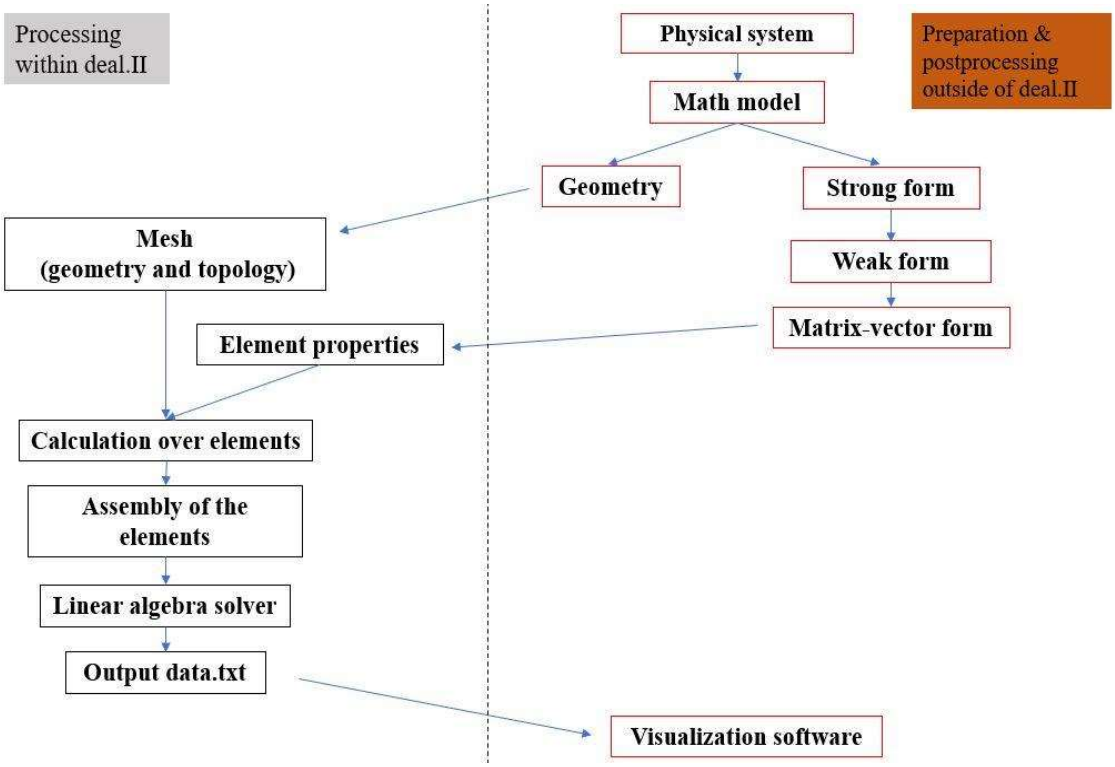

Figure 3.4: Processing flow chart for a finite element calculation using deal.II.

The processing steps (i.e. 4-6) typically involve various groupings of deal.II classes as shown in Figure 3.5. The *Triangulation* class allows us to build the geometry of the physical body and generate the mesh. Element properties, such as shape functions and number of nodes per element, are described within the Finite Element classes. The degrees of freedom (dofs) are linked to their associated nodes by the *DoFHandler* class. The *Mapping* class provides a transformation between elements in real space and reference space. The Quadrature classes set the parameters for the numerical integrations, such as weighting values and coordinates of quadrature points. The *FEValues* class provides the values at the quadrature points; examples include values of the basis functions, gradients of the basis functions, etc. Local, single element, nodal values (such as coordinates, displacements, forces, etc.) have to be combined with those of neighboring elements to create the global matrix-vector equation. This assembly can be

done through the group of Linear Algebra classes, which implicitly assumes that values at shared nodes are consistent. This global matrix-vector equation is then solved using the appropriate linear solvers. The numerical values and parameters for solution visualization are exported by the *DataOut* class in suitable formats [42].

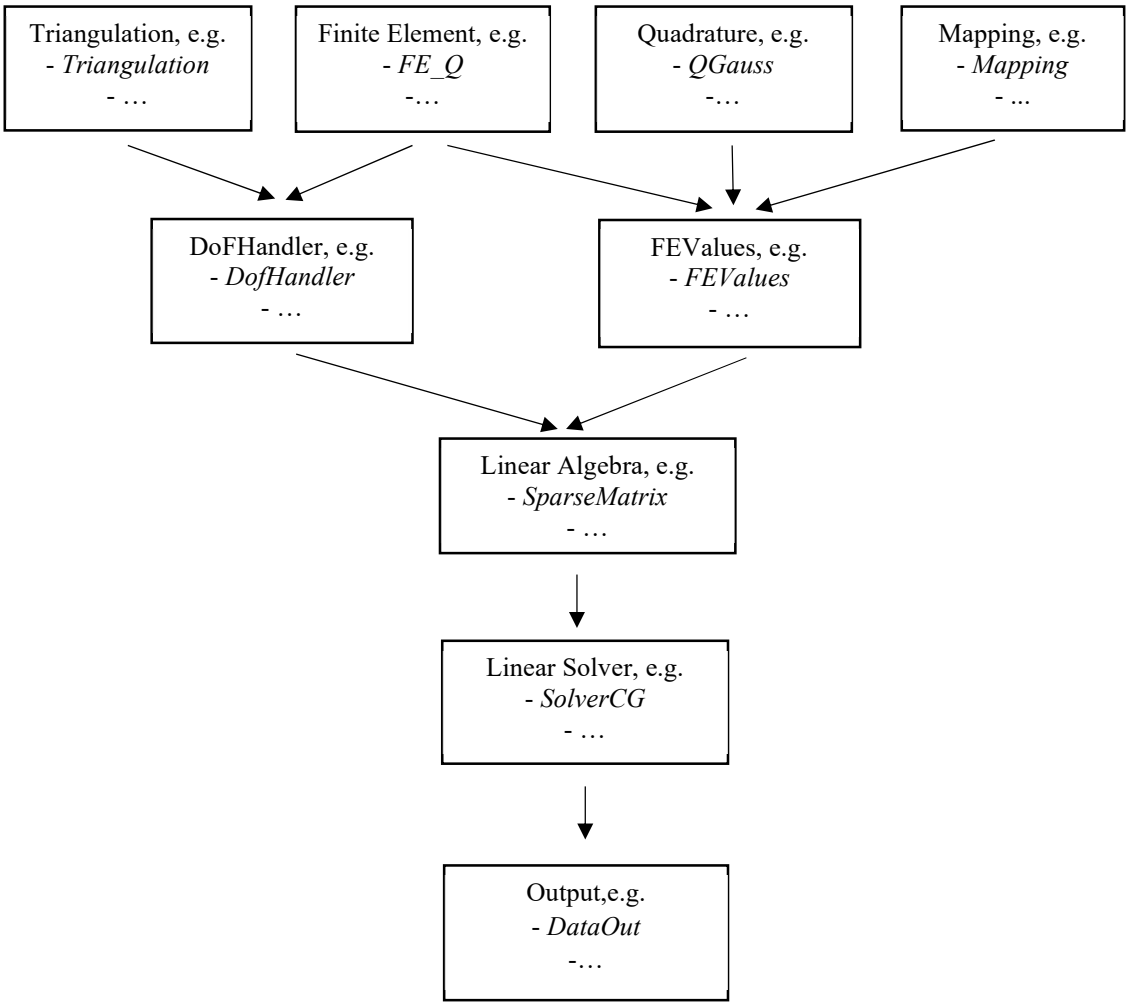

Figure 3.5: Broad categories of the steps involved in finite element analysis with examples of specific deal.II classes given in italics. (Adapted from http://www.dealii.org)

3.2.2 Describing key functions within the deal.II library

Here, we introduce some key deal.II functions that we call within an example of procedural source code, such as given in Code Excerpt 3.1. Most functions in deal.II are written in a dimension-independent way [46] such that the functions include a spatial dimension parameter. This programing feature allows us to apply the same code to simulate the system in different dimensions.

The *main* function within our example source code is made up of five functions that we name as follows: *meshing*, *setup_system*, *assembly*, *solve*, and *output*, summarized using pseudo code as follows:

```
int main ( ) {
      meshing ( );
      setup_system ( );
      assemble_system ( );
      solve ( );
      output ( );
      return 0;
      }
```

Code Excerpt 3.1

These functions, which call some key deal.II functions, are reviewed in the following.

a) The *meshing* function

Here we set the geometry and mesh information for the body of interest, by invoking the *Triangulation* class. The term *Triangulation* is used for historical reasons as triangular elements were initially used in FEM. Presently, deal.II only implements meshes which

are made from hexahedral elements (line segments in 1d, quadrilateral elements in 2d, hexahedral elements in 3d). Tetrahedral elements (i.e. triangles in 2d and tetrahedrals in 3d) are not supported. We shall illustrate examples of a half-ring or half-spherical shell constructed using curved hexahedral elements with a structured meshing, as this is the primary geometry/topology that we shall use when modelling the cornea.

Geometry creation and meshing can be completed in three ways: meshing for common geometries can be created using deal.II functions provided in the *GridGenerator* class; meshes can also be created externally within preprocessing software to provide a readable mesh file for input into deal.II; and finally, meshing can be performed by manually building up a data structure outside deal.II which defines the vertices and edges of elements. Information about the geometry and mesh is stored in the *triangulation* object that is created from the *Triangulation* class. The step-1 and step-2 deal.II tutorials provide two examples of using this class to describe a unit square and two half-rings, both filled with quadrilateral elements as shown in Figure 3.6.

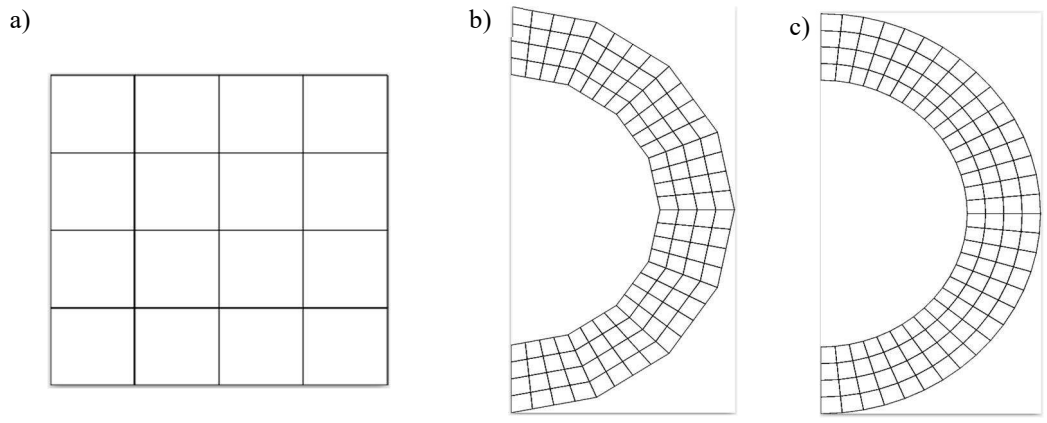

Figure 3.6: Meshing with quadrilateral elements in a) a square, b) a Cartesian- and c) a spherical-manifolded half-ring. This last example demonstrates isoperimetric elements.

An example of using the meshing function is shown in Code Excerpt 3.2 for a half ring in 2d (by setting dim=2) or a half-spherical shell in 3d (by setting dim=3):

```
void meshing(){
      …
      Triangulation<dim> triangulation;
      GridGenerator::half_hyper_shell(triangulation,
                                      shell_center,
                                      shell_radius_in,
                                      shell_radius_out,
                                      n_gridcells);
      …
      }
```

Code Excerpt 3.2

The inner and outer surfaces are defined with respect to a center position, *shell_center*, and their respective radii, *shell_radius_in* and *shell_radius_out*. In general, the mesh based on a Cartesian coordinate system may not be a good description for curved geometries, such as those in Figure 3.6. A non-Cartesian description can be found within the *Manifold* class and used to construct elements with curved edges, as seen in Figure 3.6(c).

At this stage in Code Excerpt 3.1, the necessary geometrical and some topological information for the physical body has been provided by the *triangulation* object. The remaining topology, the nodes themselves and the logic of connections between adjacent elements, and the mathematical attributes of elements, namely the basis functions, now need to be assigned.

b) The *setup_system* function

The task of the *setup_system* function is to define the variables for the finite element analysis, such as the assignment of degrees of freedom to the required matrices and all vectors.

The set of Finite Element classes enables us to assign degrees of freedom to associated elemental nodal points. In deal.II notation, those nodes are called *support points* in real space and *unit support points* in the reference space. The *FE_Q* class defines the polynomial basis functions; Code Excerpt 3.3 can be seen to construct an object, *fe*, which refers to the 2d elements with bilinear (i.e. linear in both x and y) basis functions. The *DoFHandler* class connects the dof values to the nodes of the triangulation object; that information is specified in the *dof_handler* object. For simulations involving a large number of elements, calculating the oversized matrix puts a heavy workload on CPU and requires a long run time. Recall that the finite element method divides a continuous body into many elements and calculations are performed at each node within the same local element. Any calculations that include nodes from different elements are therefore not considered, and thus most of the entries in the global system matrix are necessarily zero. This type of matrix is called *sparse*, two examples of which are depicted in Figure 3.7. The global system matrix, $\underline{\underline{A}}$, involves the dot product of two gradients of the basis functions at two nodes, as seen in Equation 2.61, and the size of the system matrix is proportional to the square of the total number of nodes.

```
FE_Q<2> fe(1);
...
DoFHandler<2> dof_handler;

dof_handler(triangulation);
...
DynamicSparsityPattern dsp(dof_handler.n_dofs());

DoFTools::make_sparsity_pattern (dof_handler,dsp);
```

Code Excerpt 3.3

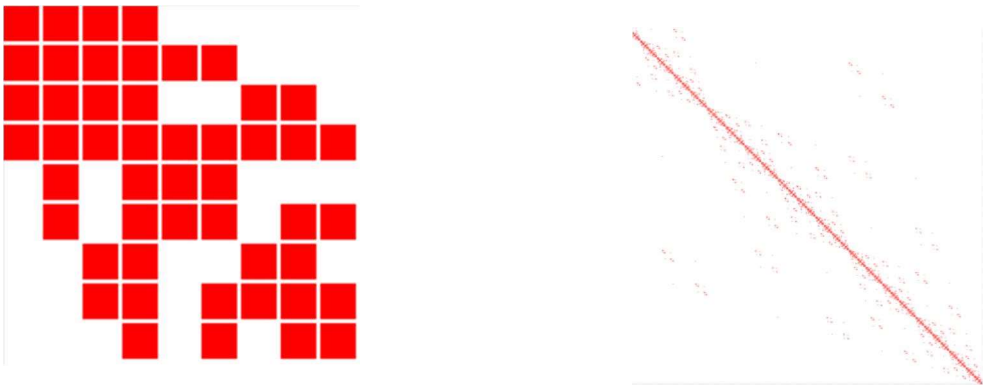

Figure 3.7: The sparse matrices obtained from 2d finite element scalar models using bilinear basis functions for a square body (left) comprising 2x2 elements and 9 dofs, and for a square body (right) comprising 32x32 elements and 1089 dofs.

c) The *assemble_system* function

Up to now, most variables, such as basis function values, have been set in preparation for future computations. In the *assemble_system* function, the components of the system matrix and forcing vector will be evaluated using the elemental form (the elemental system matrix is usually called the *cell_matrix* in deal.II) and then assembled into a global linear system. The boundary conditions are then applied to this system to obtain a unique solution.

As a concrete example, recall Equation 2.62, where the elemental components, A_{ij}^{e} and f_i^e, are calculated from integral equations that normally can not be computed

analytically. Therefore, Gaussian quadrature is employed here for the numerical approach. The *QGauss* class sets up Gaussian quadrature formulae of order *gauss_order* for the numerical integrations. Those required values, for example basis function values, basis function gradients (*shape_grad*), and the quadrature weight-multiplied determinant of the Jacobian (*J*x*W*) etc., are provided by the *FEValues* class at quadrature points of a unit element in the reference space.

The computation of the elemental system component, A_{ij}^{e}, is shown in Code Excerpt 3.4, and its quadrature approximation can be written as:

$$A_{ij}^{e} \approx \sum_{q}^{n_Q} \frac{dN_i(x_q)}{dx} E \frac{dN_j(x_q)}{dx} w_q \quad , \qquad [3.1]$$

where n_Q is the number of quadrature points along the integration direction within the element, $\{w_q\}$ are the quadrature weights for the quadrature point x_q, unrelated to weighting functions previous discussed . We then collect the elemental matrices and vectors to assemble the global ones, i.e. *system_matrix* and *force_vector*. We next include the Neumann boundary conditions to the global forcing vector and apply the Dirichlet boundary conditions to the global system. After all this, we arrive at a final global matrix-vector form which will be solved in the next step.

```
void assemble_system(){
      QGauss<dim> quadrature_formula(gauss_order);
      FEvalues<dim> fe_values(fe,
                              quadrature_formula,
                                    update_shape_grad|…|…);
      …
      for(unsign int q =0; q <n_Q,++q)
             for(int i=0; i<dofs_per_element; ++i)
               for(int j=0; j<dofs_per_element; ++j)
                  cell_matrix(i,j)+=(fe_values.shape_grad(i,q))*
                                    E*
                                    fe_values.shape_grad(j,q))*
                                    fe_values.JxW(q));
      …
      MatrixTools::apply_boundary_values(boundary_values,
                                         system_matrix,
                                         solution,
                                         force_matrix);
      }
```

Code Excerpt 3.4

d) The *solve* function

After receiving the matrix-vector form from the assembly step, the linear algebra system is ready to be solved. For a linear system with a small number of dofs, the solution vector, $\underline{d}$, can be solved by multiplying the inverse of system matrix to the forcing vector

$$\underline{d} = \underline{\underline{A}}^{-1} \underline{f} \qquad [3.2]$$

The deal.II library also has a collection of linear solvers, e.g. Conjugate Gradients method etc., suitable for larger linear systems.

e) The *output* function

After the computation step, the finite element result can be processed and visualized for further analysis. However, the deal.II library does not contain a feature for such postprocessing, so the output function is thus constructed to export the solution, as shown in Code Excerpt 3.5. The *DataOut* class is applied to write the data into a convenient format, such as gnuplot or vtk (Visualization Tookit), which can be accessed by Mathematica and VTK Viewers, respectively.

```
void output(){
      DataOut<dim> data_out;
      …
      data_out.write_gnuplot(outputgpl);
      data_out.write_vtk(outputvtk);
      }
```

Code Excerpt 3.5

3.3 Some FEM examples: solving partial differential equations using deal.II

3.3.1 Example 1: Modelling a uniaxially-loaded, linear-elastic, bar

Continuing from Section 2.2.4, here we generate a finite element model for computing the extension of the effective-1d linear-elastic bar. We discretize the bar into 1d elements and construct the matrix-vector weak form to achieve the finite element solution. The components of the elemental system matrix, A_{ij}^e, and the elemental forcing vector, f_i^e, can be obtained from Equation 2.62, and can be written as

$$A_{ij}^e = \int_{\Omega_e} \frac{d\phi_j}{dx} E \frac{d\phi_i}{dx} dx \qquad ; \qquad f_i^e = \int_{\Omega_e} \phi_i b(x) dx \qquad [3.3]$$

Here i, j label the dofs (which for this scalar problem, also label the nodes). Applying Gaussian quadrature to approximate the integral, Equation 3.3 leads to

$$A_{ij}^e \approx \sum_q^{n_Q} \frac{d\phi_j(x_q)}{dx} E \frac{d\phi_i(x_q)}{dx} w_q \qquad ; \qquad f_i^e \approx \sum_q^{n_Q} \phi_i(x_q) b(x_q) w_q \qquad [3.4]$$

These equations are needed to construct the elemental matrix-vector weak form in deal.II and further used to assemble the global linear system. After the boundary conditions are incorporated into this global linear system, the final linear system can thus be solved. The solutions of the effective-1d bar problem introduced in Section 2.1.3 using a finite element approach are illustrated in Figure 3.8 (a). Here, we adopted finite element models for the effective-1d bar with eight elements, with first-order Lagrange basis functions, and so nine scalar dofs, employing second-order Gaussian quadrature. In contrast, we also adopt for $b_0 = 2\times10^5$ a finite element model with 16 elements, third-

order basis function, and so 49 scalar dofs, employing fourth-order Gaussian quadrature. The subtraction between the analytical solution and finite element solution can be seen in Figure 3.8(b). The finite element solution is fairly accurate as the difference between the two is in order of 10^{-6}. The source code for the more refined calculation is given in Appendix A.1.

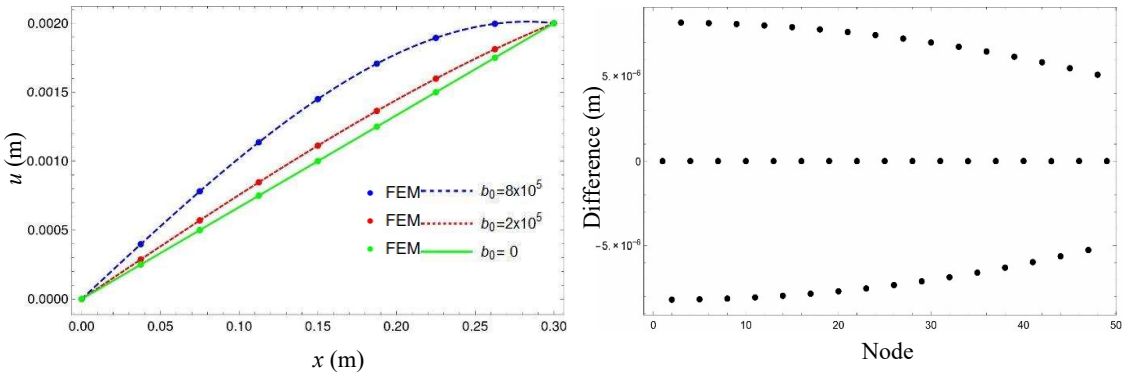

Figure 3.8 (a) Plots of the finite element solution and analytical solution of displacement from the undeformed configuration for a bar experiencing a zero (green), small (red), and large (blue) body force, with parameters as shown. The dots represent the finite element solution, adopting 8 elements, with first-order basis functions, and so 9 scalar dofs. (b) The difference for the case of $b_0 = 2\times10^5$ between the analytical solution and a finite element solution with 16 elements, third-order basis functions and so 49 scalar dofs.

3.3.2 Example 2: Solving Poisson's Equation for a half-ring and a half-spherical shell

In this section, the finite element approach is used to solve a scalar pde in a higher spatial dimension (i.e. in 2d or 3d space). We take Poisson's equation as an example adopted from the *step-3* deal.II tutorial [47], which itself features broadly in physics (e.g. Gauss's law in electrostatics), with a constant *f*,

$$\nabla^2 u(\vec{x}) = f \quad . \tag{3.5}$$

We set boundary conditions that the solution is fixed to be zero on the entire boundary surface:

$$u(\vec{x}) = 0 \quad \text{on } \partial\Omega \quad . \tag{3.6}$$

Multiplying by the weight function and integrating by parts to obtain the weak form, one has

$$\int_{\Omega} \vec{\nabla} w \cdot \vec{\nabla} u \, dV = -\int_{\Omega} w f \, dV \quad . \tag{3.7}$$

Partitioning the body into elements and substituting for the finite element solution in a finite-dimensional function space of basis functions gives

$$\sum_{e} \sum_{i,j}^{n_{\text{nodes},e}} \int_{\Omega_e} \vec{\nabla} \phi_i \cdot \vec{\nabla} \phi_j \, dV \; d_j = -\sum_{e} \sum_{i}^{n_{\text{nodes},e}} \int_{\Omega_e} \phi_i f \, dV \quad , \tag{3.8}$$

leading to components of the elemental form

$$A_{ij}^e = \int_{\Omega_e} \vec{\nabla} \phi_i \cdot \vec{\nabla} \phi_j \, dV \quad ; \quad f_i^e = -\int_{\Omega_e} \phi_i f \, dV \quad . \tag{3.9}$$

As a concrete example we solve Equation 3.5 subject to boundary conditions of Equation 3.6, for $f = -1$, and solve for a half-ring in 2d space, and a half-spherical shell

in 3d space. A finite element model is adopted with first-order Lagrange basis functions and employing first-order Gaussian quadrature. The half-ring is discretized into 240 elements (four radially and 60 azimuthally), and so contains 305 scalar dofs. The half-spherical shell initially adopts four circumferential elements and one top element (by setting the value four within the *half_hyper_shell* function in the source code), this is then refined twice to create a structured mesh of 320 elements and so contains 445 scalar dofs. The refinement, and placing of nodes, is done using the spherical manifold which resulted in the location of nodes being in placed on the required locus and the faces of elements are considered to be curved as in Figure 3.6c. The source code for the half-spherical shell calculation is given in Appendix A.2. We use *ParaView* (*ParaView 5.4.1 64-bit*, *Kitware Inc*) to visualize the finite element solutions, as shown in Figure 3.9. The opacity is set to 0.5 to observe the inside of the half-spherical shell.

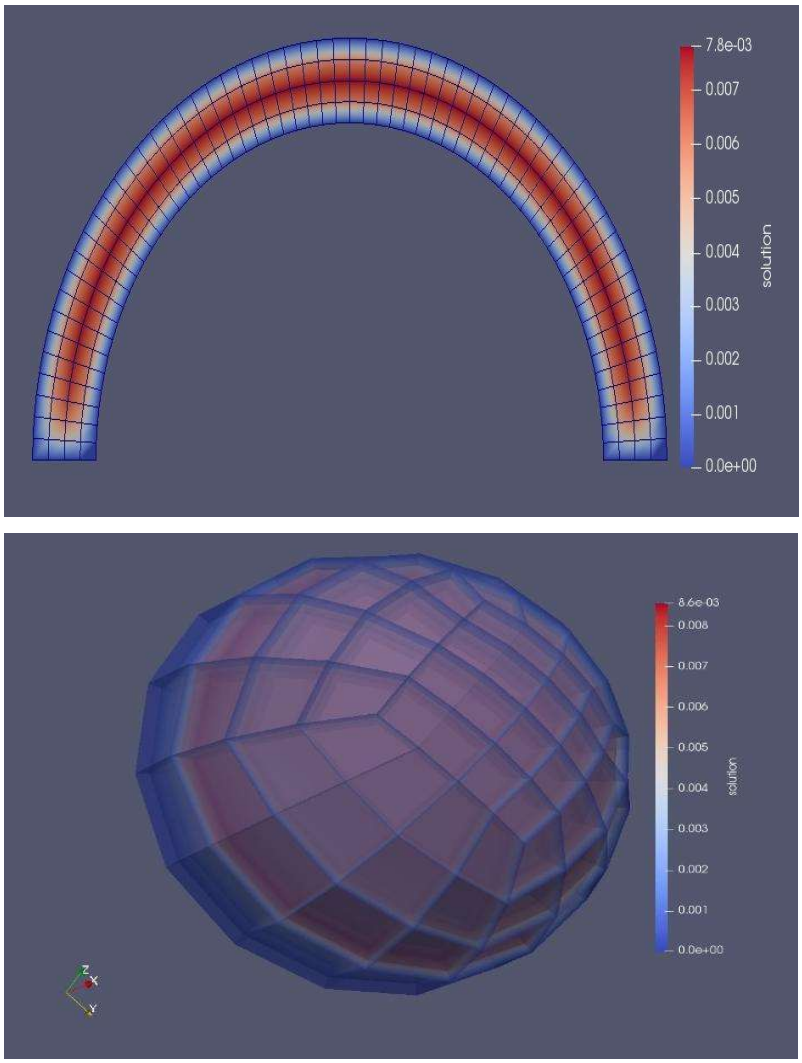

Figure 3.9. A heat map (in colour) of the finite element solutions for a scalar pde (Poisson's equation) for a half-ring (top) and a half-spherical shell (bottom), using first-order basis functions and employing second-order quadrature. For both cases $f = -1$, and the geometry is set to an inner radius, $r_{in} = 1$, and an outer radius $r_{out} = 1.25$. The half-ring is partitioned into 240 elements and the half-spherical shell into 320 elements.

The finite element solution in the centre of the half-ring and centre of the half-spherical shell can be validated with the result expected from analytically solving Poisson's equation with $f = -1$ for a *full*-ring and a *full*-spherical shell, as seen in Figure 3.10.

These analytical solutions for a full-ring, $u(s)$, and a full-spherical shell, $u(r)$, are obtained along the radial axis:

$$u(s) = -\frac{s^2}{4} + A\log(s) + B \quad \text{where} \quad A = \frac{9}{64\log(1.25)} \quad , \quad B = \frac{1}{4} \qquad [3.10]$$

$$u(r) = -\frac{r^2}{6} - \frac{A}{r} + B \qquad \text{where} \quad A = \frac{15}{32} \quad , \quad B = \frac{61}{96} \quad . \qquad [3.11]$$

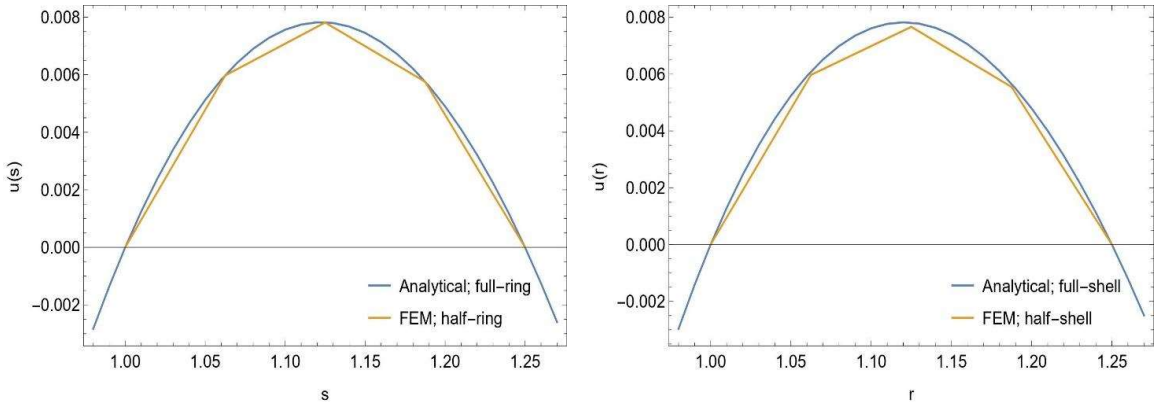

Figure 3.10. Plots of the finite element solution (light orange) in the centre of the half-ring (left) and centre of the half-spherical shell (right) with first-order basis functions employed; and the analytical solution (blue) for a full-ring (left) along the radial axis $\hat{s}$, and a full-spherical shell (right) along the radial axis $\hat{r}$.

We have demonstrated the generation of a cornea-like mesh with which we can now investigate its mechanical behavior. Scaling from a 2d body to a 3d body with the same corresponding symmetry can be seen to be a fairly trivial step. In the rest of the thesis, we shall focus on the 3d elastic deformations which involve solving a *vector* pde in 3d space.

3.3.3 Example 3: Modelling a loaded, linear-elastic, cube

In this section, we apply the finite element method to model the deformation of a three-dimensional, linear-elastic body subject to fixed tractions and fixed displacements of specified boundary surfaces. In addition, the following assumptions are made: the cube is an isotropic, homogeneous material; all body forces are neglected; and the problem to be solved is a quasi-static situation.

Consider a cube with sides of length 1 dm, as shown in Figure 3.11. The cube is loaded by applying an external traction to the top surface, $x_3 = 1$, of $\vec{t}\left(+\hat{e}_3, \vec{x}\right) = x_1 \times 10^9 \hat{e}_3$ Pa, and the displacement of the bottom surface, $x_3 = 0$, is fixed to be zero. All other boundary surfaces experience zero external tractions and are free to move, and the body force is neglected. The cube is assumed a Young's modulus $E = 150$ GPa which is similar to that of the steel.

For a *vector* problem, the boundary conditions have to be separately specified for each of the spatial dimensions. It is convenient to consider each physical boundary surface, $\partial\Omega'$, (e.g. each face of the cube) to be labelled as if it were several surfaces depending upon whether the boundary conditions are a fixed *displacement* along the *i*-th spatial direction (then labeled as $\partial\Omega'_{ui}$), or a fixed *traction* along the *i*-th spatial direction (then labeled as $\partial\Omega'_{ti}$). One requirement is that respective Dirichlet and Neumann boundaries must be disjoint; mathematically this can be written as $\partial\Omega_{ui} \bigcap \partial\Omega_{ti} = \varnothing, \quad i = 1, 2, 3$. When applied to a mesh, defined by nodes, this manifests as *one* node which belongs to

both boundaries, $\partial\Omega_{ui}$ and $\partial\Omega_{ti}$, satisfying a Dirichlet boundary and, with respect to any adjacent nodes purely within $\partial\Omega_{ti}$, a Neumann boundary.

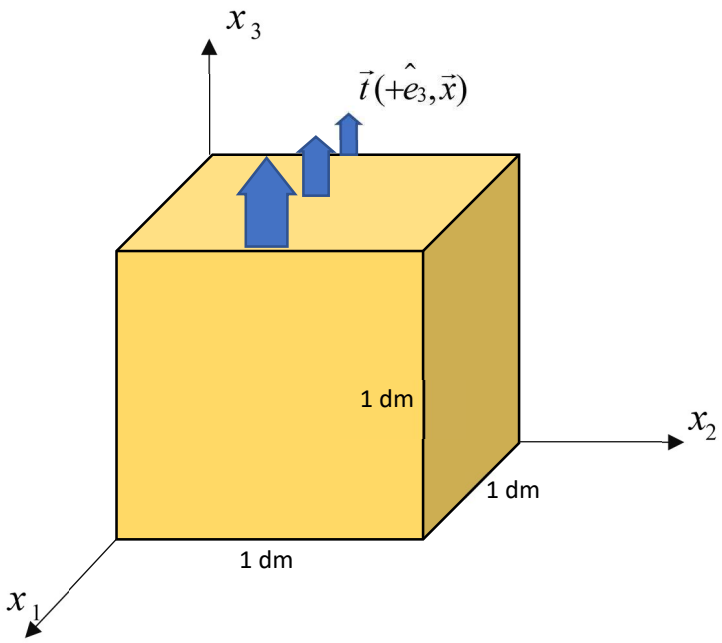

Figure 3.11. Depiction of a $1\,\mathrm{dm}\times 1\,\mathrm{dm}\times 1\,\mathrm{dm}$ cube subjected to a traction on the top surface with other boundary conditions as discussed in the text.

We thus set the boundary conditions to be

$$\text{the bottom face:}\quad \vec{u}(\vec{x})=\vec{0}\quad,\quad \vec{x}\in\partial\Omega_{ui}\quad i=\{1,2,3\} \qquad [3.12]$$

$$\text{the top face:}\quad \vec{t}\left(+\hat{e}_3,\vec{x}\right)=\underline{\underline{\sigma}}\hat{e}_3=x_1\times 10^9\hat{e}_3\,\mathrm{Pa}\quad,\quad \vec{x}\in\partial\Omega_{t3} \qquad [3.13]$$

$$\text{all other faces:}\quad \vec{t}\left(\hat{n},\vec{x}\right)=\vec{0}\quad,\quad \vec{x}\in\partial\Omega_{ti}\quad,\quad \hat{n}=\left\{\pm\hat{e}_1,\pm\hat{e}_2\right\}\quad. \qquad [3.14]$$

As described in the previous section, it is necessary to establish a math model for the physical system which consists of balance principles, the constitutive relation, kinematic relations, and boundary conditions.

We first apply the balance principle of linear momentum to analyse the forces in equilibrium. Recall that the body force is ignored, and we consider a quasi-static case, so Equation 2.21 becomes

$$\vec{\nabla}\bullet\underline{\underline{\sigma}}=\vec{0} \qquad ; \qquad \sum_{j}^{3}\frac{\partial\sigma_{ij}}{\partial x_j}=0 \quad i=\{1,2,3\} \tag{3.15}$$

Next, we derive the weak form by multiplying with a dot product a vector weight function and integrate by parts, as seen in Section 2.2.1, and arrive at:

$$\int_{\partial\Omega}\vec{w}\bullet(\underline{\underline{\sigma}}\vec{n})\,dA-\int_{\Omega}\vec{\nabla}\vec{w}:\underline{\underline{\sigma}}\,dV=0 \quad ; \quad \int_{\partial\Omega}\sum_{i}^{3}w_i\sigma_{ij}n_j dA-\int_{\Omega}\sum_{i,j}^{3}\frac{\partial w_i}{\partial x_j}\sigma_{ij}dV=0 \tag{3.16}$$

where w_i is the *i*-th component of the weight vector function, n_j is the *j*-th component of the surface normal vector, and the surface and volume integrals are taken over the entire body. In order to obtain a unique solution, we now include the boundary conditions on the boundary surfaces. In general, these are (here, the superscript *g* denotes a 'given' value):

$$\sigma_{ij}n_j=t_i^g \text{ on } \{\partial\Omega_{ti}\} \quad , \quad u_i=u_i^g \text{ on } \{\partial\Omega_{ui}\} \quad , \tag{3.17a,b}$$

When substituted these surfaces into Equation 3.16, we conclude

$$\int_{\Omega}\sum_{i,j=1}^{3}\frac{\partial w_i}{\partial x_j}\sigma_{ij}\,dV=\sum_{i=1}^{3}\left(\int_{\partial\Omega_{ti}}w_i t_i^g dA+\int_{\partial\Omega_{ui}}w_i\sigma_{ij}n_j dA\right) \tag{3.18}$$

The Dirichlet and Neumann boundary conditions must be disjoint even if they relate to the same physical boundary surface, hence, here we took the summation out from the surface integral term to accomplish the boundary integral with respect to each dimension.

Similar to section 2.2.1, the weight function at any Dirichlet boundary must be zero, $w_i = 0$ on $\{\partial\Omega_{ui}\}$, and so the integration over the $\partial\Omega_{ui}$ term will vanish.

Substituting the constitutive relation for a linear-elastic material, Equation 2.22, into Equation 3.18 gives

$$\int_{\Omega} \sum_{i,j}^{3} \frac{\partial w_i}{\partial x_j} \sum_{k,l}^{3} \mathbb{C}_{ijkl} \varepsilon_{kl} \, dV = \sum_{i=1}^{3} \int_{\delta\Omega_{ti}} w_i \, t_i^g \, dA \qquad [3.19]$$

Recall the stiffness tensor for an isotropic material can be reduced to two independent variables and the component form of the stiffness tensor can be obtained from Equation 2.24,

$$\mathbb{C}_{ijkl} = \lambda \delta_{ij} \delta_{kl} + \mu \left(\delta_{ik} \delta_{jl} + \delta_{il} \delta_{jk} \right) \qquad [3.20]$$

The Cauchy strain tensor which dictates the strain-displacement relation under small deformation, Equation 2.18, is defined as:

$$\varepsilon_{kl} = \frac{1}{2} \left(\frac{\partial u_k}{\partial x_l} + \frac{\partial u_l}{\partial x_k} \right) \quad . \qquad [3.21]$$

The stiffness tensor has a minor symmetry, $\mathbb{C}_{ijkl} = \mathbb{C}_{ijlk}$, and so due to the summation over *k* and *l* in Equation 3.19, the stress tensor written in component form (incorporating implicit summation over related indices) becomes

$$\sigma_{ij} = \mathbb{C}_{ijkl} \varepsilon_{kl} = \mathbb{C}_{ijkl} \frac{\partial u_k}{\partial x_l} \quad , \qquad [3.22]$$

and so, we can rewrite Equation 3.19 in terms of the displacement,

$$\int_{\Omega} \sum_{i,j}^{3} \frac{\partial w_i}{\partial x_j} \sum_{k,l}^{3} \mathbb{C}_{ijkl} \frac{\partial u_k}{\partial x_l} dV = \sum_{i=1}^{3} \int_{\delta\Omega_{ti}} w_i \, t_i^g \, dA \quad . \qquad [3.23]$$

Finally, we discretize the physical body into a set of finite elements and represent the finite element solution $\vec{u}^{\,h}$ in a finite-dimensional space of basis functions, which can be written as,

$$\vec{u}(\vec{x}) \approx \vec{u}^{\,h}(\vec{x}) = \sum_{n}^{n_{\text{nodes}}} \phi_n(\vec{x}) \vec{d}_n \quad ; \quad u_i^h = \sum_{n}^{n_{\text{nodes}}} \phi_n d_n^i \quad i = \{1,2,3\}, \qquad [3.24]$$

where φ_n is the basis function corresponding to the *n*-th node using global numbering, and $\vec{d}_n$ is the *vector* dof at the *n*-th node and is composed of three scalar dofs, $\{d_n^i\}$.

$$\int_{\Omega} \sum_{i,j,k,l=1}^{3} \sum_{n,m}^{n_{\text{nodes},e}} \frac{\partial \phi_n}{\partial x_j} \mathbb{C}_{ijkl} \frac{\partial \phi_m}{\partial x_l} d_m^l \, dV = \sum_{i=1}^{3} \int_{\delta\Omega_{ti}} \sum_{n}^{n_{\text{nodes},e}} \phi_n \, t_i^g \, dA \quad . \qquad [3.25]$$

Therefore, a non-boundary element can take the elemental system matrix component,

$$A_{ik}^{e,nm} = \int_{\Omega_e} \sum_{j,l=1}^{3} \frac{\partial \phi_n}{\partial x_j} \mathbb{C}_{ijkl} \frac{\partial \phi_m}{\partial x_l} dV \qquad [3.26]$$

Since the body force is neglected here, there are only Neumann tractions providing external forces within the global forcing vector. In finite element simulations, we can usually consider boundary elements to take the same elemental calculations as non-boundary ones, and then, once the global linear system is obtained from the assembly of all elements, the boundary conditions are then imposed to arrive at the final global matrix-vector weak form. The resulting linear system can then be solved using linear algebra tools.

We adopt here a finite element approach for the cube with a $10\times10\times10$ mesh and employing first-order basis functions, and so we have in total 1,331 nodes and 3,993 scalar dofs. The source code for this linear-elastic cube calculation is given in Appendix A.3. The finite element solution is shown in Figure 3.12, the displacement vector is scaled 150 times larger in the figure. As the Neumann traction increases along $\hat{e}_1$ and the base of the cube is held fixed, the body is dragged towards the $-\hat{e}_1$ direction. It is interesting to notice that, at one-third height from the bottom, $x_3 \approx 0.03\text{m}$, as shown in Figure 3.12(b), the body is squeezed in a small x_1 region and demonstrates an interesting deformation curve.

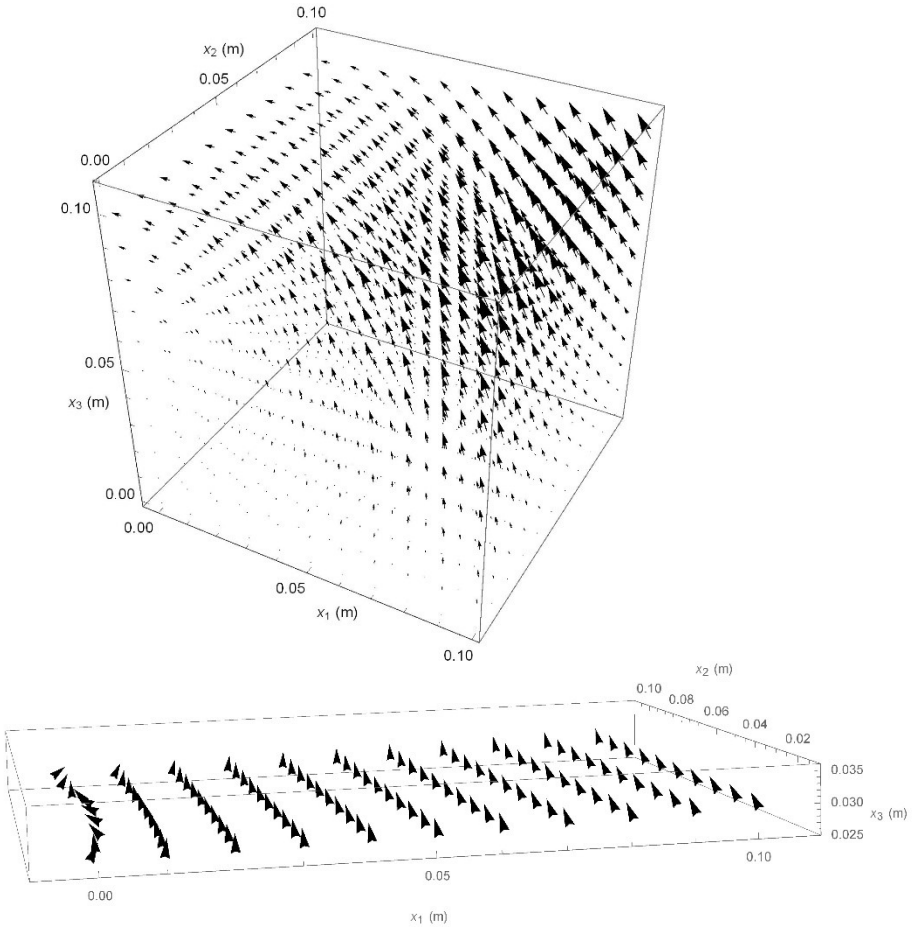

Figure 3.12. (a) Plot of the finite element solution displacement vectors for the cube with a $10\times10\times10$ mesh and using first-order basis functions. The finite element model is constructed using Hex8 elements and contains 3993 scalar dofs in total. (b) the displacement vector plot at $x_3 = 0.03\,\mathrm{m}$. The displacement vector is scaled up by 150.

This finite element model for a vector pde in 3d can now be further extended to systems with a geometry more appropriate for the cornea. By employing a cornea-like mesh, it enables us to investigate corneal linear-elastic deformation from a finite element approach. Alternatively, by involving step-by-step increments, a dynamic simulation could be achieved.

Chapter 4

Preliminary results from finite element modelling of a simulated cornea

In this chapter we introduce a linear-elastic model to simulate corneal deformation. For this, the model from Section 3.3.3, is applied together with the half-spherical shell mesh introduced in Section 3.3.2. The deformation of various isotropic, nearly-incompressible, and linear-elastic simulated corneas subject to fixed displacements and fixed tractions is performed. The body force is neglected, and the final finite element linear system takes the same form as seen in Section 3.3.3,

$$\underline{\underline{A}}\,\underline{d} = \underline{f} \qquad [4.1]$$

where the *n,m*-th component of the *e*-th element system matrix component (relating nodes *n* and *m*) associated with the *i*- and *k-th* directions reads,

$$A_{ik}^{e,nm} = \int_{\Omega_e} \sum_{j,l=1}^{3} \frac{\partial \phi_n}{\partial x_j} \mathbb{C}_{ijkl} \frac{\partial \phi_m}{\partial x_l} dV \qquad [4.2]$$

Here, we set up the material property for the cornea based on the literature reviewed in Chapter 1. First, the cornea is assumed to be homogeneous, hence the material parameters are uniformly assigned throughout the body with the modelled cornea made up from a single material layer. Poisson's ratio, ν, is assigned a value of 0.49999 to represent a nearly-incompressible material. The isotropic, linear-elastic property chosen for this

cornea model, i.e. Young's modulus, is set to $E = 2$ MPa [8]. For the geometry used in this model, a half-spherical shell with an inner radius, $r_{\text{in}} = 0.01$ m, and an outer radius, $r_{\text{out}} = 0.012\,\text{m}$ is initially adopted to approximately represent (an admittedly thick) cornea.

The base of the cornea is considered to be fixed in place (the Dirichlet boundary condition), and it experiences a loading on its inner surface corresponding to normal IOP (~2 kPa); see Figure 4.1. For simplicity, we first adopt a fixed traction purely along $\hat{e}_1$, and any body force is neglected. The outer surface experiences zero traction, and both inner and outer curved surfaces are free to move. These lead to the boundary conditions,

$$\text{on the base:} \quad \vec{u}(\vec{x}) = \vec{0} \qquad \vec{x} \in \partial\Omega_{\text{base}} \qquad [4.3]$$

$$\text{inner surface:} \quad \vec{t}\left(\hat{n}, \vec{x}\right) = (2000, 0, 0) \ \text{Pa} \qquad \vec{x} \in \partial\Omega_{inner} \qquad [4.4]$$

$$\text{outer surface:} \quad \vec{t}\left(\hat{n}, \vec{x}\right) = \vec{0} \qquad \vec{x} \in \partial\Omega_{outer} \qquad [4.5]$$

where $\hat{n}$ here is the outward normal of the relevant surface.

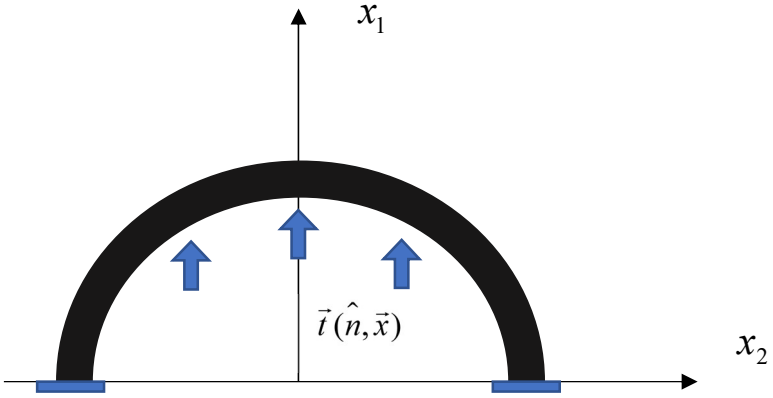

Figure 4.1. Depiction of the cross section of the simulated cornea subjected to a traction on the inner surface and a fixed displacement on the base as discussed in the text.

The corneal mesh initially adopts four circumferential hexahedral elements and one top hexahedral element (by setting the value of four within the *half_hyper_shell* function in deal.II). This is then refined three times to create a structured mesh of $2^3 = 8$ radial elements, and $5 \times 4^3 = 320$ azimuthal/polar elements. In total this is 2,560 elements, and we employ first-order basis functions, so have in total 3,033 nodes and 9,099 scalar dofs. The refined mesh is obtained with a spherical manifold (by invoking the *set_manifold* function) which locates nodes on the appropriate curved locus. The source code for this linear-elastic model calculation is given in Appendix A.4. The finite element solution is shown in Figure 4.2, where for clarity the displacement vectors in Figure 4.2(d) are scaled up by a factor of 100.

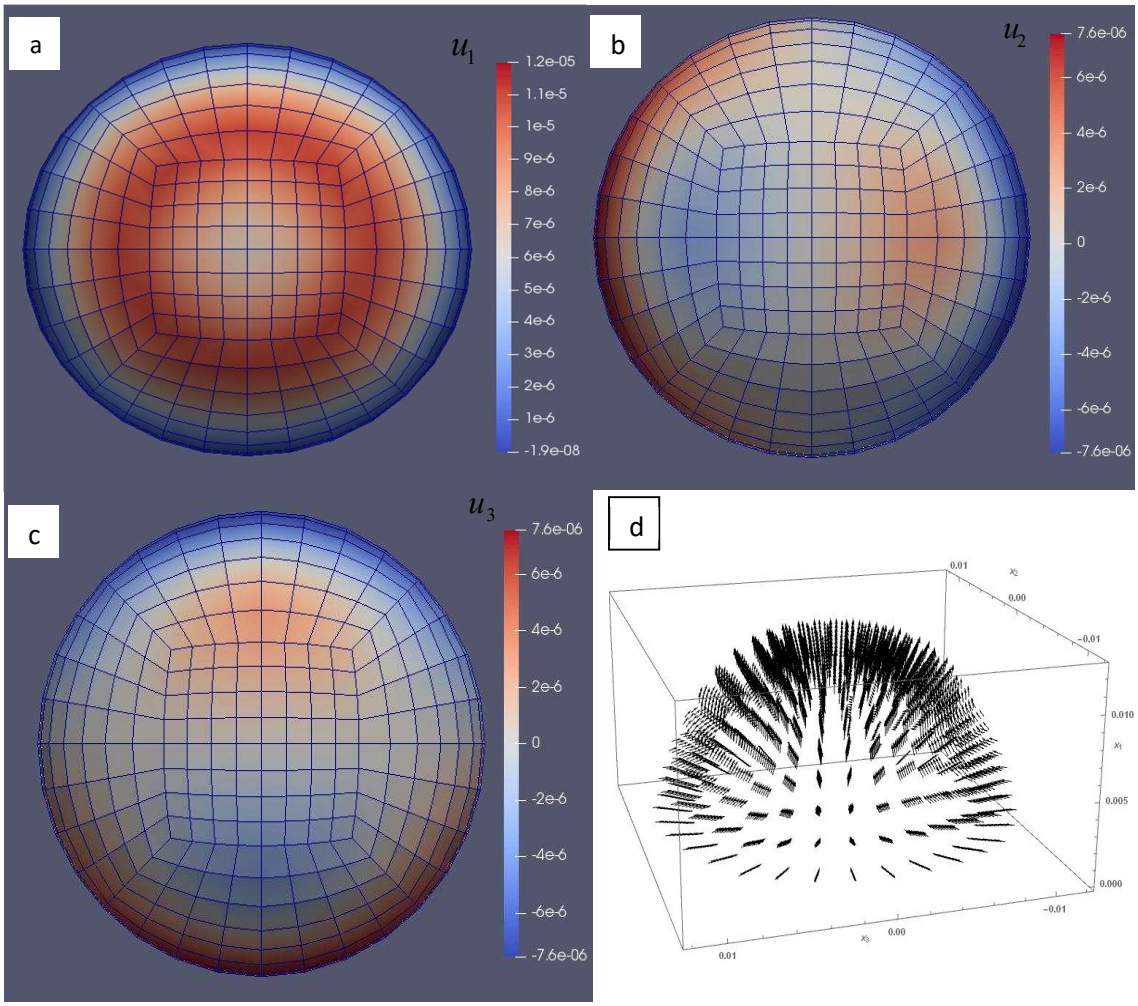

Figure 4.2 (in color). Finite element model of a linear-elastic, half-spherical shell with 1 cm inner radius, with Poisson's ratio of 0.49999, elastic modulus of 2 MPa and subject to a 2 kPa traction on all inner surfaces along $\hat{e}_1$. This adopted 2,560 elements and first-order basis functions, resulting in a total of 9,099 scalar degrees of freedom. The finite element solutions are presented in the heat plots (a-c) of the displacement components u_1, u_2 and u_3 in meters, respectively. In all heat plots, $\hat{e}_2$ and $\hat{e}_3$ are in the plane of the paper, horizontally and vertically, respectively. (d) Displacement vectors, given in meters, scaled up a factor of 100.

This 2 mm thick model cornea has its maximum displacement along $\hat{e}_1$, u_1, of 0.011982 mm on the inner surface at approximately $38°$ away from the apex. The apex region is deformed less along $\hat{e}_1$ compared to the off-axis regions, hence, the top region of the cornea is 'flattened'. This flattening can be seen in Figure 4.3(a) and (c). The

analyze if such relative deformation may provide some insights into the occurrence and stability of eye diseases of biomechanical origin, such as keratoconus.

The corneal collagen lamellae are known to be highly interwoven in the anterior surface [48], thus we now adopt a model with inhomogeneous biomechanical properties, where the cornea becomes stiffer as the stroma lamellae become more interwoven close to the anterior surface. Here, Young's modulus, E, is set to linearly increase in the radial direction, and takes the form $E(r) = \frac{r}{r_{in}} \times E_0$, where $E_0 = 2$ kPa is the Young's modulus at the inner surface with radius r_{in}. The displacement along $\hat{e}_1$, u_1, along a slice of the cornea located at $x_2 = 0$, is shown in Figure 4.3. The replacement code for simulating the inhomogeneous cornea is given in Appendix A.5.

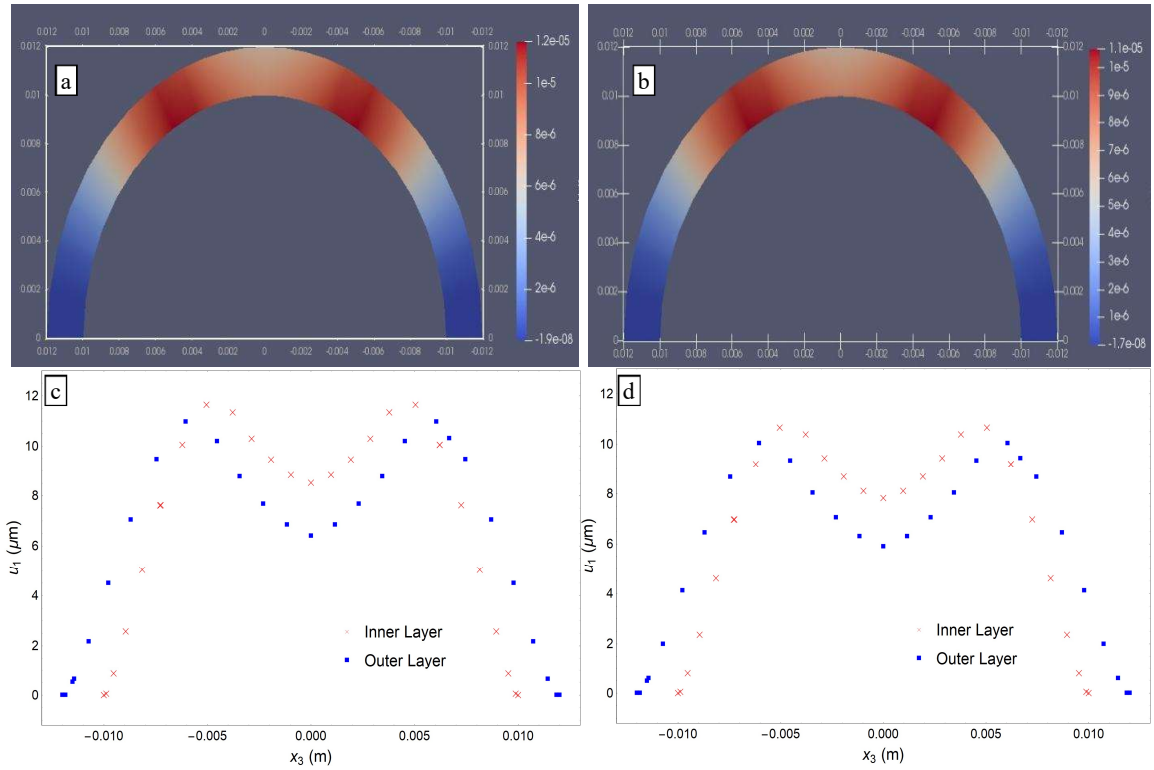

Figure 4.3. Deformation of a linear-elastic half-spherical shell subject to a traction of 2 kPa on the inner surface along $\hat{e}_1$ with (left) a homogeneous Young's modulus, $E_0 = 2$ kPa , and (right) $E(r) = \frac{r}{r_{in}} \times E_0$. The solutions are presented in the heat plots (in color) of the displacement along $\hat{e}_1$, u_1, for the (a) homogeneous and (b) inhomogeneous model. (c) and (d): the displacement of the inner layer and outer layer for each case.

For the inhomogeneous cornea with the stiffness linearly increasing along the radial direction, the u_1 displacement has a similar pattern to the homogeneous one, but with less displacement due to the larger Young's modulus towards the anterior surface.

In this chapter, we have adopted a linear-elastic model of a cornea using deal.II to predict the deformation under uniaxial loading. This can be further modified for testing the effect of pressure differences to cornea deformation. This represents the first step toward analyzing more realistic models of the cornea: true hydrostatic loading mimicking

various IOPs acting perpendicularly to all surfaces, a polar angle from $0°$ to $\sim 53°$; and a more complicated material structure that includes additional inhomogeneity, hyperelastic behaviour, and also, eventually, anisotropic properties. These considerations are expected to be conducted in the near future.

Chapter 5

Conclusion

In this study we have introduced and given concrete examples of a computational model using the finite element method which provides an effective way to simulate the deformation of different bodies, with various material properties, under a variety of conditions. The finite element method adopted here involves determining the set of differential equations that, together with boundary conditions, uniquely describe the deformation of a body under load. These set of equations result in a single governing pde, that together with the boundary conditions, solves the physical system using finite element analysis. We simulated the extension of an effective-1d, linear-elastic bar, where the analytical solution is perfectly solvable, and consistent with the finite element solution. Our finite element modelling is expanded to a higher dimension to solve Poisson's equation for a cornea-like body, here, the analytical solution can be approximated and validates the finite element solution. Involving to the complexity of a 3d vector system, we simulated the deformation of a loaded, linear-elastic, cube where an analytical approach is generally impossible. Preliminary results for the linear-elastic deformation of a simulated cornea have also been achieved using the finite element approach in which differential deformation was observed. Understanding such differential deformations in the future may give valuable perspective into eye diseases of biomechanical origin.

Bibliography

[1] A. Gefen, R. Shalom, D. Elad, and Y. Mandel, "Biomechanical analysis of the keratoconic cornea," *J. Mech. Behav. Biomed. Mater*, **2**(3), 224-236 (2009).

[2] B.O. Hedbeys, and S. Mishima, "The thickness-hydration relationship of the cornea," *Exp. Eye. Res*, **5**(2), 221-228 (1966).

[3] F. Rüfer, A. Schröder , and C. Erb, "White-to-white corneal diameter: Normal values in healthy humans obtained with the orbscan II topography system," *Cornea*, **24**(3), 259–261 (2005).

[4] P.J. Shih, C.J. Huang, T.H. Huang, H.C. Lin, J.Y. Yen, I.J. Wang, H.J. Cao, W.P. Shih, and C.A. Dai, "Estimation of the corneal Young's modulus in vivo based on a fluid-filled spherical-shell model with Scheimpflug imaging," *J. Ophthal*, **2017**, 5410143/1-11 (2017).

[5] M. Singh, Z. Han, J. Li, S. Vantipalli, S.R. Aglyamov, M.D. Twa, and K.V. Larin, "Quantifying the effects of hydration on corneal stiffness with noncontact optical coherence elastography," *J. Cataract. Refract. Surg*, **44**(8), 1023-1031 (2018).

[6] D.A Hoeltzel, P. Altman, K. Buzard, and K. Cheo, "Strip extensiometry for comparison of the mechanical response of bovine, rabbit and human cornea," *J. Biomech. Eng,* **114**(2), 202-215 (1992).

[7] Y. Zeng, J. Yang, K. Huang, Z. Lee, and X. Lee, "A comparison of biomechanical properties between human and porcine cornea" *J. biomech*, **34**, 533-537 (2001).

[8] A.C. Jayasuriya, S. Ghosh, J.I. Scheinbeim, V. Lubkin, G. Bennett, and P. Kramer, "A study of piezoelectric and mechanical anisotropies of the human cornea," *Biosensors and Bioelectronics*, **18**, 381-387 (2003).

[9] S.J. Antony, "Imaging shear stress distribution and evaluating the stress concentration factor of the human eye," Sci. R*ep*, **5**, 8899 (2015).

[10] B.I. Akca , E.W. Chang, S. Kling, A. Ramier, G. Scarcelli, S. Marcos, and S.H. Yun, "Observation of sound-induced corneal vibrational modes by optical coherence tomography," *Biomedical Optics Express*, **6**(9), 3313-3319 (2015).

[11] A. Elasheikh, D. Alhasso, and P. Rama, "Biomechanical properties of human and porcine cornea," *Experimental Eye Research*, **86**, 783-790 (2008).

[12] M. Lanza, M. Cennamo, S. Iaccarino, V. Romano, M. Bifani, C. Irregolare, and A. Lanza, "Evaluation of corneal deformation analyzed with a Scheimpflug based device," *Cont. Lens. Anterior. Eye,* **38**(2), 89–93 (2015).

[13] A. Montanino, M. Angelillo, and A. Pandolfi, "Modelling with a meshfree approach the cornea-aqueous humor interaction during the air puff test," *J. Mech. Behav. Biomed. Mater*, **77**, 205-216 (2018).

[14] S. Kling, N. Bekesi, C. Dorronsoro, D. Pacual, and S, Marcos, "Corneal viscoelastic properties from finite-element analysis of *in vivo* air-puff deformation," *PLOS ONE*, **9**(8), e104904 (2014).

[15] A. Elasheikh, and K. Anderson, "Comparative study of corneal strip extensometry and inflation tests," *J.R. Soc. Interface*, **2**,177-185 (2005).

[16] S.L-Y. Woo, A.S. Kobayashi, W.A. Schlegel, and C. Lawrence, "Nonlinear material properties of intact cornea and sclera," *Exp. Eye. Res*. **14**,29-39 (1972).

[17] A. Elasheikh, B. Geraghty, P. Rama, M. Campanell, and K.M. Meek, "Characterization of age-related variation in corneal biomechanical properties," *J.R. Soc. Interface*, **7**,1475-1485 (2010).

[18] S. Feizi, M.R. Jafarinasab, F. Karimian, H. Hasanpour, and A. Masudi, "Central and peripheral corneal thickness measurement in normal and keratoconic eyes using three corneal pachymeters," *J. Ophthalmic. Vis. Res*, **9**, 296–304 (2014).

[19] K.M. Meek. *Collagen*. Edited by P. Fratzl. (Springer, Boston, MA, 2008) p.359-396

[20] M.S. Sridhar, "Anatomy of cornea and ocular surface," *Indian. J. Ophthalmol*, **66**(2), 190-194 (2018).

[21] K.M. Meek, and C. Knupp, "Corneal structure and transparency," *Prog. Retin. Eye. Res*, **49**,1-16 (2015).

[22] D.F. Holmes, and K.E. Kadler, "The precision of lateral size control in the assembly of corneal collagen fibrils," *J. Mol. Biol,* **345**, 773-784 (2005).

[23] R.M. Aspden, and D.W.L. Hukins, "Determination of the direction of preferred orientation and the orientation distribution function of collagen fibrils in connective tissues from high-angle X-ray diffraction patterns," *J. Appl. Cryst,* **12,** 306-311 (1979).

[24] C. Boote, D. Dennis, Y. Huang, A.J. Quantock, and K.M. Meek, "Lamellar orientation in human cornea in relation to mechanical properties," *J. Struct. Biol,* **149**(1), 1-6 (2005).

[25] K.M. Meek, and C. Boote, "The use of X-ray scattering techniques to quantify the orientation and distribution of collagen in the corneal stroma," *Prog. Retin. Eye. Res*, **28**, 369-392 (2009).

[26] D.E. Freund, R.L. McCally, and R.A. Farrell, "Effects of fibril orientations on light scattering in the cornea," *J. Opt. Soc. Am. A*, **3**(11), 1970-1982 (1986)

[27] E.A. Boettner, and R.J. Wolter, "Transmission of the ocular media," *Invest. Ophthalmol. Vis. Sci*, **1**, 776-783 (1962).

[28] D.A. Luce, "Determining in vivo biomechanical properties of the cornea with an ocular response analyzer," *J. Cataract. Refract. Surg*, **31**(1), 156–162 (2005).

[29] H. Erdogan, Z. Akingol, O. Cam, and S. Sencan, "A comparison of NCT, Goldman application tonometry values with and without fluorescein," *Clini. Ophthalmol*, **12,** 2183-2188 (2018).

[30] N. Terai, F. Raiskup, M. Haustein, L.E. Pillunat, and E. Spoer, "Identification of biomechanical properties of the cornea: the ocular response analyzer," *Current. Eye. Res*, **37**(7),553–562 (2012).

[31] M. Sullivan-Mee, S.C. Billingsley, and A.D. Patel, "Ocular response analyzer in subjects with and without glaucoma," *Optom. Vis. Sci,* **85**, 463–470 (2008).

[32] I. Simonini, M. Angelillo, and A. Pandolfi, "Theoretical and numerical analysis of the corneal air puff test," *J. Mech. Phys. Solids*, **93**, 118-134 (2016).

[33] G.A. Holzapfel, *Nonlinear Solid Mechanics: A Continuum Approach for Engineering*, 1st ed. (Wiley, Toronto, Canada, 2000).

[34] P. Kelly, *Solid Mechanics Lecture Notes - An Introduction to Solid Mechanics,* (The University of Auckland, Auckland, New Zealand 2013). Retrieved on April 24th 2020.

[35] J. Cheng, and L.T Zhang, "A general approach to derive stress and elasticity tensors for hyperelastic isotropic and anisotropic biomaterials," *Int. J. Comput. Methods*, **15**(1), 1850028 (2018).

[36] R. Hooke, *Lectures of spring explaining the power of springing bodies,* (London, 1678).

[37] B. Kim, S.B. Lee, J. Lee, S. Cho, H. Park, S. Yeom, and S.H. Park, "A comparison among Neo-Hookean model, Mooney-Rivlin model, and Ogden model for chloroprene rubber," *Int. J. Pre. Engine Manu*, **13**, 759-764 (2012).

[38] A.G. Guy, *Introduction to Materials Science*, (McGraw-Hill, New York, United States, 1972).

[39] K. Garikipati, *Lecture notes: Introduction to finite element methods*, (University of Michigan, Ann Arbor, United States, 2014).

[40] S. Mazumder, *Numerical methods for partial differential equations: Finite difference and finite volume methods,* 1st ed, (Academic Press, San Diego, United States, 2016), Ch.1.

[41] J. Blazek, Computational Fluid Dynamics: *Principles of solution of the governing equations,* 2rd ed, (Elsevier, United Kingdom, 2005) p.29-72.

[42] W. Bangerth, R. Hartmann, and G. Kanschat, "deal.II—a general purpose object oriented finite element library," TOMS, **33**(4), 24/1-27 (2007).

[43] J. Fish, and T. Belytschko, *A first course in finite elements*, 1st ed, (Wiley, Chippenham Wiltshire, 2007).

[44] E. Zheng, F. Ma, and D. Zhang, "A least-squares non-polynomial finite element method for solving the polygonal-line grating problem," *J. Math. Analy. Appli*, **397**(2), 550-560 (2013).

[45] I. Girotto, *High-Performance Computing at the ICTP: Challenges of Large Scale Scientific Simulations and Programs for Education*, ICTP, Retrieved from

http://indico.ictp.it/event/8680/session/29/contribution/109/material/slides/0.pdf on May 11th 2020.

[46] W. Bangerth, "Using modern features of C++ for adaptive finite element methods: Dimension-independent programming in deal.II." In Proceedings of the 16th IMACS World. (2000).

[47] W. Bangerth, R. Hartmann, and G. Kanschat. deal.II Differential equations analysis library, Technical reference. *http://www.dealii.org* .

[48] S.J. Petsche, D. Chernyak, J. Martiz, M.E. Levenston, and P.M. Pinsky, "Depth-dependent transverse shear properties of the human corneal stroma," *Invest. Ophthalmol. Vis. Sci,* **53**(2),873-880 (2012).

Appendix A

Source code for FEM using deal.II

Appendix A.1: Source code for Example 1.

```
1.  // Appendix A1 for example 1
2.  // grid
3.  #include <deal.II/grid/tria.h>
4.  #include <deal.II/grid/tria_accessor.h>
5.  #include <deal.II/grid/tria_iterator.h>
6.  #include <deal.II/grid/grid_generator.h>
7.  #include <deal.II/grid/grid_out.h>
8.  //
9.  #include <deal.II/fe/fe_q.h>
10. #include <deal.II/fe/fe_system.h>
11. #include <deal.II/fe/fe_values.h>
12. #include <deal.II/dofs/dof_handler.h>
13. #include <deal.II/dofs/dof_tools.h>
14. #include <deal.II/dofs/dof_accessor.h>
15.
16. #include <deal.II/base/quadrature_lib.h>
17. #include <deal.II/base/function.h>
18. #include <deal.II/base/logstream.h>
19. #include <deal.II/base/function.h>
20. #include <deal.II/numerics/vector_tools.h>
21. #include <deal.II/numerics/matrix_tools.h>
22. // linear algebra
23. #include <deal.II/lac/vector.h>
24. #include <deal.II/lac/full_matrix.h>
25. #include <deal.II/lac/sparse_matrix.h>
26. #include <deal.II/lac/dynamic_sparsity_pattern.h>
27. #include <deal.II/lac/sparsity_pattern.h>
28. #include <deal.II/lac/sparse_direct.h>
29. #include <deal.II/lac/solver_cg.h>
30. #include <deal.II/lac/precondition.h>
31. #include <deal.II/numerics/data_out.h>
32. #include <deal.II/lac/matrix_out.h>
33. //Standard library
34. #include <fstream>
35. #include <iostream>
36.
37. using namespace dealii;
38.
39. template <int dim>
40. class BarElastic
41. {
42. public:
43.   BarElastic ();
44.
```

```
45.   void run ();
46.
47.  private:
48.   void meshing ();
49.   void setup_system ();
50.   void assemble_system ();
51.   void solve ();
52.   void output_results () const;
53.   void boundary_conditions();
54.
55.
56.   Triangulation<dim>     triangulation;
57.   FE_Q<dim>              fe;
58.   DoFHandler<dim>        dof_handler;
59.
60.   SparsityPattern sparsity_pattern;
61.   SparseMatrix<double> system_matrix;
62.
63.   Vector<double>       solution;
64.   Vector<double>       system_rhs;
65.
66.   std::map<types::global_dof_index, double> boundary_values;
67. };
68.
69.
70. template <int dim>
71. void BarElastic<dim>::boundary_conditions()
72. {
73.    MappingQ1<dim, dim> mapping;
74.    std::vector<Point<dim,double>> dof_position(dof_handler.n_dofs());
75.
76.    DoFTools::map_dofs_to_support_points(mapping,
77.                                         dof_handler,
78.                                         dof_position);
79.    boundary_values[0] = 0;
80.
81.    for (unsigned int i = 0; i < dof_handler.n_dofs(); i++)
82.     {
83.         if (dof_position[i][0] ==0.3) { boundary_values[i] = 0.002;}
84.     }
85. }
86.
87.
88. template <int dim>
89. BarElastic<dim>::BarElastic ()
90.   :
91.   fe (3),
92.   dof_handler (triangulation)
93. {}
94.
95. template <int dim>
96. void BarElastic<dim>::meshing ()
97. {
98.   GridGenerator::hyper_cube (triangulation, 0, 0.3);
99.   triangulation.refine_global (4);
100.
101.         std::cout << "Number of active cells: "
102.                   << triangulation.n_active_cells()
103.                   << std::endl;
104.       }
105.
```

```
106.      template <int dim>
107.      void BarElastic<dim>::setup_system ()
108.      {
109.        dof_handler.distribute_dofs (fe);
110.        std::cout << "Number of degrees of freedom: "
111.                  << dof_handler.n_dofs()
112.                  << std::endl;
113.
114.        DynamicSparsityPattern dsp(dof_handler.n_dofs());
115.        DoFTools::make_sparsity_pattern (dof_handler, dsp);
116.        sparsity_pattern.copy_from(dsp);
117.
118.
119.        system_matrix.reinit (sparsity_pattern);
120.
121.        solution.reinit (dof_handler.n_dofs());
122.        system_rhs.reinit (dof_handler.n_dofs());
123.      }
124.
125.      template <int dim>
126.      void BarElastic<dim>::assemble_system()
127.      {
128.        QGauss<dim> quadrature_formula(4);
129.        FEValues<dim> fe_values(fe,
130.                                quadrature_formula,
131.                                update_values | update_gradients |
132.                                update_JxW_values | update_quadrature_points);
133.        const unsigned int dofs_per_cell = fe.dofs_per_cell;
134.        const unsigned int n_q_points    = quadrature_formula.size();
135.        FullMatrix<double> cell_matrix(dofs_per_cell, dofs_per_cell);
136.        Vector<double>     cell_rhs(dofs_per_cell);
137.        std::vector<types::global_dof_index> local_dof_indices(dofs_per_cell);

138.        const unsigned int bodyforce = 2e5;
139.        const unsigned int youngs = 3e6;
140.        for (const auto &cell: dof_handler.active_cell_iterators())
141.          {
142.
143.            fe_values.reinit (cell);
144.            cell_matrix = 0;
145.            cell_rhs = 0;
146.
147.            for (unsigned int q_index=0; q_index<n_q_points; ++q_index)
148.                for (unsigned int i=0; i<dofs_per_cell; ++i)
149.                  for (unsigned int j=0; j<dofs_per_cell; ++j)
150.                    cell_matrix(i,j) += (fe_values.shape_grad (i, q_index) *
151.                                         fe_values.shape_grad (j, q_index) *
152.                                         youngs *
153.                                         fe_values.JxW (q_index));
154.
155.              for (unsigned int q_index=0; q_index<n_q_points; ++q_index)
156.                for (unsigned int i=0; i<dofs_per_cell; ++i)
157.                  cell_rhs(i) += (fe_values.shape_value (i, q_index) *
158.                                  fe_values.quadrature_point(q_index)[0] *
159.                                  bodyforce *
160.                                  fe_values.JxW (q_index));
161.
162.
163.            cell->get_dof_indices (local_dof_indices);
164.
165.
```

```
166.             for (unsigned int i=0; i<dofs_per_cell; ++i)
167.               for (unsigned int j=0; j<dofs_per_cell; ++j)
168.                 system_matrix.add (local_dof_indices[i],
169.                                    local_dof_indices[j],
170.                                    cell_matrix(i,j));
171.
172.
173.             for (unsigned int i=0; i<dofs_per_cell; ++i)
174.               system_rhs(local_dof_indices[i]) += cell_rhs(i);
175.           }
176.
177.
178.         boundary_conditions();
179.
180.         MatrixTools::apply_boundary_values (boundary_values,
181.                                             system_matrix,
182.                                             solution,
183.                                             system_rhs,
184.                                             false);
185.       }
186.
187.       template <int dim>
188.       void BarElastic<dim>::solve ()
189.       {
190.          SparseDirectUMFPACK A;
191.          A.initialize(system_matrix);
192.          A.vmult(solution, system_rhs);
193.
194.       }
195.
196.
197.       template <int dim>
198.       void BarElastic<dim>::output_results () const
199.       {
200.
201.         DataOut<dim> data_out;
202.
203.         data_out.attach_dof_handler (dof_handler);
204.         data_out.add_data_vector (solution, "solution");
205.         data_out.build_patches ();
206.
207.         std::ofstream output ("solution.gpl");
208.         data_out.write_gnuplot (output);
209.         std::ofstream outputFile ("output.txt");
210.         outputFile <<"" ;
211.         auto first = true;
212.         for (float f : solution)
213.           {
214.           if (!first) { outputFile << ","; }
215.           first = false;
216.           outputFile << f;
217.           }
218.       outputFile <<  ""<< std::endl;
219.
220.       }
221.
222.       template <int dim>
223.       void BarElastic<dim>::run ()
224.       {
225.         meshing ();
226.         setup_system ();
```

```
227.         assemble_system ();
228.         solve ();
229.         output_results ();
230.       }
231.
232.
233.       int main ()
234.       {
235.         BarElastic<1> barelastic;
236.         barelastic.run ();
237.         return 0;
238.       }
```

Appendix A.2: Source code for Example 2.

```
1.  // Appendix A.2 Example 2,  Poisson's Eqn in 2d and 3d for cornea-like shape.
2.  // grid
3.  #include <deal.II/grid/tria.h>
4.  #include <deal.II/grid/tria_accessor.h>
5.  #include <deal.II/grid/tria_iterator.h>
6.  #include <deal.II/grid/grid_generator.h>
7.  #include <deal.II/grid/grid_out.h>
8.  #include <deal.II/grid/manifold_lib.h>
9.
10. //
11. #include <deal.II/fe/fe_q.h>
12. #include <deal.II/dofs/dof_handler.h>
13. #include <deal.II/dofs/dof_tools.h>
14. #include <deal.II/fe/fe_values.h>
15. #include <deal.II/base/quadrature_lib.h>
16. #include <deal.II/base/function.h>
17. #include <deal.II/numerics/vector_tools.h>
18. #include <deal.II/numerics/matrix_tools.h>
19.
20. // linear algebra
21. #include <deal.II/lac/vector.h>
22. #include <deal.II/lac/full_matrix.h>
23. #include <deal.II/lac/sparse_matrix.h>
24. #include <deal.II/lac/dynamic_sparsity_pattern.h>
25. #include <deal.II/numerics/data_out.h>
26. #include <deal.II/lac/sparse_direct.h>
27.
28. // standard library
29. #include <fstream>
30. #include <iostream>
31.
32.
33. using namespace dealii;
34.
35. template<int dim>
36. class Example2
37. {
38. public:
```

```
39.     Example2();
40.     void run();
41.   private:
42.     void generate_grid(unsigned int numberOFrefinement);
43.     void setup_system();
44.     void assemble_system();
45.     void solver();
46.     void output();
47.
48.     Triangulation<dim> triangulation;
49.     FE_Q<dim> fe;
50.     DoFHandler<dim> dof_handler;
51.
52.     SparsityPattern sparsity_pattern;
53.     SparseMatrix<double> system_matrix;
54.
55.     Vector<double> solution;
56.     Vector<double> system_rhs;
57.     };
58.
59.
60.     template<int dim>
61.     Example2<dim>::Example2()
62.       :
63.       fe(1),
64.       dof_handler (triangulation)
65.       {}
66.
67.
68.    template <int dim>
69.    void Example2<dim>::generate_grid(unsigned int numberOFrefinement)
70.    {
71.      double shell_radius_in = 1.0;
72.      double shell_radius_out = 1.25;
73.      const Point<dim> shell_center(0, 0, 0);
74.           GridGenerator::half_hyper_shell(triangulation,
75.                                shell_center,
76.                                shell_radius_in,
77.                                shell_radius_out
78.                                   );
79.
80.      const SphericalManifold<dim> boundary_description(shell_center);
81.      triangulation.set_manifold (0, boundary_description);
82.      triangulation.refine_global (numberOFrefinement);
83.
84.      std::ofstream outputfile("Half Shell");
85.      GridOut gridout;
86.      gridout.write_eps(triangulation, outputfile);
87.      std::cout << "Grid written to Half Shell";
88.    }
89.
90.  template <int dim>
91.  void Example2<dim>::setup_system ()
92. {
93.   dof_handler.distribute_dofs (fe);
94.   std::cout << "Number of degrees of freedom: "
95.             << dof_handler.n_dofs()
96.             << std::endl;
97.
98.   std::cout << "Total number of element:"
99.              << triangulation.n_active_cells()
```

```
100.                    << std::endl;
101.
102.        DynamicSparsityPattern dsp(dof_handler.n_dofs());
103.        DoFTools::make_sparsity_pattern (dof_handler, dsp);
104.        sparsity_pattern.copy_from(dsp);
105.        system_matrix.reinit (sparsity_pattern);
106.        solution.reinit (dof_handler.n_dofs());
107.        system_rhs.reinit (dof_handler.n_dofs());
108.      }
109.
110.
111.
112.        template <int dim>
113.        void Example2<dim>::assemble_system()
114.      {
115.        QGauss<dim> quadrature_formula(2);
116.        FEValues<dim> fe_values(fe,
117.                                quadrature_formula,
118.                                update_values | update_gradients |
119.                                update_JxW_values);
120.        const unsigned int dofs_per_cell = fe.dofs_per_cell;
121.        const unsigned int n_q_points    = quadrature_formula.size();
122.        FullMatrix<double> cell_matrix(dofs_per_cell, dofs_per_cell);
123.        Vector<double>     cell_rhs(dofs_per_cell);
124.        std::vector<types::global_dof_index> local_dof_indices(dofs_per_cell);

125.        for (const auto &cell: dof_handler.active_cell_iterators())
126.          {
127.
128.            fe_values.reinit (cell);
129.            cell_matrix = 0;
130.            cell_rhs = 0;
131.
132.            for (unsigned int q_index=0; q_index<n_q_points; ++q_index)
133.              {
134.
135.                for (unsigned int i=0; i<dofs_per_cell; ++i)
136.                  for (unsigned int j=0; j<dofs_per_cell; ++j)
137.                    cell_matrix(i,j) += (fe_values.shape_grad (i, q_index) *
138.                                         fe_values.shape_grad (j, q_index) *
139.                                         fe_values.JxW (q_index));
140.
141.                for (unsigned int i=0; i<dofs_per_cell; ++i)
142.                  cell_rhs(i) += (fe_values.shape_value (i, q_index) *
143.                                  fe_values.JxW (q_index));
144.              }
145.
146.            cell->get_dof_indices (local_dof_indices);
147.
148.
149.            for (unsigned int i=0; i<dofs_per_cell; ++i)
150.              for (unsigned int j=0; j<dofs_per_cell; ++j)
151.                system_matrix.add (local_dof_indices[i],
152.                                   local_dof_indices[j],
153.                                   cell_matrix(i,j));
154.
155.
156.            for (unsigned int i=0; i<dofs_per_cell; ++i)
157.              system_rhs(local_dof_indices[i]) += cell_rhs(i);
158.          }
159.
```

```
160.
161.          std::map<types::global_dof_index,double> boundary_values;
162.          VectorTools::interpolate_boundary_values (dof_handler,
163.                                                    0,
164.                                                    Functions::ZeroFunction<dim>(
    ),
165.                                                    boundary_values);
166.
167.          MatrixTools::apply_boundary_values (boundary_values,
168.                                              system_matrix,
169.                                              solution,
170.                                              system_rhs);
171.        }
172.
173.
174.        template <int dim>
175.        void Example2<dim>::solver()
176.        {
177.            SparseDirectUMFPACK A;
178.            A.initialize(system_matrix);
179.            A.vmult(solution, system_rhs);
180.        }
181.
182.        template <int dim>
183.        void Example2<dim>::output()
184.        {
185.          DataOut<dim> data_out;
186.          data_out.attach_dof_handler(dof_handler);
187.          data_out.add_data_vector(solution, "solution");
188.          data_out.build_patches();
189.          std::ofstream output( "Half Shell.vtk");
190.          data_out.write_vtk(output);
191.          std::cout << "Solision written to Half Shell.vtk";
192.          std::ofstream outputgpl ("solution.gpl");
193.          data_out.write_gnuplot (outputgpl);
194.
195.        }
196.
197.            template <int dim>
198.            void Example2<dim>::run()
199.            {
200.              generate_grid(2);
201.              setup_system ();
202.              assemble_system();
203.              solver();
204.              output();
205.            }
206.
207.        int main ()
208.        {
209.          Example2<3> example2;
210.          example2.run();
211.          return 0;
212.        }
```

Appendix A.3: Source code for Example 3.

```
1.  // Appendix A3
2.  // example 3: linear cube
3.  // grid
4.  #include <deal.II/grid/tria.h>
5.  #include <deal.II/grid/tria_accessor.h>
6.  #include <deal.II/grid/tria_iterator.h>
7.  #include <deal.II/grid/grid_generator.h>
8.  #include <deal.II/grid/grid_out.h>
9.  #include <deal.II/grid/manifold_lib.h>
10.
11. //
12. #include <deal.II/fe/fe_q.h>
13. #include <deal.II/fe/fe_system.h>
14. #include <deal.II/fe/fe_values.h>
15. #include <deal.II/dofs/dof_handler.h>
16. #include <deal.II/dofs/dof_tools.h>
17. #include <deal.II/dofs/dof_accessor.h>
18.
19. #include <deal.II/base/quadrature_lib.h>
20. #include <deal.II/base/function.h>
21. #include <deal.II/base/logstream.h>
22. #include <deal.II/numerics/vector_tools.h>
23. #include <deal.II/numerics/matrix_tools.h>
24.
25. // linear algebra
26. #include <deal.II/lac/vector.h>
27. #include <deal.II/lac/full_matrix.h>
28. #include <deal.II/lac/sparse_matrix.h>
29. #include <deal.II/lac/dynamic_sparsity_pattern.h>
30. #include <deal.II/lac/sparsity_pattern.h>
31. #include <deal.II/lac/sparse_direct.h>
32. #include <deal.II/numerics/data_out.h>
33. #include <deal.II/lac/matrix_out.h>
34.
35. // standard library
36. #include <fstream>
37. #include <iostream>
38.
39.
40. using namespace dealii;
41.
42. template <int dim>
43. class CubeVector
44. {
45.  public:
46.     CubeVector();
47.     ~CubeVector();
48.     void run();
49.
50.
51.  private:
52.     void meshing();
53.     void setup_system();
54.     void assemble();
55.     void solve();
56.     void output();
```

```
57.     double coefficient(const Point<dim> &p_coeff,  unsigned int i,  unsigned int
    j, unsigned int k,  unsigned int l);
58.     void boundary_conditions();
59.
60.     Triangulation<dim> triangulation;
61.     FESystem<dim>      fesystem;
62.     DoFHandler<dim> dof_handler;
63.
64.     SparsityPattern sparsity_pattern;
65.     SparseMatrix<double> system_matrix;
66.
67.     Vector<double> solution;
68.     Vector<double> system_rhs;
69.
70.     std::vector<std::string> solution_name;
71.     std::map<types::global_dof_index, double> boundary_values;
72.     };
73.
74. template <int dim>
75. double CubeVector<dim>::coefficient(const Point<dim> &p_coeff,  unsigned int i,
     unsigned int j, unsigned int k,  unsigned int l)/*function coefficient(Point, i
    , j, k, l)*/
76. {
77.     double E = 2e11;
78.     double nu = 0.3;
79.     double lambda = (E*nu)/((1.+nu)*(1.-2.*nu));
80.     double mu = E/(2.*(1.+nu));
81.     double C = lambda*(i==j)*(k==l) + mu*((i==k)*(j==l) + (i==l)*(j==k));
82.     return C;
83. }
84.
85.
86.
87. template <int dim>
88. void CubeVector<dim>::boundary_conditions()
89. {
90.    MappingQ1<dim, dim> mapping;
91.    std::vector<Point<dim,double>> dof_position(dof_handler.n_dofs());
92.
93.    DoFTools::map_dofs_to_support_points(mapping,
94.                                         dof_handler,
95.                                         dof_position);
96.
97.    for (unsigned int i = 0; i < dof_handler.n_dofs(); i++)
98.     {
99.         if (dof_position[i][2] ==0) { boundary_values[i] = 0;}
100.           }
101.       }
102.
103.       template <int dim>
104.       CubeVector<dim>::CubeVector()
105.             :
106.             fesystem (FE_Q<dim>(1), dim),
107.             dof_handler (triangulation)
108.             {}
109.
110.
111.       template <int dim>
112.       CubeVector<dim>::~CubeVector()
113.       {
114.           dof_handler.clear();
```

```
115.       }
116.
117.
118.       template <int dim>
119.       void CubeVector<dim>::meshing()
120.       {
121.           const Point<dim> LL(0, 0, 0);
122.           const Point<dim> RH(0.1,0.1, 0.1);
123.           std::vector<unsigned int> repetition(dim); //element along each axis,
      eg mesh 3x3x1
124.              repetition[0] =10;
125.              repetition[1] =10;
126.              repetition[2] =10;
127.           GridGenerator::subdivided_hyper_rectangle(triangulation,
128.                                                     repetition,
129.                                                     LL,
130.                                                     RH);
131.           std::string mesh_name = "Rectangle-mesh";
132.           std::ofstream outputmeshfile(mesh_name);
133.           GridOut gridout;
134.           gridout.write_eps(triangulation, outputmeshfile);
135.           std::cout << " Mesh file output to "
136.                     << mesh_name
137.                     << std::endl;
138.
139.       }
140.
141.
142.       template <int dim>
143.       void CubeVector<dim>::setup_system ()
144.       {
145.          const unsigned int total_dofs = dof_handler.n_dofs();
146.          dof_handler.distribute_dofs (fesystem);
147.          std::cout << "Total number of scaler degrees of freedom: "
148.                    << dof_handler.n_dofs()
149.                    << std::endl;
150.
151.
152.          std::cout << "Total number of element:"
153.                     << triangulation.n_active_cells()
154.                     << std::endl;
155.
156.          sparsity_pattern.reinit(dof_handler.n_dofs(), dof_handler.n_dofs(),dof
    _handler.max_couplings_between_dofs());
157.          DoFTools::make_sparsity_pattern (dof_handler, sparsity_pattern);
158.          sparsity_pattern.compress();
159.          system_matrix.reinit (sparsity_pattern);
160.          solution.reinit (dof_handler.n_dofs());
161.          system_rhs.reinit (dof_handler.n_dofs());
162.       }
163.
164.       template <int dim>
165.       void CubeVector<dim>::assemble()
166.       {
167.          QGauss<dim> quadrature_formula(2);
168.          QGauss<2>   face_quadrature_formula(2);
169.
170.          FEValues<dim> fe_values(fesystem, quadrature_formula,
171.                                  update_values |
172.                                  update_gradients |
173.                                  update_JxW_values |
```

```
174.                                     update_quadrature_points);
175.
176.          FEFaceValues<dim> fe_face_values (fesystem, face_quadrature_formula,
177.                                            update_values |
178.                                            update_gradients |
179.                                            update_JxW_values |
180.                                            update_quadrature_points);
181.
182.          const unsigned int dofs_per_cell = fesystem.dofs_per_cell;
183.          const unsigned int n_q_points    = quadrature_formula.size();
184.          const unsigned int nodes_per_cell =dofs_per_cell/dim;
185.
186.          std::cout << "Number of nodes per element: "
187.                    << nodes_per_cell
188.                    << std::endl;
189.
190.          FullMatrix<double>   cell_matrix (dofs_per_cell, dofs_per_cell);
191.          Vector<double>       cell_rhs (dofs_per_cell);
192.
193.          std::vector<unsigned int> local_dof_indices (dofs_per_cell);
194.
195.          typename DoFHandler<dim>::active_cell_iterator cell = dof_handler.begi
    n_active(),
196.                                                         endc = dof_handler.end
    ();
197.         for (; cell!=endc; ++cell)                              // loop over
    element
198.           {
199.             cell_matrix = 0;
200.             cell_rhs = 0;
201.
202.             fe_values.reinit (cell);                            // associated
     elemental variables
203.
204.          for (unsigned int q_index = 0; q_index<n_q_points; ++q_index)
205.           for (unsigned int A = 0; A < nodes_per_cell; A++)
206.            for (unsigned int i = 0; i < dim; i++)
207.             for (unsigned int B = 0; B<nodes_per_cell; B++)
208.              for (unsigned int k = 0; k<dim; k++)
209.               for (unsigned int j = 0; j<dim; j++)
210.                for (unsigned int l = 0; l<dim; l++)
211.                    {
212.                       double stiffness = coefficient(fe_values.quadrature_point
     (q_index), i, j, k, l);
213.
214.                       cell_matrix(dim*A+i,dim*B+k) += (stiffness *
215.                                                fe_values.shape_grad(dim*A+i,q_index
    )[j]*
216.                                                fe_values.shape_grad(dim*B+k,q_index
    )[l]*
217.                                                fe_values.JxW(q_index));
218.                    }
219.
220.              Vector<double> traction(dim);
221.              traction = 0;
222.             for (unsigned int f = 0;f< GeometryInfo<dim>::faces_per_cell;++f)
223.             {
224.                 if (cell->face(f)->center()[2] == 0.1)
225.                 {
226.                     fe_face_values.reinit(cell, f);
```

```
227.                     for (unsigned int face_q = 0; face_q<face_quadrature_formul
    a.size(); ++face_q)
228.                     {
229.                     double x = fe_face_values.quadrature_point(face_q)[0];
230.                     traction[0] = 0;
231.                     traction[1] = 0;
232.                     traction[2] = 1e9*x;
233.
234.
235.                         for (unsigned int A=0;A<nodes_per_cell;A++)
236.                               for (unsigned int i=0; i<dim;i++)
237.                                   {
238.                                    cell_rhs[dim*A+i] +=traction[i]*
239.                                    fe_face_values.shape_value(dim*A+i, face_q)*
240.                                     fe_face_values.JxW(face_q);
241.                                   }
242.                       }
243.                 }
244.             }
245.             cell->get_dof_indices (local_dof_indices);
246.             for (unsigned int i=0; i<dofs_per_cell; ++i)
247.               {
248.                   system_rhs(local_dof_indices[i]) += cell_rhs(i);
249.               for (unsigned int j=0; j<dofs_per_cell; ++j)
250.                   {
251.                   system_matrix.add (local_dof_indices[i],
252.                                      local_dof_indices[j],
253.                                      cell_matrix(i,j));
254.                    }
255.               }
256.           }
257.
258.       std::ofstream outputFile ("output_f.txt");
259.           outputFile <<"" ;
260.           auto first = true;
261.           for (float f : system_rhs)
262.         {
263.           if (!first) { outputFile << " "; }
264.           first = false;
265.           outputFile << f;
266.         }
267.          outputFile << "" << std::endl;
268.
269.
270.           boundary_conditions();
271.
272.           MatrixTools::apply_boundary_values (boundary_values,
273.                                               system_matrix,
274.                                               solution,
275.                                               system_rhs,
276.                                               false);
277.       }
278.
279.        template <int dim>
280.        void CubeVector<dim>::solve()
281.        {
282.            SparseDirectUMFPACK A;
283.            A.initialize(system_matrix);
284.            A.vmult(solution, system_rhs);
285.        }
286.
```

```
287.        template <int dim>
288.        void CubeVector<dim>::output()
289.        {
290.            solution_name.emplace_back("u_x");
291.            solution_name.emplace_back("u_y");
292.            solution_name.emplace_back("u_z");
293.
294.            DataOut<dim> data_out;
295.            data_out.attach_dof_handler(dof_handler);
296.
297.            data_out.add_data_vector(solution, solution_name);
298.            data_out.build_patches();
299.            std::ofstream outputgpl ("solution.gpl");
300.            data_out.write_gnuplot (outputgpl);
301.        }
302.
303.        template <int dim>
304.        void CubeVector<dim>::run()
305.        {
306.            meshing();
307.            setup_system();
308.            assemble();
309.            solve();
310.            output();
311.        }
312.
313.        int main()
314.        {
315.
316.            CubeVector<3> cubevector;
317.            cubevector.run();
318.            return 0;
319.        }
```

Appendix A.4: Source code for a simulated cornea

```
1.  // Appendix A.4
2.  //Modelling a time-
    independent, half spherical shell, under small deformations when experiencing a
    pressure difference between the interior and exterior surfaces
3.  // grid
4.  #include <deal.II/grid/tria.h>
5.  #include <deal.II/grid/tria_accessor.h>
6.  #include <deal.II/grid/tria_iterator.h>
7.  #include <deal.II/grid/grid_generator.h>
8.  #include <deal.II/grid/grid_out.h>
9.  #include <deal.II/grid/manifold_lib.h>
10.
11. //
12. #include <deal.II/fe/fe_q.h>
13. #include <deal.II/fe/fe_system.h>
14. #include <deal.II/fe/fe_values.h>
15. #include <deal.II/dofs/dof_handler.h>
```

```
16. #include <deal.II/dofs/dof_tools.h>
17. #include <deal.II/dofs/dof_accessor.h>
18.
19. #include <deal.II/base/quadrature_lib.h>
20. #include <deal.II/base/function.h>
21. #include <deal.II/base/logstream.h>
22. #include <deal.II/base/function.h>
23. #include <deal.II/numerics/vector_tools.h>
24. #include <deal.II/numerics/matrix_tools.h>
25.
26. // linear algebra
27. #include <deal.II/lac/vector.h>
28. #include <deal.II/lac/full_matrix.h>
29. #include <deal.II/lac/sparse_matrix.h>
30. #include <deal.II/lac/dynamic_sparsity_pattern.h>
31. #include <deal.II/lac/sparsity_pattern.h>
32. #include <deal.II/lac/sparse_direct.h>
33. #include <deal.II/numerics/data_out.h>
34. // C++ standard library
35. #include <fstream>
36. #include <iostream>
37. #include <math.h>
38.
39. using namespace dealii;
40.
41. template <int dim>
42. class CorneaElastic
43. {
44.  public:
45.     CorneaElastic();
46.     ~CorneaElastic();
47.     void run();
48.
49.  private:
50.     void meshing();
51.     void setup_system();
52.     void assemble();
53.     void solve();
54.     void output();
55.     double coefficient(const Point<dim> &p_coeff,  unsigned int i,  unsigned int
     j, unsigned int k,  unsigned int l);
56.     void boundary_conditions();
57.
58.     Triangulation<dim> triangulation;
59.     FESystem<dim>      fesystem;
60.     DoFHandler<dim> dof_handler;
61.
62.     SparsityPattern sparsity_pattern;
63.     SparseMatrix<double> system_matrix;
64.
65.     Vector<double> solution;
66.     Vector<double> system_rhs;
67.
68.     std::vector<std::string> solution_name;
69.     std::map<types::global_dof_index, double> boundary_values;
70.     };
71.
72. // Isotropic linear elastic model
73. template <int dim>
```

```
74. double CorneaElastic<dim>::coefficient(const Point<dim> &p_coeff,  unsigned int
    i,  unsigned int j, unsigned int k,  unsigned int l)/*function coefficient(Point
    , i, j, k, l)*/
75. {
76.
77.     const Point<dim> shell_center(0, 0, 0);
78.
79.
80.     double E = 2e6;
81.     double nu = 0.49999;                                        // nearly incompress
    ible
82.     double lambda = (E*nu)/((1.+nu)*(1.-
    2.*nu));            // Lame constant lambda
83.     double mu = E/(2.*(1.+nu));                                 // Lame constant mu

84.     double C = lambda*(i==j)*(k==l) + mu*((i==k)*(j==l) + (i==l)*(j==k));
85.     //stiffness tensor C_ijkl
86.     return C;
87. }
88.
89. template <int dim>
90. void CorneaElastic<dim>::boundary_conditions()
91. {
92.    MappingQ1<dim, dim> mapping;
93.    std::vector<Point<dim,double>> dof_position(dof_handler.n_dofs());
94.
95.    DoFTools::map_dofs_to_support_points(mapping,
96.                                         dof_handler,
97.                                         dof_position);
98.
99.    for (unsigned int i = 0; i < dof_handler.n_dofs(); i++)
100.           {
101.               if (dof_position[i][0] ==0) { boundary_values[i] = 0;}
102.           }
103.       }
104.
105.
106.       template <int dim>
107.       CorneaElastic<dim>::CorneaElastic()
108.             :
109.             fesystem (FE_Q<dim>(1), dim),
110.             dof_handler (triangulation)
111.             {}
112.
113.
114.       template <int dim>
115.       CorneaElastic<dim>::~CorneaElastic()
116.       {
117.           dof_handler.clear();
118.       }
119.
120.
121.       template <int dim>
122.       void CorneaElastic<dim>::meshing()
123.       {
124.           double shell_radius_in = 0.01;
125.           double shell_radius_out = 0.012;
126.           const Point<dim> shell_center(0, 0, 0);
127.           const SphericalManifold<dim> surface_description(shell_center);
128.           GridGenerator::half_hyper_shell(triangulation,
129.                                           shell_center,
```

```
                                        shell_radius_in,
                                        shell_radius_out,
                                        4   );
            triangulation.reset_all_manifolds();
            triangulation.set_all_manifold_ids(0);
            triangulation.set_manifold(0, surface_description);
            triangulation.refine_global (3);

             std::ofstream outputfile("Spherical manifolded mesh for cornea");
             GridOut gridout;
             gridout.write_eps(triangulation, outputfile);
             std::cout << "Grid written to Spherical manifolded mesh for cornea"

                       << std::endl;

        }

        template <int dim>
        void CorneaElastic<dim>::setup_system ()
        {
           const unsigned int total_dofs = dof_handler.n_dofs();
           dof_handler.distribute_dofs (fesystem);
           std::cout << "Total number of scaler degrees of freedom: "
                     << dof_handler.n_dofs()
                     << std::endl;


           std::cout << "Total number of element:"
                      << triangulation.n_active_cells()
                      << std::endl;

           sparsity_pattern.reinit(dof_handler.n_dofs(), dof_handler.n_dofs(),dof
    _handler.max_couplings_between_dofs());
           DoFTools::make_sparsity_pattern (dof_handler, sparsity_pattern);
           sparsity_pattern.compress();
           system_matrix.reinit (sparsity_pattern);
           solution.reinit (dof_handler.n_dofs());
           system_rhs.reinit (dof_handler.n_dofs());
           }

        template <int dim>
        void CorneaElastic<dim>::assemble()
        {
           QGauss<dim> quadrature_formula(2);
           QGauss<2>   face_quadrature_formula(2);

           FEValues<dim> fe_values(fesystem, quadrature_formula,
                                   update_values |
                                   update_gradients |
                                   update_JxW_values |
                                   update_quadrature_points);

           FEFaceValues<dim> fe_face_values (fesystem, face_quadrature_formula,
                                             update_values |
                                             update_gradients |
                                             update_JxW_values |
                                             update_quadrature_points |
                                             update_normal_vectors);

           const unsigned int dofs_per_cell = fesystem.dofs_per_cell;
           const unsigned int n_q_points    = quadrature_formula.size();
```

```
189.          const unsigned int nodes_per_cell =dofs_per_cell/dim;
190.
191.          std::cout << "Number of nodes per element: "
192.                    << nodes_per_cell
193.                    << std::endl;
194.
195.
196.          FullMatrix<double>   cell_matrix (dofs_per_cell, dofs_per_cell);
197.          Vector<double>       cell_rhs (dofs_per_cell);
198.
199.          std::vector<unsigned int> local_dof_indices (dofs_per_cell);
200.
201.
202.         typename DoFHandler<dim>::active_cell_iterator cell = dof_handler.begin
    _active(), endc = dof_handler.end();
203.         for (; cell!=endc; ++cell)                        // loop over element
204.           {
205.             cell_matrix = 0;
206.             cell_rhs = 0;
207.
208.             fe_values.reinit (cell);
209.           for (unsigned int q_index = 0; q_index<n_q_points; ++q_index)
210.             for (unsigned int A = 0; A < nodes_per_cell; A++)
211.               for (unsigned int i = 0; i < dim; i++)
212.                 for (unsigned int B = 0; B<nodes_per_cell; B++)
213.                   for (unsigned int k = 0; k<dim; k++)
214.                     for (unsigned int j = 0; j<dim; j++)
215.                       for (unsigned int l = 0 ;l<dim; l++)
216.                         {
217.
218.                         double stiffness = coefficient(fe_values.quadrature_point
     (q_index), i, j, k, l);
219.
220.                         cell_matrix(dim*A+i,dim*B+k) += (stiffness *
221.                                     fe_values.shape_grad(dim*A+i,q_index)[j]*
222.                                     fe_values.shape_grad(dim*B+k,q_index)[l]*
223.                                     fe_values.JxW(q_index));
224.                         // associated with solution d[B node, K dim]
225.                         }
226.
227.               Vector<double> traction(dim);
228.               traction = 0;
229.              for (unsigned int f = 0;f < GeometryInfo<dim>::faces_per_cell; ++f)

230.                {
231.                  const Point<dim> shell_center(0, 0, 0);
232.                  const double distance_to_face = shell_center.distance(cell->fac
    e(f)->center());
233.
234.                  if (cell->face(f)->at_boundary() &&std::fabs(distance_to_face -
     0.01) < 1e-4)
235.                  {
236.
237.                      fe_face_values.reinit(cell, f);
238.                      for (unsigned int face_q = 0; face_q<face_quadrature_formul
    a.size(); ++face_q)
239.                      {
240.                      double x = fe_face_values.quadrature_point(face_q)[0];
241.
242.                      traction[0] = 2000;
243.                      traction[1] = 0;
```

```
                    traction[2] = 0;

                       for (unsigned int A=0;A<nodes_per_cell;A++)
                           for (unsigned int i=0; i<dim;i++)
                                  {
                                     cell_rhs[dim*A+i] +=traction[i]*
                                     fe_face_values.shape_value(dim*A+i, face_q
  )*
                                     fe_face_values.JxW(face_q);
                                   }
                       }
                  }

            }
            cell->get_dof_indices (local_dof_indices);
            for (unsigned int A=0; A<dofs_per_cell; ++A)
              {
                  system_rhs(local_dof_indices[A]) += cell_rhs(A);
              for (unsigned int B=0; B<dofs_per_cell; ++B)
                  {
                  system_matrix.add (local_dof_indices[A],
                                     local_dof_indices[B],
                                     cell_matrix(A,B));
                               }
              }
          }

      std::ofstream outputFile ("output_f.txt");
          outputFile <<"" ;
          auto first = true;
          for (float f : system_rhs)
        {
          if (!first) { outputFile << " "; }
          first = false;
          outputFile << f;
        }
         outputFile << "" << std::endl;


           boundary_conditions();

           MatrixTools::apply_boundary_values (boundary_values,
                                               system_matrix,
                                               solution,
                                               system_rhs,
                                               false);
      }

       template <int dim>
       void CorneaElastic<dim>::solve()
       {
           SparseDirectUMFPACK A;
           A.initialize(system_matrix);
           A.vmult(solution, system_rhs);
       }


      template <int dim>
      void CorneaElastic<dim>::output()
      {
          solution_name.emplace_back("u_x");
```

```
304.            solution_name.emplace_back("u_y");
305.            solution_name.emplace_back("u_z");
306.
307.            DataOut<dim> data_out;
308.            data_out.attach_dof_handler(dof_handler);
309.
310.            data_out.add_data_vector(solution, solution_name);
311.            data_out.build_patches();
312.            std::ofstream outputgpl ("solt.gpl");
313.            data_out.write_gnuplot (outputgpl);
314.            std::ofstream outputvtk ("solt.vtk");
315.            data_out.write_vtk (outputvtk);
316.
317.
318.        }
319.
320.
321.        template <int dim>
322.        void CorneaElastic<dim>::run()
323.        {
324.            meshing();
325.            setup_system();
326.            assemble();
327.            solve();
328.            output();
329.        }
330.
331.        int main()
332.        {
333.            CorneaElastic<3> CorneaElastic;
334.            CorneaElastic.run();
335.            return 0;
336.        }
```

Appendix A.5: Replacement code for the inhomogeneous cornea

The following code can be used to model the inhomogeneous corneal deformation by replace the *coefficient* function in Appendix A.4.

```
1.  template <int dim>
2.  double CorneaElastic<dim>::coefficient(const Point<dim> &p_coeff,  unsigned int
    i,  unsigned int j, unsigned int k,  unsigned int l)/*function coefficient(Point
    , i, j, k, l)*/
3.  {
4.
5.      const Point<dim> shell_center(0, 0, 0);
6.      const double distance_to_cell = shell_center.distance(p_coeff) *100;
7.      double E = 2e6*distance_to_cell;
8.
9.      double nu = 0.49999;                                // nearly incompressible
10.     double lambda = (E*nu)/((1.+nu)*(1.-
    2.*nu));            // Lame constant lambda
11.     double mu = E/(2.*(1.+nu));                          // Lame constant mu
12.     double C = lambda*(i==j)*(k==l) + mu*((i==k)*(j==l) + (i==l)*(j==k));
13.     //stiffness tensor C_ijkl
14.     return C;
15. }
```

Appendix B

Copyrighted material

Permission to use Figure 2.5.

Copyright Clearance Center Marketplace™

McGraw Hill LLC - License Terms and Conditions

This is a License Agreement between Yuchen Song, Trent University ("You") and McGraw Hill LLC ("Publisher") provided by Copyright Clearance Center ("CCC"). The license consists of your order details, the terms and conditions provided by McGraw Hill LLC, and the CCC terms and conditions.

All payments must be made in full to CCC.

Order Date	21-May-2020	Type of Use	Republish in a thesis/dissertation
Order license ID	1036569-1	Publisher	MCGRAW-HILL PUBLISHING COMPANY
ISBN-13	9780070253100	Portion	Image/photo/illustration

LICENSED CONTENT

Publication Title	Introduction to materials science	Country	United States of America
Author/Editor	GUY, ALBERT GLASGOW	Rightsholder	McGraw Hill LLC
Date	01/01/1972	Publication Type	Book
Language	English		

REQUEST DETAILS

Portion Type	Image/photo/illustration	Distribution	Worldwide
Number of images / photos / illustrations	1	Translation	Original language of publication
		Copies for the disabled?	No
Format (select all that apply)	Print, Electronic	Minor editing privileges?	No
Who will republish the content?	Academic institution	Incidental promotional use?	No
Duration of Use	Life of current edition	Currency	USD
Lifetime Unit Quantity	Up to 499		
Rights Requested	Main product		

NEW WORK DETAILS

Title	The Adoption of a Finite Element Model of Material Deformation Relevant to Studying Corneal Biomechanics	Institution name	Trent University
		Expected presentation date	2020-05-28
Instructor name	Prof. Rayf Shiell		

ADDITIONAL DETAILS

Order reference number	LAJEVA9	The requesting person / organization to appear on the license	Yuchen Song, Trent University

REUSE CONTENT DETAILS

Title, description or numeric reference of the portion(s)	Fig: Four examples of the mechanical response of materials corresponding to (a) elastic, (b) plastic, (c) elastomeric, and (d) viscoelastic behaviour are illustrated using nominal stress-nominal strain curves.	Title of the article/chapter the portion is from	Introduction to Materials Science
		Author of portion(s)	GUY, ALBERT GLASGOW
		Issue, if republishing an article from a serial	N/A
		Publication date of portion	1972-01-01
Editor of portion(s)	N/A		
Volume of serial or monograph	N/A		
Page or page range of portion	Not known: from p. 318; Gersten & Smith "Physics..		