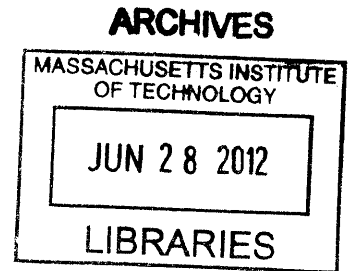
Skin Strain Analysis Software for the Study of Human Skin Deformation

by

Andrew T. Marecki

S.B. Mechanical Engineering
Massachusetts Institute of Technology (2010)

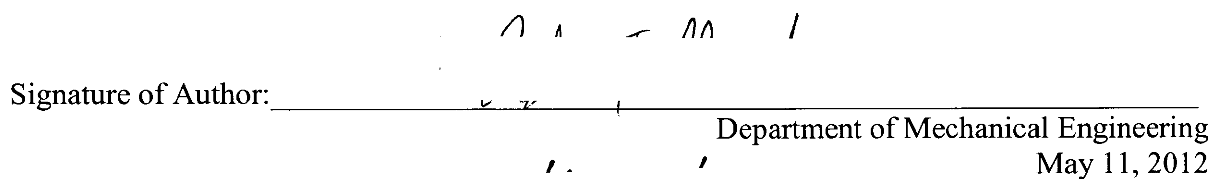

ARCHIVES
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
JUN 28 2012
LIBRARIES

Submitted to the Department of Mechanical Engineering
in partial fulfillment of the requirements for the degree of
Master of Science in Mechanical Engineering
at the

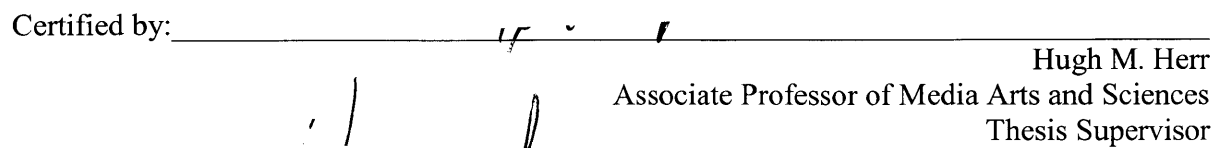
MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2012

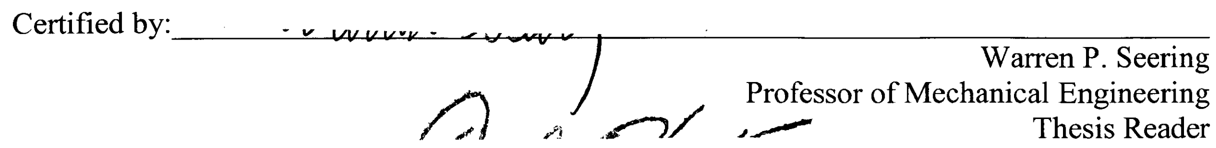
© Massachusetts Institute of Technology 2012. All rights reserved.



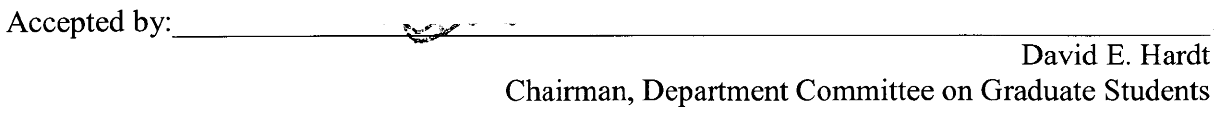
Signature of Author: ______________________
Department of Mechanical Engineering
May 11, 2012



Certified by: ______________________
Hugh M. Herr
Associate Professor of Media Arts and Sciences
Thesis Supervisor



Certified by: ______________________
Warren P. Seering
Professor of Mechanical Engineering
Thesis Reader



Accepted by: ______________________
David E. Hardt
Chairman, Department Committee on Graduate Students

Skin Strain Analysis Software for the Study of Human Skin Deformation

by

Andrew T. Marecki

Submitted to the Department of Mechanical Engineering
on May 11, 2012 in partial fulfillment of the
requirements for the degree of
Master of Science in Mechanical Engineering

Abstract

Skin strain studies have never been conducted in a precise and automated fashion. Previous in vivo strain investigations have been labor intensive and the data resolution was extremely limited such that their results were largely qualitative. There is a need for a better system to collect, compute, and output strain measurements of the skin in vivo for the purpose of designing better mechanical interfaces with the body. Interfaces that have the same strain behavior as human skin can minimize shear forces and discomfort for the user. One particular application is improving the design of prosthetic liners for amputees, creating a second skin sleeve that provides support without hindering movement.

A custom approach offering high resolution marker density, automatic point tracking and correspondences, and computational transparency is presented in this thesis. The entire computational toolbox is presented, which takes in high resolution digital photographs, tracks points on the surface of the body, corresponds points between body poses, computes a series of strain measures, and graphically displays these data. The results of studies of a full bodied human knee and a transtibial amputee's residual limb are presented here as well.

Thesis Supervisor: Hugh M. Herr
Title: Associate Professor of Media Arts and Sciences

Acknowledgements

I would like to acknowledge the contributions of all the members of the MIT Biomechatronics Group for their support. In particular, I'd like to thank Hugh Herr for advising me and helping to form the vision of this thesis, Bruce Deffenbaugh for his constant flow of ideas and advice, Grant Elliott for his mentorship of me as both an undergraduate researcher and a graduate student, and Sarah Hunter for keeping the entire lab group organized and sane. I would also like to thank Hazel Briner for her constant support throughout the long nights in the lab, both on research and class projects. I'd finally like to thank my family, Tom, Beverley, Madeleine, Tommy, Mel, Lena, Mikhaila, Serena, Evangeline, and Shane for their love throughout my entire life.

Contents

1 Introduction 6
1.1 Motivation 6
1.2 Thesis Contents 7

2 Literature Review 8

3 Strain Mechanics 14
3.1 Engineering Strain 14
3.2 Finite Strain Theory 16
3.3 Strain Transformations 19
3.4 Strain in the Context of Human Skin Deformation 21

4 Experimental Methodology 24
4.1 Photogrammetry 24
4.2 Point Detection 27
4.3 Correspondence 29
4.3.1 Iterative Closest Point Algorithm 32
4.3.2 Geometric Compatibility and Triangulation Correction 34
4.4 Strain Computations 36

5 Results 44
5.1 Validation 44
5.1.1 Zero Strain Test Case 44
5.1.2 Uniaxial Tension Test Case 47
5.2 Transtibial Amputation Study 51

6 Conclusions 54
6.1 Applications 54
6.2 Future Work 57

Bibliography 58
Appendices 59
Appendix A: Point Detection and Tracking 60
Appendix B: Correspondence Problem. 62
Appendix C: Strain Computations and Plotting 82

List of Figures

2.1 Iberall's lines of nonextension mapped over the entire human body 9
2.2 Longitudinal strain plot of a human knee from Bethke's study 10
2.3 Views of a human ankle with motion capture markers from Marreiros's study 12
3.1 Two dimensional deformation of a material element under normal strain 14
3.2 Deformation of a material element under shear strain 15
3.3 Deformation of a triangular element in three dimensions 16
3.4 Alignment of the normal vectors of two triangular elements 17
3.5 Geometric visualization of a singular value decomposition 18
3.6 Strain transformations graphically depicted by Mohr's circle 19
3.7 Strain states corresponding to two locations on Mohr's circle 20
3.8 Distortions of a circle to an ellipse as studied by Iberall 21
3.9 Strain states corresponding to different principal stretch ratios 22
3.10 Zero strain directions in an element experiencing positive and negative principal strains 23
4.1 Screenshot of Autodesk's 123D Catch photogrammetric program 26
4.2 Texture images from each step of the dot identification process 27
4.3 Reconstructed 3D model with filtered texture and point centers identified 27
4.4 Voronoi diagram and Delaunay triangulation of a two dimensional data set 30
4.5 Crust triangulation of a point cloud from a human knee 31
4.6 Local correspondence matching between two body poses 33
4.7 Partially solved correspondence problem with triangulation discrepancy 35
4.8 Fully solved correspondence problem for photogrammetric data of a knee 36
4.9 Front view of the equivalent strain plot of a knee during partial flexion 37
4.10 Back view of the equivalent strain plot of a knee 38
4.11 Strain field of a knee in flexion 39
4.12 Principal strain field of a knee with detail views 41
4.13 Back view of the zero strain field of a knee 42
4.14 Front view of the zero strain field of a knee 43
5.1 Photogrammetric model reconstructions of the graph paper test case 44
5.2 Equivalent strain plot of the graph paper test 45
5.3 Strain field of the graph paper test 46
5.4 Photogrammetric model reconstruction of the latex tubing test case 47
5.5 Equivalent strain plot of the latex tubing test 48
5.6 Strain field of the latex tubing test 50
5.7 Three reconstructed photogrammetric models of a transtibial residual limb 51
5.8 Triangulated point clouds of a transtibial residual limb 52
5.9 Equivalent strain plot of a transtibial residual limb study 52
5.10 Strain field of transtibial limb study with detail views 53

1. Introduction

Human limbs consist of a heterogeneous structure of bone, muscle, and skin. All of these components have their own mechanical properties, and it is this combination of behaviors which allows us to move dynamically. Bone creates structure, muscles are biological actuators, and skin creates a protective, elastic layer around the body. One of skin's main functions is to serve as the body's mechanical interface to the external world. The understanding of its mechanical behavior is a key aspect of biomechanical studies.

Skin strain has never been calculated in a precise and automated fashion. Arthur Iberall [1970] initiated studies of skin deformation as a result of joint motion on a living subject. His qualitative methods of assessing the directions of zero skin stretch shaped the design of his pressure suit and led the way for further research in this field. More recent attempts at quantifying the skin strain field were conducted by Kristen Bethke [2005] and Sara Marreiros [2010]. They used a 3D laser scanner and a motion capture system respectively to track points on the surface of the skin. However, the skin marker resolution was severely limited by both 3D capture systems and the data processing routines required significant user input. There is a need for an automated, high resolution system for optically assessing skin strain in vivo.

The purpose of this thesis is to explore how skin stretches on the human body through optical analysis. This system uses photogrammetric tools to reconstruct a 3D model of the body from digital photographs, which allows arbitrary marker density. The density of visually distinguishable dots on the body is much higher than that of motion capture or laser scanner systems. Furthermore, a software toolbox is developed to automatically analyze photographs of the body, track points on the surface, compute point correspondences, calculate various strain measures, and display the data in a number of ways.

1.1 Motivation

This thesis furthers the study of skin strain on living subjects primarily for the purpose of developing better mechanical interfaces with the body. This knowledge can contribute to the design of more comfortable second-skin interfaces for prosthetic or exoskeletal applications. Current prosthetic liners composed of silicon rubber do not dynamically adapt to changes in body posture. Understanding the strain behavior of the targeted body area, including the maximum, minimum, and zero strain fields is a critical part of designing a better interface.

Secondary applications of the technology developed in this thesis include improving the design of high performance, form fitting clothing. Knowing how the skin deforms can directly correlate with the materials selection and geometry of clothing design. Yet another application is the improvement of computer animations, particularly of the human face. This system can be used to automatically assess a number of different facial gestures and provide a quantitative analysis of the skin's deformation. Chapter 6 continues this discussion in detail.

1.2 Thesis Contents

This thesis presents the previous work on analyzing skin strain in vivo and subsequently details every step of this particular skin strain study. It explains strain mechanics and the finite element model, details the experimental methods and computational processes, and shows a series of pilot studies using these methods.

Chapter 2 provides an overview of previous work on studying skin strain that results from joint motion. This section details the methods, contributions, and areas for improvement of each study.

Chapter 3 presents background on basic strain mechanics and finite strain theory. It describes the strain concepts and equations that are implemented in the custom finite element software.

Chapter 4 explains the rest of the computational pipeline of the experimental method. This includes the photogrammetric methods used to create the 3D reconstruction of the surface of the body, marker tracking of the control points, and the correspondence solver used to relate points between various body poses. It also shows how the strain information can be displayed graphically in several ways.

Chapter 5 shows pilot results from a series of studies using this system. Two test cases verify the accuracy of both the photogrammetric and computational methods. The results of a high resolution test on a transtibial amputee's residual limb is also presented here.

Chapter 6 concludes this thesis by reflecting on the presented study and discussing both the applications of this optical strain analysis software and future developments on this project.

2. Literature Review

There have only been a few studies on the effect of joint motion on skin deformation in vivo. The study of skin strain on a living subject is critical to the design of mechanical interfaces with the body. The knowledge of the behavior of a surface is one key dimension, which when coupled with other studies, such as tissue depth and stiffness can drive the design of a better mechanical interface with the body.

The work of Arthur Iberall, published in 1970, analyzed the strain field of human skin in vivo during joint motion [8, 9]. He is credited for being the first to identify the "lines of nonextension" along the surface of the body. Along these paths, there is virtually no skin stretch, only shape changes as joint posture is varied. Figure 2.1 shows the lines of nonextension over the surface of the entire body as identified by Iberall. These lines represent potential locations for constraining or attaching to the human body. Since these lines of nonextension experience zero strain by definition, axially rigid materials that are still flexible in bending can be used in these areas. Iberall used inextensible cabling along the lines of nonextension to provide structure for a mechanical counter pressure suit while still allowing full range of motion.

Iberall's study represented a first step in skin strain studies on a living subject. While his mapping is qualitative only, it showed the potential for both how to study skin deformations and how to use these data as design rules for a mechanical interface with the body.

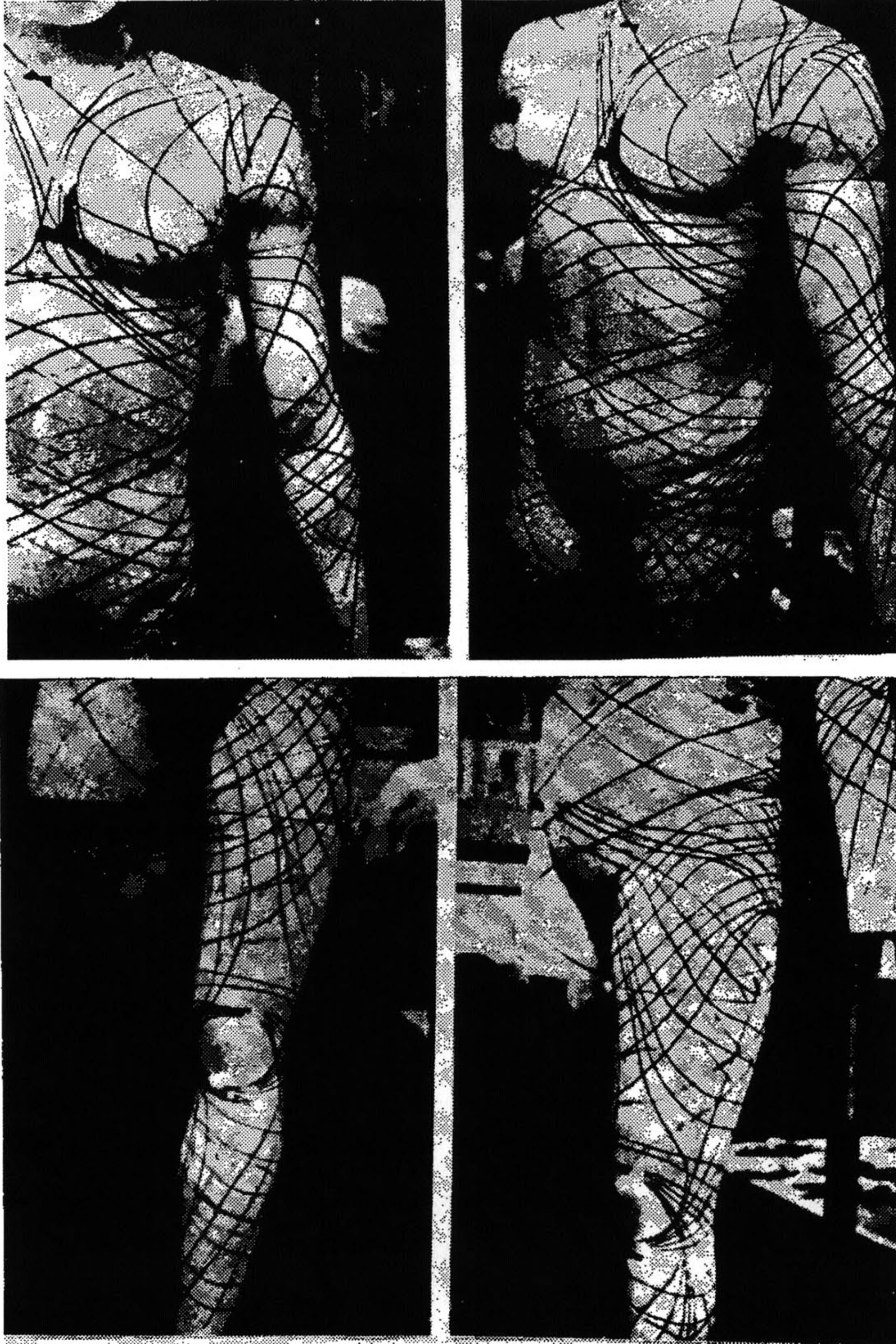


Figure 2.1: Iberall's lines of nonextension mapped over the entire human body [8, 9].

In 2005, Kristen Bethke at the Massachusetts Institute of Technology analyzed the strain field of the human knee joint [4]. Her master's thesis detailed her methods of collecting 3D skin marker location data and computing the skin strain field with the intention of improving space suit design. Bethke measured skin's response to motion through the use of a 3D laser scanner. She first marked the leg with 156 stiff paint dot markers that were roughly 3cm away from one another. The Cyberware whole body 3D laser scanner then created a model of the human leg at various knee flexion angles. The marker points were identified manually from the laser scan, distinguishable by their raised appearance on the surface of the body. Figure 2.2 shows the pilot results from Bethke's knee strain study.

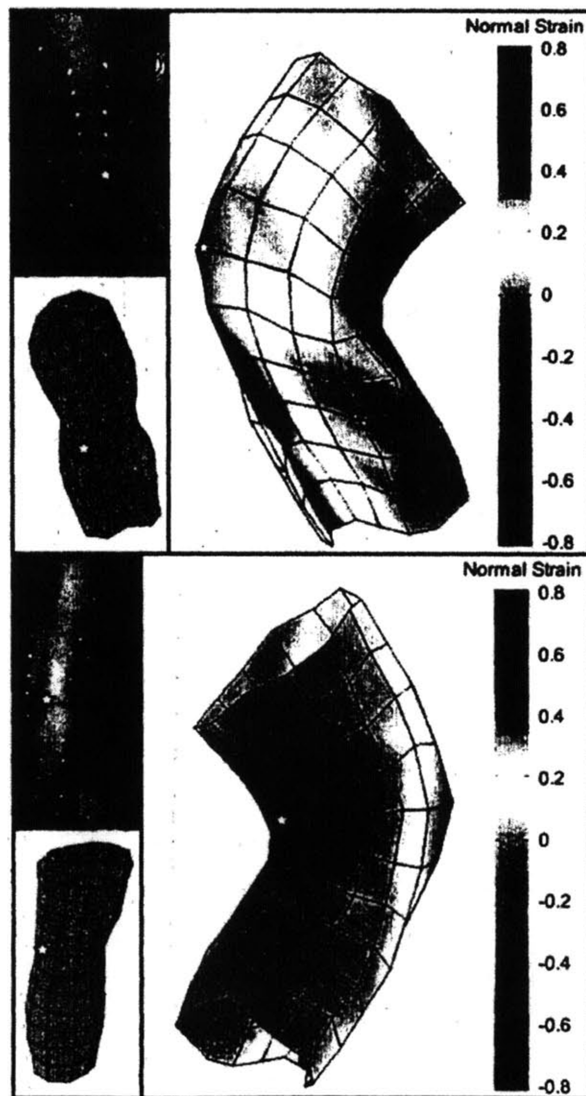


Figure 2.2: Longitudinal strain plot of a human knee during flexion from Bethke's study. Note the use of a rectangular grid and the 3cm vertex resolution [4].

Bethke's strain analysis was based on the square grid of points along the leg's surface visible in figure 2.2. Her calculations used the eight nearest neighbors to calculate the strain field to calculate the average strain at the various points on the body. This mean average approach, coupled with the study's relatively low resolution means that the results only offer a general understanding of the strain on the body's surface. Furthermore, the rectangular point distribution does not represent the actual surface of the body as well as a comparable triangularized surface. More accurate local strains can also be calculated from the first principles of basic linear, isotropic, elastic mechanics for the triangularized cloud.

Bethke's computational methods also involve an error checking and correction routine. For some unknown reason her data processing scripts occasionally yielded strain component values that were beyond the failure strain of human skin. To fix these errors, she used the average of the strains in the neighboring locations. There is no explanation of why these physically impossible strains occur in the data. While Bethke provides a general method of capturing skin strain data, the marker resolution and computational methods do not provide the level of detail needed to make a second skin mechanical interface.

In her master's thesis for the University of Lisbon in 2010, Sara Marreiros studied the skin strain field of a human ankle joint [11]. She studied the ankle's strain field with the motivation of designing a second skin ankle foot orthosis to correct drop foot, a gait feature frequently caused by a stroke. Her study used a proprietary motion capture system with a limited capture volume and relatively low marker density. The data processing procedure also required significant user input and used ABAQUS for the finite element analysis.

Marreiros used the Qualisys Motion Capture System with four infrared cameras to collect the three dimensional information for the strain analysis. This expensive, proprietary system could only capture a complete 360° view of the ankle because it only had four cameras. Infrared reflective stickers were placed on a 2cm grid over the surface of the ankle. For the two presented trials, 123 and 141 markers were used, respectively. Figure 2.3 shows the marker configuration of one trial.

Figure 2.3: Sample images from Marreiros's study of the human ankle joint. Note the 2cm grid of infrared markers placed on the body [11].

This motion capture system requires significant user input to obtain useable 3D information. All of the markers must be manually identified by the user and errors in the marker trajectories had to also be corrected manually. Another important factor in the acquisition of the spatial information is the lack of precision in the calculated positions and trajectories. Marreiros commented that the use of only four cameras to reconstruct only a partial view of the ankle was a significant limitation.

The strain computations were performed by ABAQUS, a proprietary finite element analysis program. This program implements a nonlinear analysis of the deformation of the point cloud and outputs the strain field information. Marreiros provides only pilot results of her methods, providing no verification of the accuracy of her analysis software. While she presents a general analysis of the skin strain behavior around the ankle joint, much can be done to improve the accuracy, transparency, and automation of her process.

The strain analysis software presented in this thesis addresses many of the shortcomings of previous research in this field. The first issue is marker resolution. Iberall used circular ink stamps to manually assess the strain field on the body, which meant that the resolution of his study was proportional to the time spent on it. Bethke and Marreiros used more sophisticated data collection methods, namely laser scanning and motion capture, respectively but encountered other barriers to obtaining higher resolution data. As aforementioned, their marker resolutions

were limited to approximately 2cm by these methods. The proposed optical methods use digital photographs for the 3D model reconstruction. If the dots on the body are distinguishable from the photographs, they will also be identifiable in the reconstruction. The resolution of marker points can therefore be arbitrary and set according to the desired level of detail.

The software package presented here also minimizes user input in the data processing steps. Unlike the other studies, the proposed method handles point identification and correspondence automatically. Once the 3D information is obtained, the strain computations are executed by a custom finite element toolbox. All of the strain functions described in chapter 3 are derived from the basic principles of linear, elastic mechanics. This system is designed to be as transparent and automatic as possible. The rest of the computational methods are described in detail in chapter 4.

3. Strain Mechanics

The strain analysis tool presented here uses basic elastic strain relations to calculate how the skin deforms. A finite element analysis program is created to facilitate this study by taking absolute position and triangulation data as inputs and outputting a strain field analysis. These results include normal and shear strains with respect to both the original coordinate system and the principal coordinate frame.

3.1 Engineering Strain

Strain is a normalized measure of a deformation relative to a reference distance. It is a dimensionless quantity which describes the percentage or fractional change in length of a body. Equation 3.1 defines engineering strain, which is based on the specimen's initial length, L_o, rather than its overall length after deformation, $L+L_o$. Here, l is the total length of the deformed configuration and δ is the relative deformation [7].

$$\varepsilon = \frac{\delta}{L_o} = \frac{L-L_o}{L_o} \tag{3.1}$$

In three-space, strain can be represented by a second order tensor. It is a linear map between input and output deformation vectors. The principle of linearity allows us to examine the components of strain independently. The second order strain tensor is shown in equation 3.2.

$$\underline{\varepsilon} = \begin{bmatrix} \varepsilon_{xx} & \varepsilon_{xy} & \varepsilon_{xz} \\ \varepsilon_{yx} & \varepsilon_{yy} & \varepsilon_{yz} \\ \varepsilon_{zx} & \varepsilon_{zy} & \varepsilon_{zz} \end{bmatrix} \tag{3.2}$$

The diagonal components of this tensor represent normal strains and the off-diagonal components are shear strains. Normal strains, depicted in figure 3.1, indicate scaling transformations.

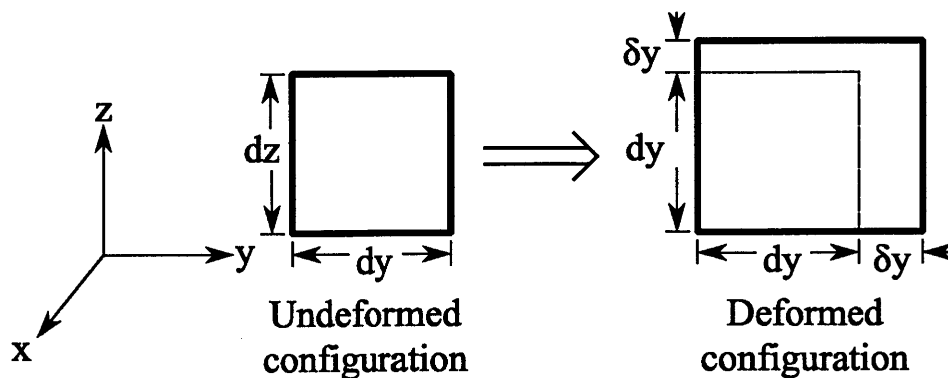


Figure 3.1: Deformation of a material element experiencing normal strains.

For the figure shown in figure 3.1, the normal strains in the y and z directions are calculated by equations 3.3 and 3.4, respectively.

$$\varepsilon_{yy} = \frac{\delta y}{dy} \tag{3.3}$$

$$\varepsilon_{zz} = \frac{\delta z}{dz} \tag{3.4}$$

Shear strains correspond to angular deformations between two adjacent vectors. This can be caused by the translation of one parallel plane relative to another. Figure 3.2 depicts a simple shear case in which the edges of a material element are rotated.

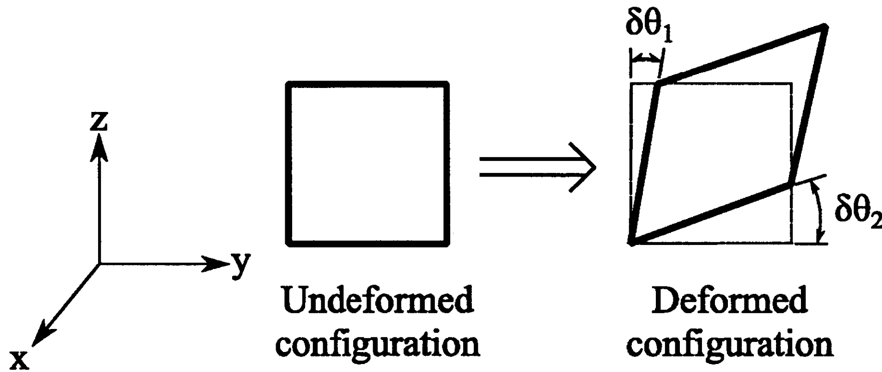


Figure 3.2: Deformation of a material element experiencing shear strains.

The shear strain experienced by the material element in figure 3.2 is calculated by equation 3.5. In this equation, γ is the engineering shear strain relative to an orthogonal basis.

$$\varepsilon_{yz} = \varepsilon_{zy} = \frac{1}{2}\gamma_{yz} = \frac{1}{2}(\delta\theta_1 + \delta\theta_2) \tag{3.5}$$

The linearity of strain mechanics allows us to superimpose various strain states, resulting in an effective strain configuration. We can also analyze the components of strain independently, compile the results, and even perform transformations on these strain states. This strain theory extends to three dimensions, but only two dimensions are analyzed in the presented strain study. Only triangular faces, not tetrahedral volumes, are assessed as we are investigating how the surface of the skin deforms.

3.2 Finite Strain Theory

The basic unit of this finite element analysis is a constant strain triangle (CST). This means that each triangle has exactly one strain state with no variation allowed throughout the element. The CST architecture sacrifices accuracy, which can be remedied by tracking a greater number of triangles via increased skin marker density. In this analysis, the deformation of every triangle is computed individually and is independent of all the other elements.

The analysis of the deformation of one triangle involves a sequence of linear algebra operations. Figure 3.3 shows the undeformed and deformed configurations of a triangle in three-space. The undeformed, original triangle has vertices, p_1, p_2, p_3, and associated normal vector, N_1. The deformed triangle has vertices, p'_1, p'_2, p'_3, and associated normal vector, N_2. The two normals are defined by the cross product of any two adjacent vectors of each respective triangle.

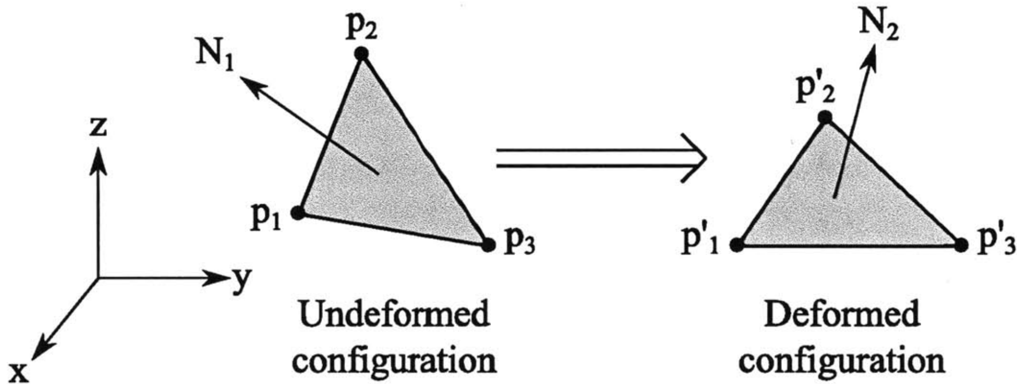


Figure 3.3: Deformation of a single triangular element.

Equations 3.6 and 3.7 define the normal vectors in terms of the vertices of each element.

$$N_1 = (p_1 - p_2) \times (p_1 - p_3) \tag{3.6}$$

$$N_2 = (p'_1 - p'_2) \times (p'_1 - p'_3) \tag{3.7}$$

To analyze the deformation of the two-dimensional triangle in three-space, both configurations of the triangle must be rotated to the same coordinate frame. An inverse Euler rotation is used to identify the rotation matrix needed to align the two normal vectors. A rotation matrix is computed and applied to each triangle to align the respective normal with the z axis. Figure 3.4 shows the rotated triangles which can now be analyzed in two dimensions. Note that this operation is a rigid body rotation to a new coordinate frame, so the triangles are preserved. The 'r' represents the new dimensions and coordinates in the rotated frame.

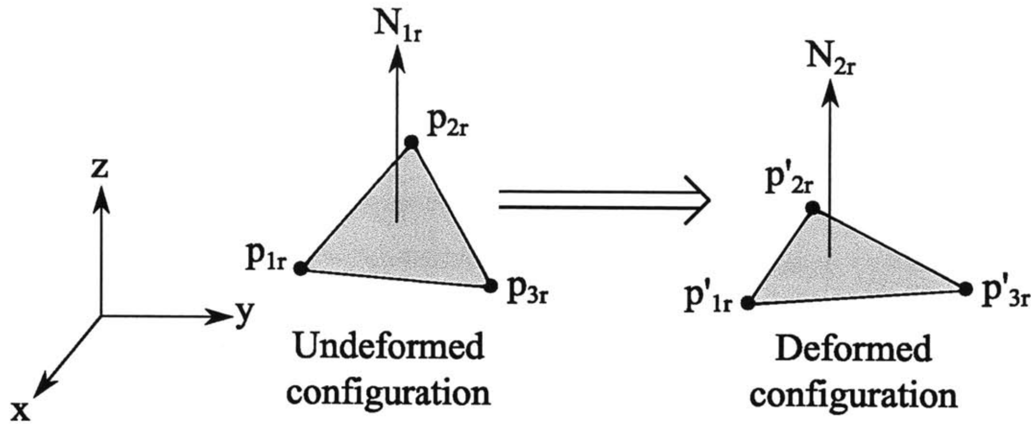


Figure 3.4: Both triangle configurations are rotated to the same plane so the two dimensional deformation can be analyzed.

The two dimensional analysis derives the affine transform that links the two triangle configurations. In two-space, the basic equation governing an affine transform is given by equation 3.8. Here the *x* vector represents the undeformed configuration, while *y* represents the deformed one. A is a matrix that transforms the original vector, which may include rotation and scaling. The b vector represents a pure translation.

$$\vec{y} = A\,\vec{x} + \vec{b} \tag{3.8}$$

The affine transform can be rewritten in a more useful way, containing only a single matrix multiplication operation, if an augmented matrix is used. Equation 3.9 presents the affine transformation matrix in its augmented form.

$$\begin{bmatrix} \vec{y} \\ 1 \end{bmatrix} = \begin{bmatrix} A & \vec{b} \\ 0, \ldots 0 & 1 \end{bmatrix} \begin{bmatrix} \vec{x} \\ 1 \end{bmatrix} \tag{3.9}$$

Equation 3.9 can be used in this particular situation to relate the two triangle configurations. Once the triangles have been rotated to be parallel to the x-y plane and therefore have parallel normal vectors parallel to the z axis, we can reduce this to a two-dimensional problem. Since the z coordinates of the triangles now simply correspond to a translation, this dimension can be neglected. The affine transform can be calculated from the three initial coordinate pairs and the three final pairs. The affine transformation matrix requires equation 3.10 to be satisfied for each corresponding coordinate pair. Here A is a 2x2 matrix governing rotations and scaling.

$$\begin{bmatrix} x_f \\ y_f \\ 1 \end{bmatrix} = \begin{bmatrix} & A & \Delta x \\ & & \Delta y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_i \\ y_i \\ 1 \end{bmatrix} \tag{3.10}$$

Once the 3x3 affine transformation matrix is calculated, we can pull out the A matrix to identify the strain of this particular triangular element caused by the deformation from the initial to the final configuration. A singular value decomposition (SVD) of A yields three 2x2 matrices, U, Σ, and V^*. Equation 3.11 shows the SVD of the A matrix.

$$A = U\Sigma V^* \tag{3.11}$$

U is a real unitary matrix containing the left singular vectors of A and V^* contains the right singular vectors. Since A is always positive definite, U and V* are simply rotation matrices. Σ is a diagonal matrix that contains the singular values of A in descending order. The singular values in two-space can be geometrically interpreted as the semiaxes of an ellipse, which are one half of the major and minor axes. In terms of strain, the first and second singular values correspond to the maximum and minimum principal strains of the A matrix, respectively. Figure 3.5 shows a visualization of the SVD decomposition. This representation is geometrically significant because the third configuration shows the principal strain directions of the triangular element and the fourth image shows the strain directions in the original coordinate frame. Here the vector lengths represent normal strains and the angle between the vectors represents shear strain.

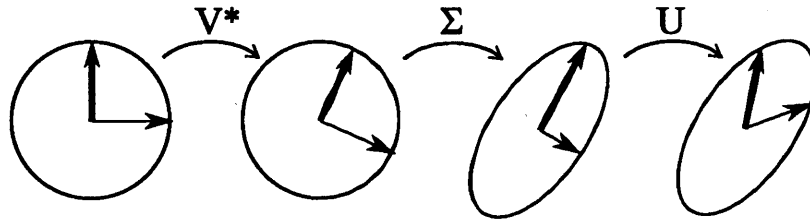


Figure 3.5: Geometric visualization of a singular value decomposition.

The SVD yields all of the information we need to calculate the strain state in the original and the principal coordinate frame. Furthermore, this analysis can be applied to every finite element to show the deformation of an entire body. The following section describes in detail how this SVD information is interpreted in terms of strain states in varying coordinate frames.

3.3 Strain Transformations

The singular value decomposition of the affine transformation matrix yields the principal strains and the rotation matrices that relate the original coordinate system to the principal frame. While coordinate systems are arbitrary, the principal strains represent the largest and smallest absolute deformation of an element All of the possible strain states in a two dimensional element can be described graphically by Mohr's circle. Figure 3.6 shows Mohr's circle for a general strain state. The horizontal axis corresponds to normal strains and the vertical axis represents shear strains. Every point on the circle has a corresponding ordered pair with a normal strain value and a shear strain value. This ordered pair represents the strain state on one face of the element. The strain state on the other face, which is perpendicular to the original one, is described by the ordered pair that lies diametrically across from the original point. The principal strain state is defined by the strain states that lie on the normal strain axis where the shear strains are zero. One elemental face has a normal strain ε_1 and the other has a smaller normal strain ε_2.

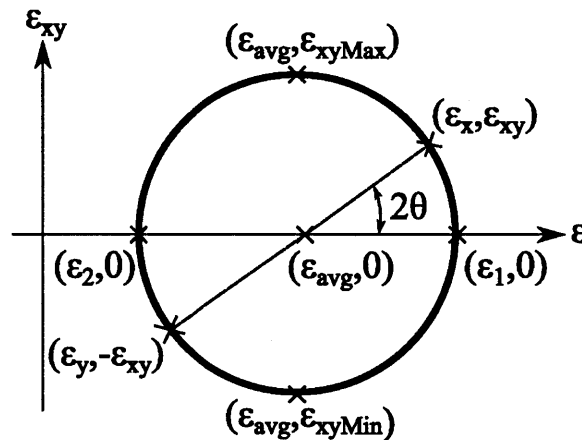


Figure 3.6: Mohr's circle is a graphical representation of strain transformations.

The connection between the strain state of an element and its representation on Mohr's circle is further clarified by figure 3.7 below. It depicts one strain configuration corresponding to the principal frame and another that is rotated by an angle θ counter clockwise from the major principal axis.

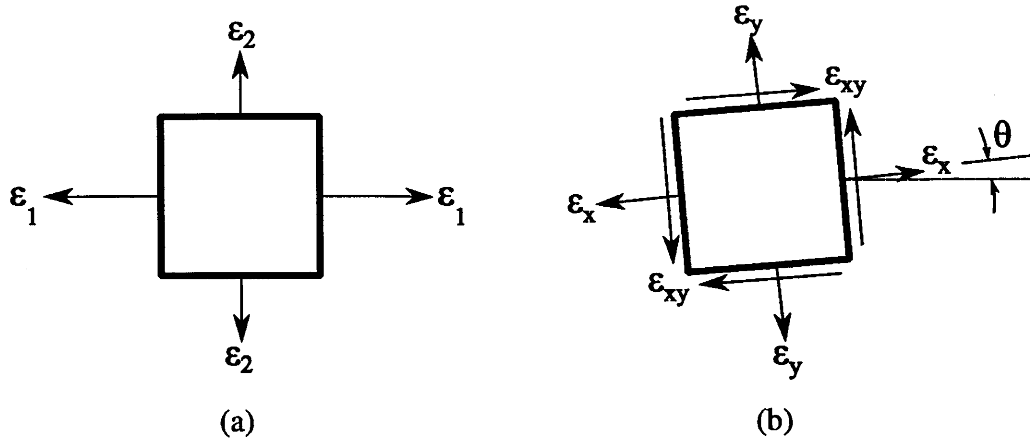


Figure 3.7: Strain states corresponding to Mohr's circle shown in figure 3.6. State (a) is represented by the principal coordinates on the normal strain axis and state (b) is represented by the strain configurations on the dotted line.

The principal strains and maximum shear strain can be calculated explicitly by equations 3.12 and 3.13, respectively [7]. The minimum shear strain is the opposite of the maximum shear strain. The principal strain state can be found with the information about one strain state alone.

$$\varepsilon_{1,2} = \frac{\varepsilon_x+\varepsilon_y}{2} \pm \sqrt{\left(\frac{\varepsilon_x-\varepsilon_y}{2}\right)^2 + \varepsilon_{xy}{}^2} \tag{3.12}$$

$$\varepsilon_{xyMax} = \sqrt{\left(\frac{\varepsilon_x-\varepsilon_y}{2}\right)^2 + \varepsilon_{xy}{}^2} \tag{3.13}$$

The overall strain of an element can also be quantified as a scalar value. Equation 3.14 computes the von Mises or equivalent strain ε_e from the principal strains, ε_1 and ε_2 [10]. This is a simplification of the traditional equivalent strain formula as there is no strain in a third dimension. The elements in this analysis are two dimensional surface patches, so average strain is calculated in the plane of each.

$$\varepsilon_e = \frac{1}{2}\sqrt{(\varepsilon_1 - \varepsilon_2)^2 + \varepsilon_1{}^2 + \varepsilon_2{}^2} \tag{3.14}$$

A scalar value is useful for displaying the strain results in one dimension. Average strain can be mapped to a range of colors and plotted on the surface of constant strain triangles. This is useful for understanding the overall strain behavior of a body.

3.4 Strain in the Context of Human Skin Deformation

Since this strain analysis tool is primarily intended to study the behavior of human skin, we can investigate certain aspects of skin deformation to further our knowledge about mechanical interfaces with the skin. One can argue that the optimal interface with the skin is a material that behaves like skin itself, stretching in the same ways as the joints are flexed.

Areas of minimum and maximum extension are likely to be important in interface design. By extracting the minimum and maximum principal strains from each triangular element we can create maps of minimum and maximum extension, respectively. More extensible materials with a low elastic modulus could be used in areas of maximum extension, while stiffer materials could lie along minimum extension areas.

Areas of zero strain, where there is neither extension or contraction, are another key area of interest. Just as Iberall identified lines of non extension (LONE), we can identify directions of zero strain from the previous strain transformation calculations. Iberall used ink stamps to view the distortion of a circle to an ellipse during joint motion as shown in Figure 3.8. The major and minor axes of each ellipse represented the directions of local maximum and minimum strain, respectively.

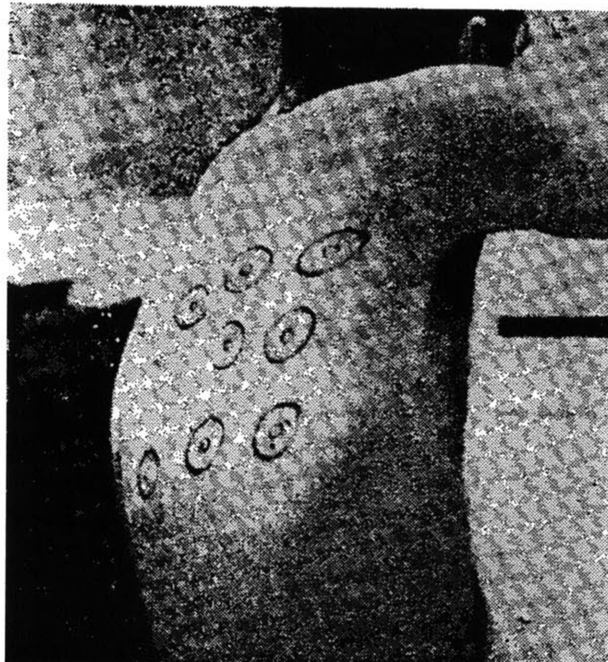


Figure 3.8: Distortions of a circle to near-elliptical form [8, 9].

Triangular elements that experience zero strain in some direction can be identified by their principal stretches. If the maximum principal stretch is greater than zero and the minimum principal stretch is less than zero, then there exist two directions in that element along which the strain is zero. Figure 3.9 depicts the various strain states that are possible in a two dimensional deformation.

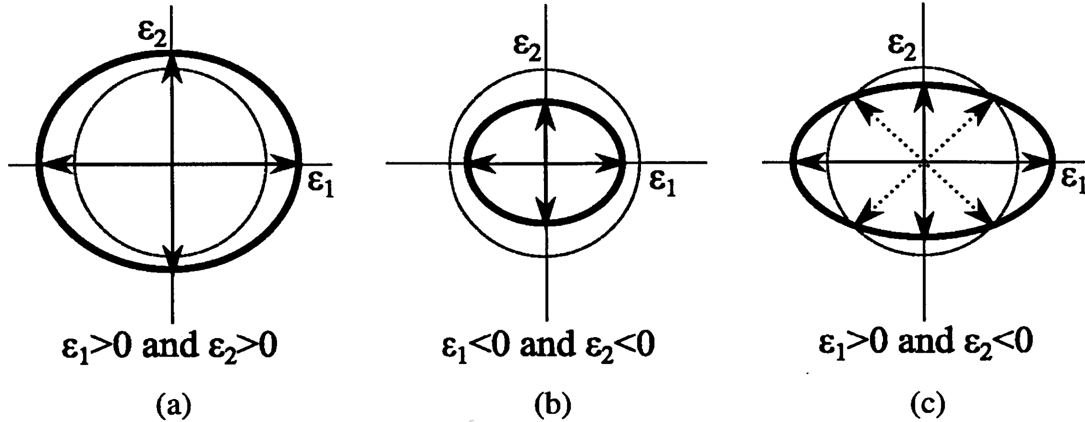


Figure 3.9: Strain states corresponding different principal stretch ratios. The dashed circles represent the original shape of the element and the solid ellipses represent the deformed configurations. State (a) represents positive dilation in all directions, while state (b) shows negative dilation for both principal stretches. State (c) has both a positive and a negative stretch ratio. The dotted vectors show the directions of zero strain.

For any ellipse corresponding to case (c) in figure 3.9 there exists two non-extension directions at two respective angles with respect to the principal axes. These angles depend on the amount of deformation in each of the principal directions which defines the ellipse's shape. The directions of the lines of non-extension are calculated using the equation of the ellipse defined by the principal stretches and identifying the points where it intersects the unit circle. Equation 3.15 is the equation of an ellipse in Cartesian coordinates centered at the origin with semi-major axis a and semi-minor axis b.

$$\frac{x^2}{a^2} + \frac{y^2}{b^2} = 1 \tag{3.15}$$

Polar coordinates can be used to identify the locations where the unit circle intersects the ellipse. Figure 3.10 depicts this particular strain state and show the directions and angles of the zero strain field. The maximum principal stretch is equivalent to the length of the major axis, which is twice the length of the semi-major axis. Similarly, the minimum principal stretch

corresponds to the ellipse's minor axis. The angles θ_1 and θ_2 where these intersections occur can be computed from the principal stretches.

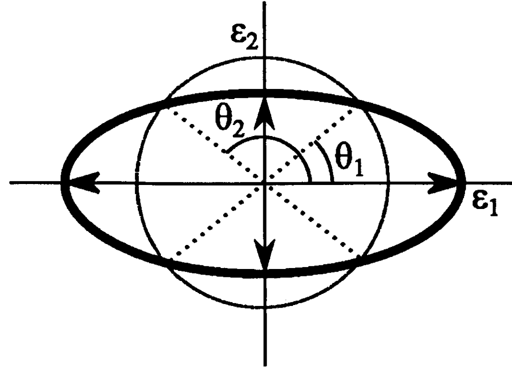


Figure 3.10: Directions of zero strain and their corresponding angles with respect to the maximum principal axis.

Using the respective polar substitutions for *x* and *y* and the principal stretches, equation 3.16 can be derived from equation 3.15.

$$\frac{(r*\cos\theta_1)^2}{\left(\frac{\varepsilon_1}{2}\right)^2}+\frac{(r*sin\theta_1)^2}{\left(\frac{\varepsilon_2}{2}\right)^2}=1 \tag{3.16}$$

Equation 3.16 can be solved for the angle θ_1 that corresponds to the intersection between the ellipse and the unit circle, where *r=0.5*. Simplifying equation 3.16 with this information yields equation 3.17.

$$\frac{(\cos\theta_1)^2}{(\varepsilon_1)^2}+\frac{(sin\theta_1)^2}{(\varepsilon_2)^2}=1 \tag{3.17}$$

By symmetry and supplementary angle relations, θ_2 can be computed in terms of θ_1. Equation 3.18 calculates $\theta_2$in radians.

$$\theta_2=\pi-\theta_1 \tag{3.18}$$

These zero strain vectors are subsequently rotated to the original coordinate frame of the respective element using information from the *U* rotation matrix of the element's SVD. This angular displacement is computed from the quadrant-corrected arctangent of matrix *U's* row elements. Equation 3.20 explicitly calculates the angle φ of the principal frame with respect to the original basis using the definition of *U* in equation 3.19.

$$U = \begin{bmatrix} u_{11} & u_{12} \\ u_{21} & u_{22} \end{bmatrix} \quad (3.19)$$

$$\varphi = \arctan\left(\frac{u_{12}}{u_{11}}\right) = \arctan\left(\frac{u_{22}}{u_{21}}\right) \quad (3.20)$$

Vectors corresponding to these non-extension directions are plotted on the virtual reconstruction of the body to create the zero strain map.

4. Experimental Methodology

The skin strain field analysis procedure consists of taking photographs of the target in various deformed poses, reconstructing a three dimensional model of each configuration from the two dimensional images, and computing the strain transformations from the geometric data. The data analysis is designed to be fully automated, taking in two dimensional images and producing a set of three dimensional strain plots. This is extremely significant as the scripts are able to perform every aspect of the processing without user input. A basic digital camera is used to collect the photographs for the model reconstruction and point tracking. The three dimensional reconstruction is computed by Autodesk's 123D Catch program, a free, cloud-computing based service. MeshLab, a free, open source system for processing and editing 3D triangular meshes, is used to prepare the 3D models for analysis. The rest of the data processing steps are accomplished by Matlab scripts. These scripts filter and identify the points on the three dimensional model, correspond their locations in each posture, and compute the strain behavior of the surface of the body.

4.1 Photogrammetry

There are a number of programs that can reconstruct 3D scenes from a set of digital photographs, including Autodesk's 123D Catch [2], ARC 3D [1], Microsoft's PhotoSynth [13], and Bundler [5]. These are often based on structure from motion (SFM) algorithms in which multiple views of an object, provided by movement of the camera can provide enough information to create a 3D digital surface model. Such an algorithm does not require prior knowledge of the scene contents, the camera, or correspondences between features in the scene. The SFM algorithm employs methods similar to how creatures with eyes can recover 3D structure information from 2D retinal images. It identifies similar features in each image and

tracks their trajectory throughout the image sequence. Every digital photograph also has a focal depth tag which provides additional information for the reconstruction.

This method is flexible, only requiring line of sight to gather the necessary image information. No feature markers are required unlike traditional motion capture systems that depend on infrared reflective markers for tracking. Another requirement for a proper 3D reconstruction is that every desired region or point on the target must be visible in at least three or four images so its location can be correctly triangulated. For every face of a cylindrical body to be reconstructed, photographs are required about every fifteen degrees about the target. Additional rings of photographs taken along the axial direction of the cylinder can improve accuracy through additional views of the features and increase the capture volume. In general, approximately twenty photographs can very accurately capture the entire human leg, from the hip to the ankle.

There is almost no restriction on the capture volume of a photogrammetric SFM system. It is able to reconstruct models for targets of a variety of sizes as long as the object can be fully captured in the photo frame. This flexibility makes it a suitable method for analyzing the variable shape of the human body.

Autodesk's 123D Catch was photogrammetric SFM program of choice for this strain analysis software. This free, cloud-computing based service can reconstruct a set of thirty images often in under ten minutes. While its automatic 3D reconstruction is extremely powerful, the program also allows manual correspondences for increased accuracy in any problem areas. Such problems in the model reconstruction can be caused by inconsistent lighting or specular surfaces. Figure 4.1 shows a screenshot from 123D Catch depicting the 3D model reconstruction of a transtibial amputee's residual limb. This program can also export files in a number of formats, including the wavefront .obj file format. This open format is text-readable and lists all of the information about the 3D geometry, including vertices, faces, normals, and texture coordinates. This means that it can easily be imported into a mathematics program, such as Matlab. The .obj file also includes an image file that contains all of the textures which are mapped to the surface. The texture image is the key to automatically tracking the marks painted on the skin, a method discussed in the subsequent section.

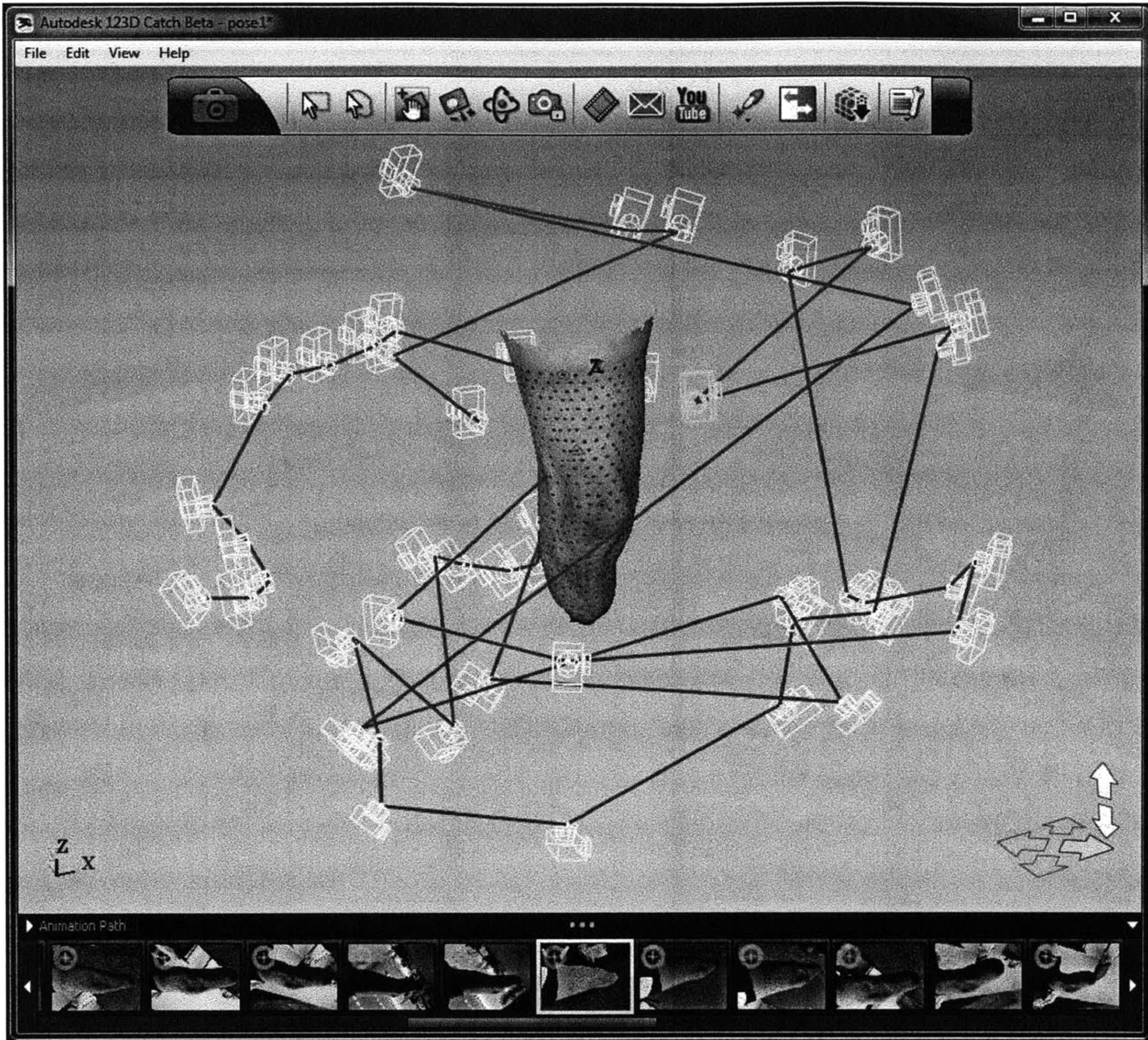


Figure 4.1: Autodesk 123D Catch program displaying the 3D model reconstruction of an amputee's residual limb. Note the input images below the model as well as the projected camera positions outlined in white.

The SFM method as employed by these programs is a flexible, accurate, and inexpensive means of generating a 3D model. The detailed 3D information as well as the necessary photographic textures make this system ideal for the strain analysis of human skin. This analysis investigates how the skin on the body deforms as joint posture is varied. This requires skin markings which do not impede the movement of the skin. Dots produced by an ink marker were chosen for their ease of application, consistent size and shape, and high contrast with skin.

4.2 Point Detection

The next step of this automatic strain analysis pipeline is the identification of marker points on the 3D model of the body part. This is an important part of the process that significantly reduces the need for user input in the data processing procedure.

A two dimensional median filter is used to isolate the dots in the .obj texture image. Median filtering is a nonlinear operation that reduces noise and preserves edges in an image. While often used to remove noise, such as salt and pepper noise, it can also be used to enhance these features. In this case, the noise is the dot pattern on the body. A two dimensional median filter computes a specific output pixel by taking the median value of a local neighborhood of points. Here it identifies the image of the skin without the marker dots by smoothing out changes in surface color and lighting so the dots disappear. A simple subtraction operation between the filtered image and the original leaves only the dots behind. Finally, inverting this image can produce dark dots on a light background with high contrast. Figure 4.2 shows the texture image from a study of a transtibial amputee's residual limb as it is being processed by the point identification script. Image (a) shows the original texture exported from the photogrammetric program, while image (b) is the median filtered texture. Image (c) is the difference between images (b) and (a). The final operation is an inversion of image (c) to produce image (d). The only significant marks on image (d) are the marker dots. In this particular study, 706 points on the body were tracked, which is significantly higher than the previous strain studies that used a maximum of 156 markers.

The properly filtered texture can now be wrapped on the model surface so the 3D locations of the control points can be identified in MeshLab. Image (a) of figure 4.3 shows the 3D model of a transtibial amputee's residual limb with the filtered texture wrapped on the surface. Mapping the texture to the surface and filtering by color enables the selection of the triangular faces that touch the dots. This sparse surface is extracted and exported for further processing.

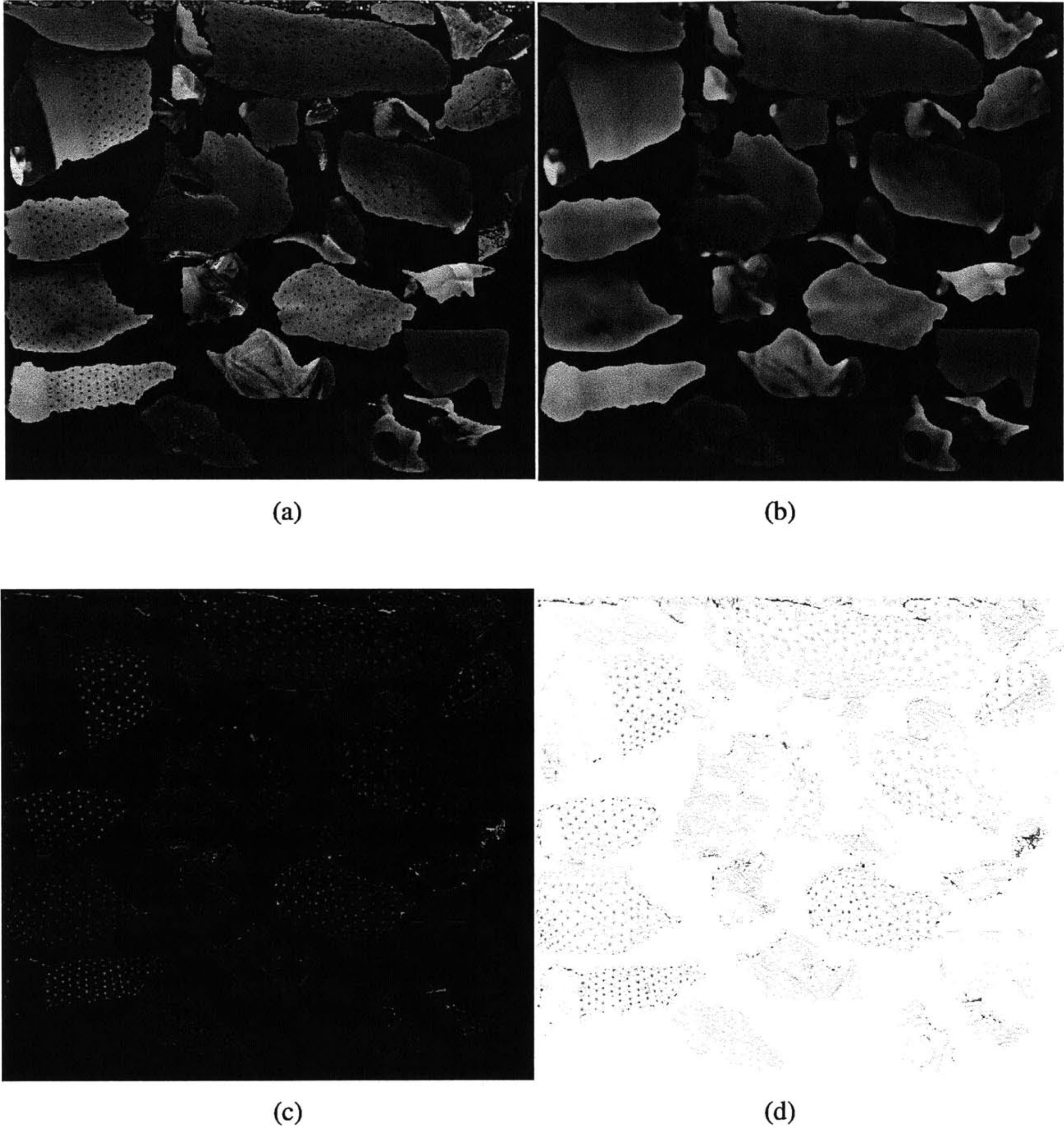
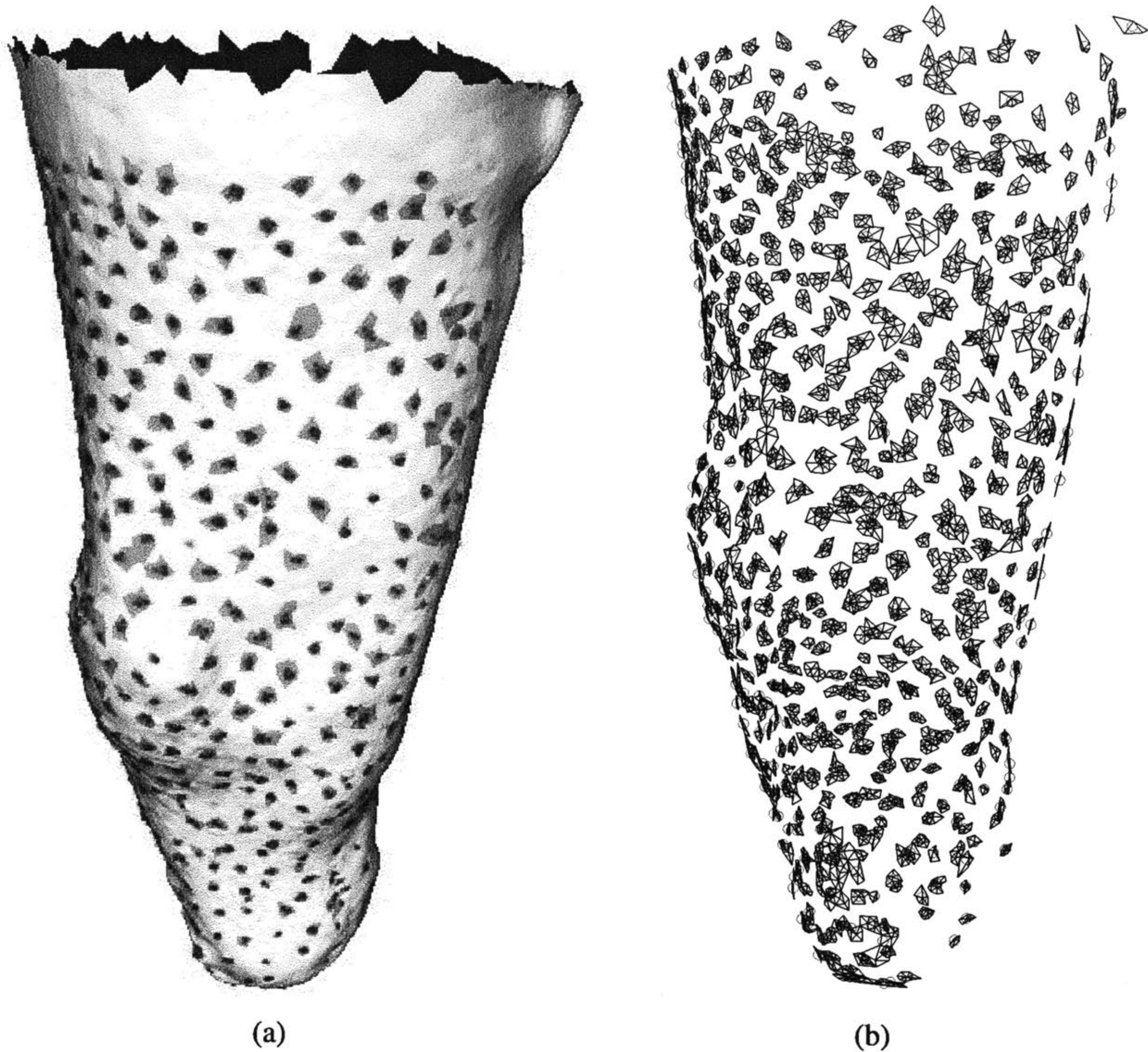


Figure 4.2: Texture images from each step of the dot filtering process. Image (a) shows the original texture, the median filter has been performed in image (b), image (c) shows the result of subtracting image (a) from image (b), and image (d) is the inverse of image (c).



(a) (b)

Figure 4.3: Image (a) shows the filtered texture wrapped on the surface of the model geometry with the faces corresponding to the dot locations automatically highlighted in red. Image (b) shows the extracted face clusters and their respective centroids denoted by red circles.

The sparse triangulated surfaces are imported in Matlab for subsequent processing. The *findpts* function loads the filtered, segmented .obj file and identifies the various triangulated patches, which correspond to the marker dots on the surface of the body. These patches of connected triangle clusters are found by a vertex walking algorithm. This algorithm picks a triangular face and identifies all of the faces that have exactly two vertices in common with it. The same two vertex matching search is executed on these neighbors until no additional triangles fitting the criterion are identified. The program iterates this search for the rest of the faces,

tabulating a list of faces that are in each cluster of triangular surfaces. The geometric center of each triangle group is computed using the mean of the vertex coordinates of the specified faces. Image (b) of figure 4.3 shows the result of these computations. This completes the point identification process which outputs a point cloud of tracked markers for each body pose.

4.3 Correspondence

With the markings on the surface of the body now identified and isolated, the correspondence problem between body poses must be solved. The point clouds from each body must be ordered and use the same triangulation for the strain metrics to be computed properly. The various strain functions require vertex coordinates from corresponding triangles as inputs, so the correct correspondence between points and triangles between body configurations is critical to the accuracy of the computations. The correspondence problem deals with how to provide automatic feature matching between two images or data sets. The two data sets can be visual or geometric, but in all situations there is a change in viewing angle, a deformation, or the passage of time that makes the scenes different. The correspondence problem is actually solved by the feature matching algorithms of the structure from motion techniques. Programs such as 123D Catch correspond features between images to derive 3D information to construct the model.

The correspondence problem is a significant barrier to automatic data processing as the program must figure out the relationships between the points in the clouds from two separate scenes. Due to this difficulty, the correspondence problem is often solved partially or completely by manual feature matching from user input. A method of automatically computing correspondences between two disordered point clouds is presented and used in this strain analysis.

The correspondence problem specific to this strain analysis concerns the matching of points between the 3D point clouds from two body poses. One pose is labeled as the original pose and the other, the deformed pose. The point order and triangulation of the deformed pose must be matched to those of the original pose. This is done through a number of computational methods, including constitutive relations, geometric compatibility, and probability. Another significant characteristic of this particular correspondence problem is the non-rigid deformation of the mesh coupled with significant translations and rotations. This is a natural consequence of drastically changing joint posture. There are clearly large translations and rotations of the points

on the skin as well as non-zero skin strains as it wraps over the surface of the body. The program must therefore allow every point to be transformed in an affine manner. There exists no program that can solve the correspondence problem for a points on a 3D crust that experiences non-rigid deformation and large translations. The *correspondence* Matlab script uses all of these techniques to solve the correspondence problem for the skin marking locations in a non-rigid, large deformation situation.

The program first imports both candidate point clouds and assigns a label to each vertex. If the script is successful, corresponding points will have the same numerical label in the end. The surface of the original pose is triangulated using a crust algorithm. This algorithm reconstructs three dimensional surface of arbitrary topology from unordered points. It uses both the Delaunay and Voronoi computations to identify the crust. The Voronoi diagram shown in figure 4.4 partitions a surface into convex polygons, each containing exactly one feature point [15]. The edges of the polygons are located at the midpoint of the distance between the nearest neighbors of the feature points. Every boundary is equidistant to the two nearest points and the intersections of these edges are equidistant to three points. The Delaunay triangulation also depicted in Figure 4.4 is the unique triangulation in which every circumcircle of a triangle does not enclose a feature point [14]. The circumcircle of a triangle is a circle that intersects all three of its vertices. This property means that all of the Delaunay triangles have maximized their minimum angle given the constraints of the finite data set.

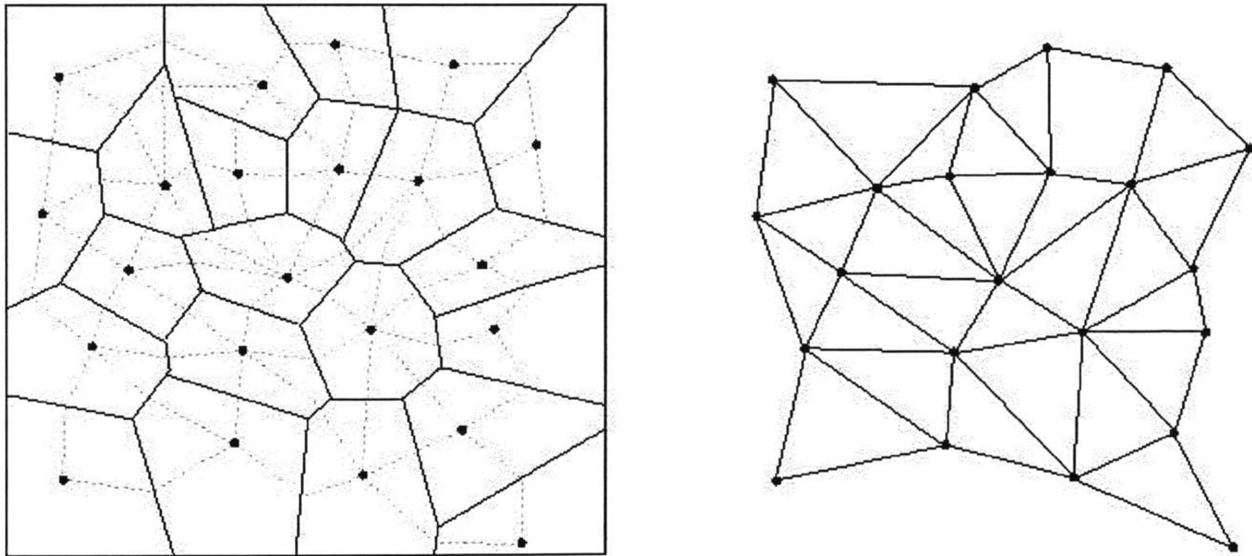


Figure 4.4: Voronoi diagram (left) and Delaunay triangulation (right) of a sample data set [14, 15].

The crust algorithm computes the Voronoi diagram of the sample points, which yields a set of Voronoi vertices. Subsequently, the Delaunay triangulation of the combined data set of sample points and Voronoi vertices is computed. The crust surface is defined by the edges of the Delaunay triangulation whose end points are both in the sample point set [3, 6]. This method can successfully triangulate any closed or open surface as long as there are no sharp edges. It works very well for surface information from the human body as it is smooth and differentiable. The crust algorithm is often accepted as the fastest and most reliable surface triangulation method. A crust script written by Luigi Giaccari is used in this strain analysis software. Figure 4.5 shows the crust of a human knee from data points generated using the photogrammetric methods described in Section 4.1.

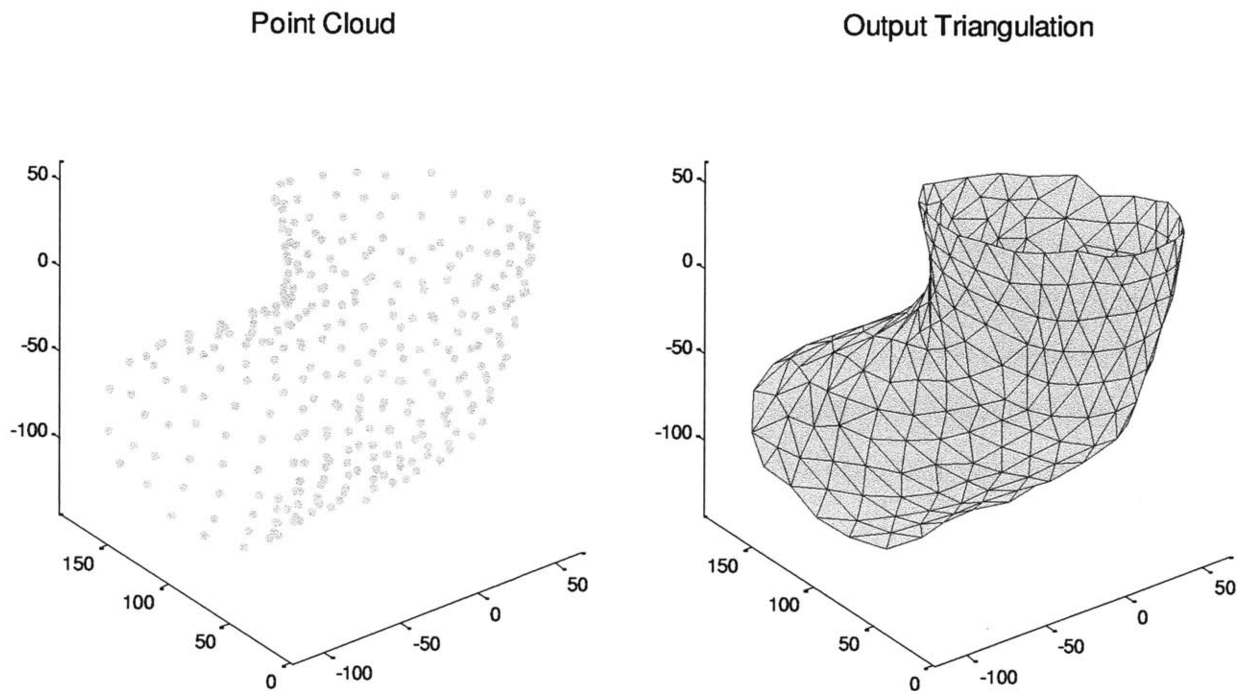


Figure 4.5: Point cloud (left) and corresponding crust triangulation (right) of a human knee.

Once the original surface is triangulated, the feature matching can begin. The *correspondence* script plots both point clouds and the original triangulation and provides a basic user interface for point selection, manual correspondences, and automatic correspondence solving. The program currently requires at least one manual correspondence to begin the automatic solver, but this can easily be eliminated in future versions by using a different marking on the body that can be distinguished by color, shape, size, or any combination of these

characteristics. Ideally this starting point is in a fairly regular, smooth region of the body so the algorithm can have a solid starting point. For example, a point on the thigh is a better starting point than one on the patella, as the thigh experiences more uniform and predictable skin strain than the patella. Starting from the known correspondence, the program uses a number of methods to walk through the data set and assign correspondences.

The first step in corresponding the point cloud from the deformed posture is performing the crust triangulation of this data set. The crust algorithm again produces the 3D surface reconstruction of the second pose, but this triangulation cannot be assumed to be the same as that of the original pose. The movement of the skin on the body may be such that the triangulation of the marks on the skin changes as a function of joint posture. Therefore, the crust of the deformed configuration is taken as a starting point for the correspondence algorithm.

4.3.1 Iterative Closest Point Algorithm

One of the automatic solving techniques is a local non-rigid point matching algorithm. This script takes in a known correspondence for investigation as well as both point clouds and triangulations and a list of all other known correspondences. All of this information is used to assign correspondences to the adjacent vertices of the target correspondence in both poses.

First, the adjacent vertices of the target correspondence in both triangulations are identified. These are vertices that are connected by an edge to the reference point as defined by the respective triangulation. Figure 4.6 shows the original and deformed configurations of a human knee as it undergoes a change in joint posture. Point 49 is the manual correspondence and the surrounding points are the corresponded adjacent vertices. The labels are the same as the correspondence has already been correctly computed. Previous to this computation, the point labels in the deformed configuration were all different than those in the original pose. If the number of adjacent vertices in both cases is not equivalent, this means that there is either a triangulation error or an incorrect correspondence. Both of these errors are addressed and corrected in subsequent routines. The affine transformation algorithm requires the same number of input and output points. If there is agreement between the local adjacent vertex sets, the affine transformation script can be executed on the local sets of points, usually around seven points for each pose in the case of a roughly equilateral triangle grid.

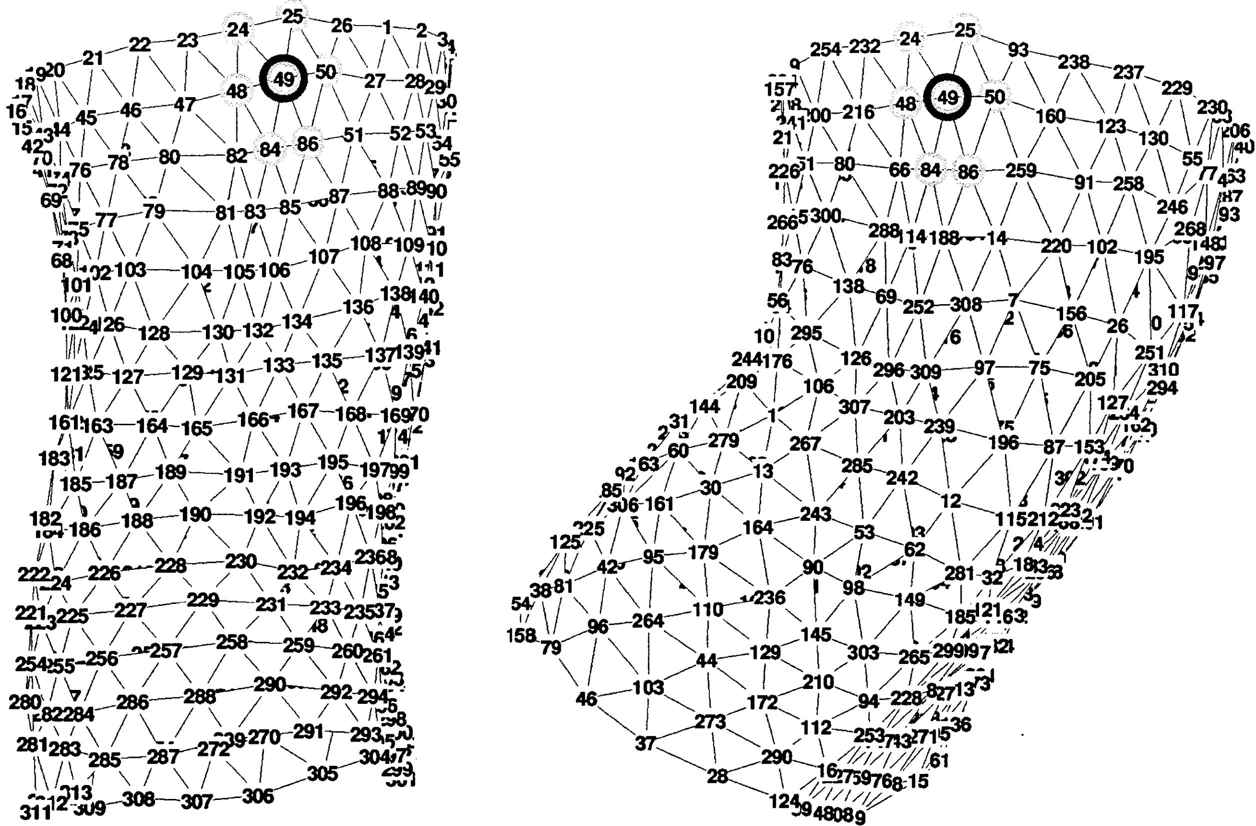


Figure 4.6: Computed local correspondences shown in green and manual correspondence shown in red. The left diagram is the original pose and the right is the deformed pose of a human knee.

Before correspondence is attempted, the deformed local point set is rigidly transformed to improve the likelihood that the program will make the correct correspondences. The known center point of each local group offers important position information. Every point in the deformed group is translated in three dimensions so the center points are coincident. Orientation errors that prevent proper matching may still exist. For example, for a roughly equilateral set of seven points, consisting of a center point and six connected neighbors, a rotation by modulus of sixty degrees can cause incorrect correspondences. This is remedied by utilizing other information about the point sets. If one of the points other than the center correspondence have been previously corresponded and are in the list of confirmed correspondences, the 3D information with respect to this point can be used to further orient the point sets. To accomplish this transformation, the rays connecting the center point to the second known correspondence are computed for each local point set. The rotation vector in angle-axis form is calculated for the

rotation from the deformed point set's ray to the original point set's ray. The corresponding rotation matrix is constructed using this angle-axis information. Finally, the deformed point set is multiplied by this rotation matrix to better orient the points with the original set. Again, these transformations serve to better align the data sets for the ease of matching points through the use of previous knowledge.

With the data set orientation now improved, the non-rigid correspondence algorithm can be executed. This algorithm is based on the iterative closest point method (ICP) that minimizes the mean of the squared distance between two point clouds. An affine implementation of this method iteratively changes the rotation, translation, and stretch matrices to minimize the distance error. This particular script was written by Ajmal Saeed Mian from the University of Western Australia. The algorithm works quite well for the small, local point sets as they tend to experience uniform translation, rotation, and strain. If the total error is below a certain margin and all of the points are corresponded, the script retriangulates the deformed cloud and reorders its points based on the new information. It also adds the newly corresponded vertices to the list of correspondences.

In this manner, the program walks through the deformed point cloud, computing the non-rigid correspondences and expanding the search from one solved correspondence to another. Once this search method is exhausted, other checks are performed before returning to this main correspondence algorithm. In particular, geometric compatibility and triangulation adjustments are performed automatically which allow the ICP program to further expand its search.

4.3.2 Geometric Compatibility and Triangulation Correction

As aforementioned, it cannot be assumed that the deformed point cloud's triangulation is the same as that of the original posture. The ICP non-rigid correspondence algorithm requires the same triangulation to be successful, so an automatic correction routine is needed to relabel the points in the deformed cloud and change the triangulation when necessary.

While the affine correspondence script ensures the constitutive relations between the points, the geometric compatibility adjustments enforce continuity. Geometric compatibility checks that the resulting configuration of points and triangular edges form a surface that is smooth and differentiable. One such check scans the known correspondences and verifies that their adjacent vertices and the neighbors of those vertices are in agreement. If there is any

discrepancy, the vertex is investigated further to identify the error as due to point labeling or incorrect triangulation.

The first check in this correction algorithm corrects a triangulation error in which there are three known, corresponded vertices for a triangle in the original cloud and only two corresponded vertices in the candidate triangle of the deformed cloud. A combination of checks of adjacent vertices, nearby solved correspondences, and shared vertices verifies that the area of interest is the same region in both triangulations. Figure 4.7 shows the targeted triangulation error of this check. The magenta circles indicate confirmed correspondences, while the green ones represent predicted correspondences. After this is confirmed, the triangulation is changed to agree with that of the original point cloud.

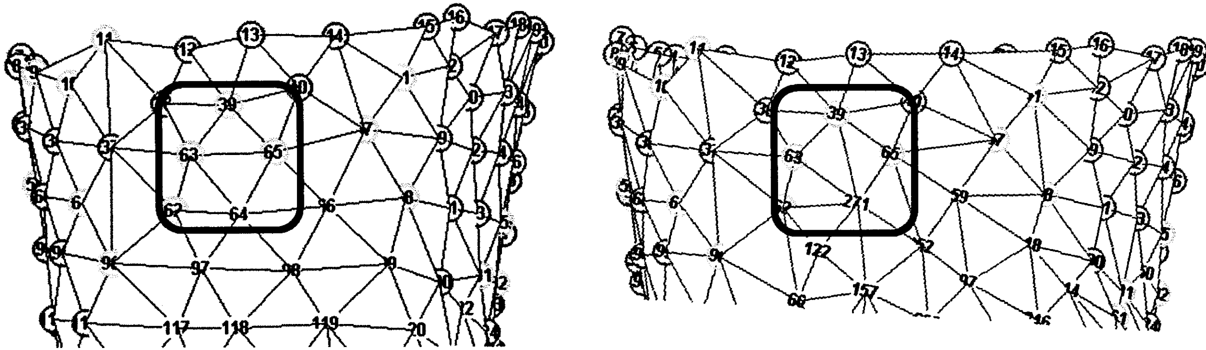


Figure 4.7: Correspondence problem partially solved between an original mesh (left) and a deformed mesh (right). Note the triangulation discrepancy highlighted by the box.

Another compatibility check corrects a triangulation error where there are two corresponded vertices in the first cloud and three in the second. This is fairly similar to the previous case and this script again remedies the problem. Other subroutines determine whether a correspondence discrepancy is due to incorrect labeling and remove them accordingly. The final result is the completed correspondence problem for the deformed cloud, shown in figure 4.8.

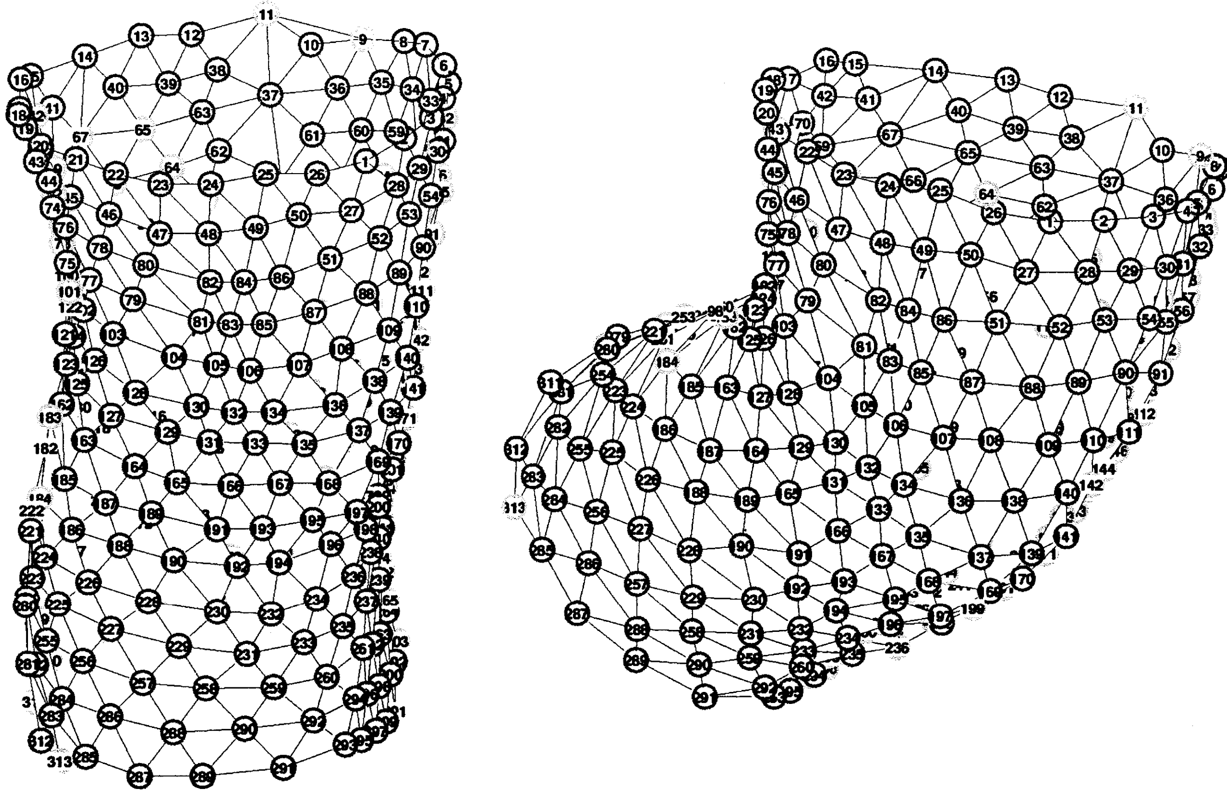


Figure 4.8: Completed correspondence problem for photogrammetric data processed with the automatic analysis software.

4.4 Strain Computations

With the point clouds from the assorted body poses now corresponded, a number of numerical strain methods can be executed to analyze the behavior of the surface. The computation of the various strain measures is described in section 2. In summary, a constant strain triangle analysis is employed to assess the equivalent strain, the strain field, the principal strain field, and the zero strain field. All of these methods are transparent as they were written specifically for this strain analysis toolbox.

The equivalent strain plot color codes the triangulated surface of the body according to the equivalent strain of each element. Mapping the overall strain of each triangle to a scalar value is useful for display purposes, providing a general view of how the skin behaves on the body as its posture is changed. Figure 4.9 shows the equivalent strain plot of a human knee undergoing partial flexion. While there is approximately zero strain along much of the surface, significant non-zero strains occur on the patella and behind the knee.

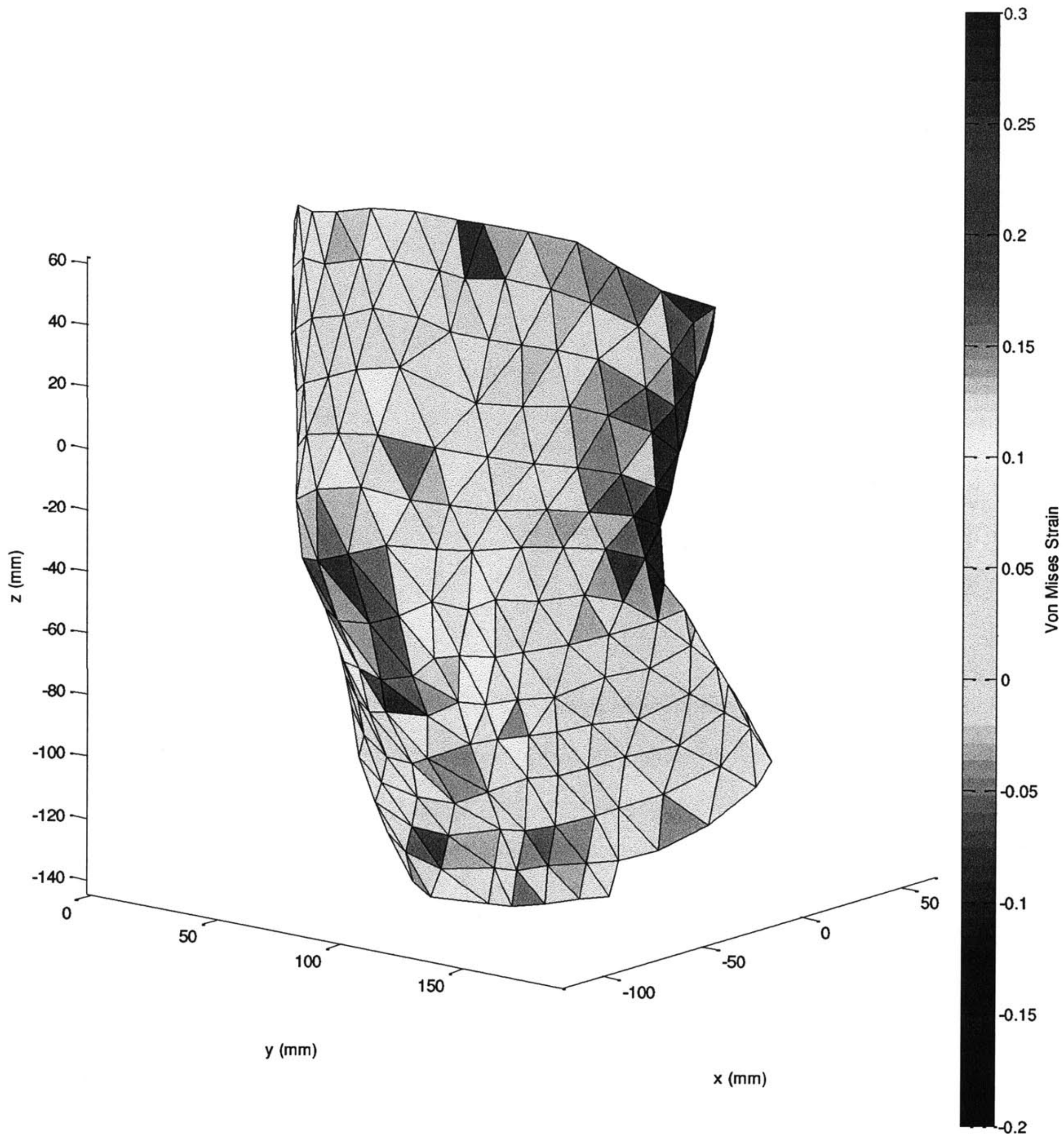


Figure 4.9: Equivalent strain plot of a human knee in partial flexion. Note the areas of high positive strain at the patella.

At the patella there are higher average strains due to the stretching of the skin over the joint during flexion. The plot shows averages ranging from 10% to 25% in this region. It can be predicted that negative strains should occur at the back of the knee. Figure 4.10 verifies this claim, indicating a range of negative strains from -5% to -15% at the back of the knee. These

plots are useful for describing the average strain of each skin element on a one dimensional scale.

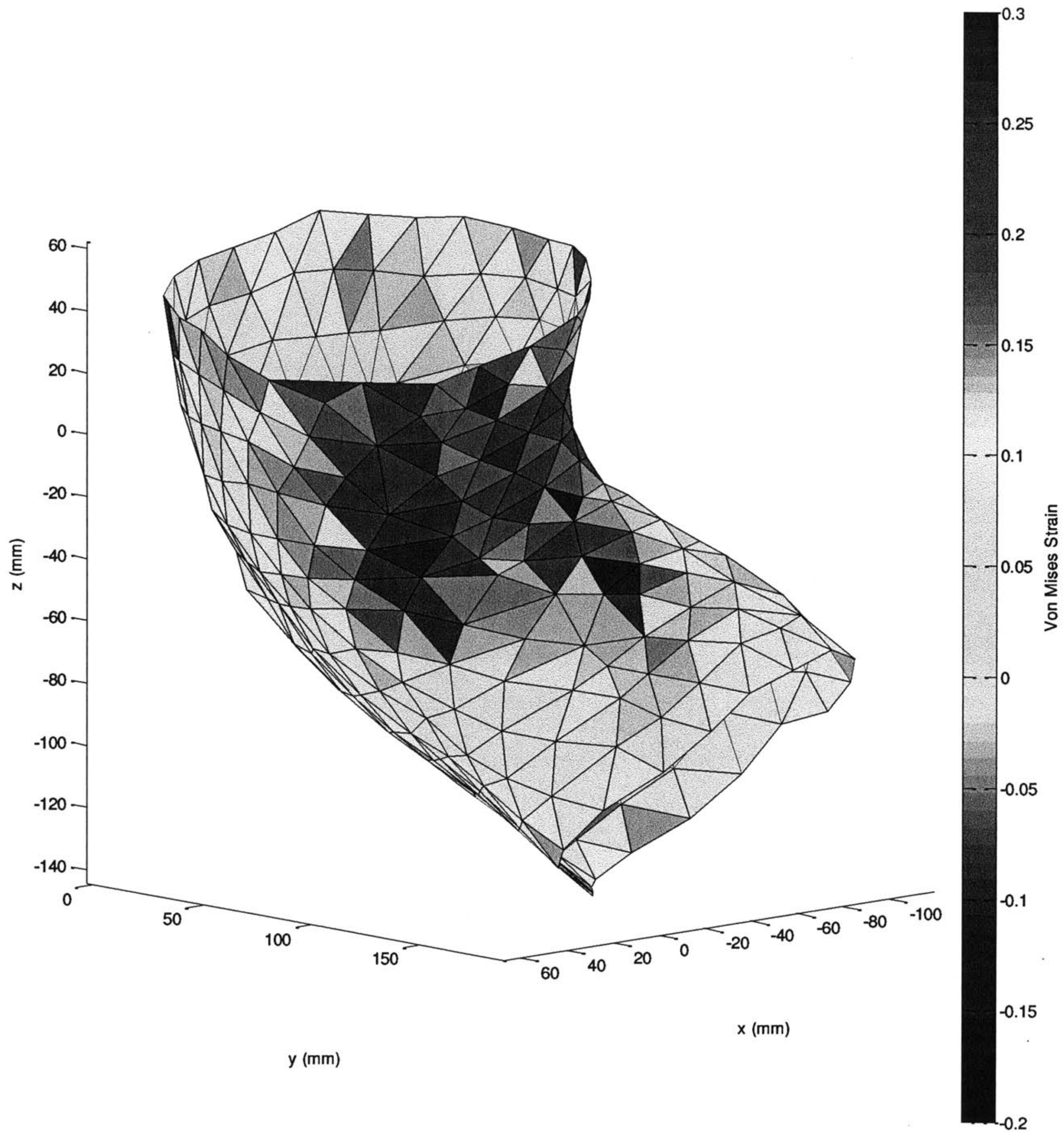


Figure 4.10: Posterior view of a human knee in a flexed posture. Note the areas of negative strain at the back of the knee.

The strain field provides a more complete depiction of how the skin deforms. Every element contains two normal strain vectors that show the stretch of the two dimensional element. The level of shear strain is indicated by the angle between the normal vectors. A right angle corresponds with zero shear and decreasing angles correlate with increasing shear strain. Figure 4.11 shows the strain field of a human knee in flexion. The strains exhibited here are largely dominated by normal rather than shear strain as the elements tend to dilate rather than change their angle.

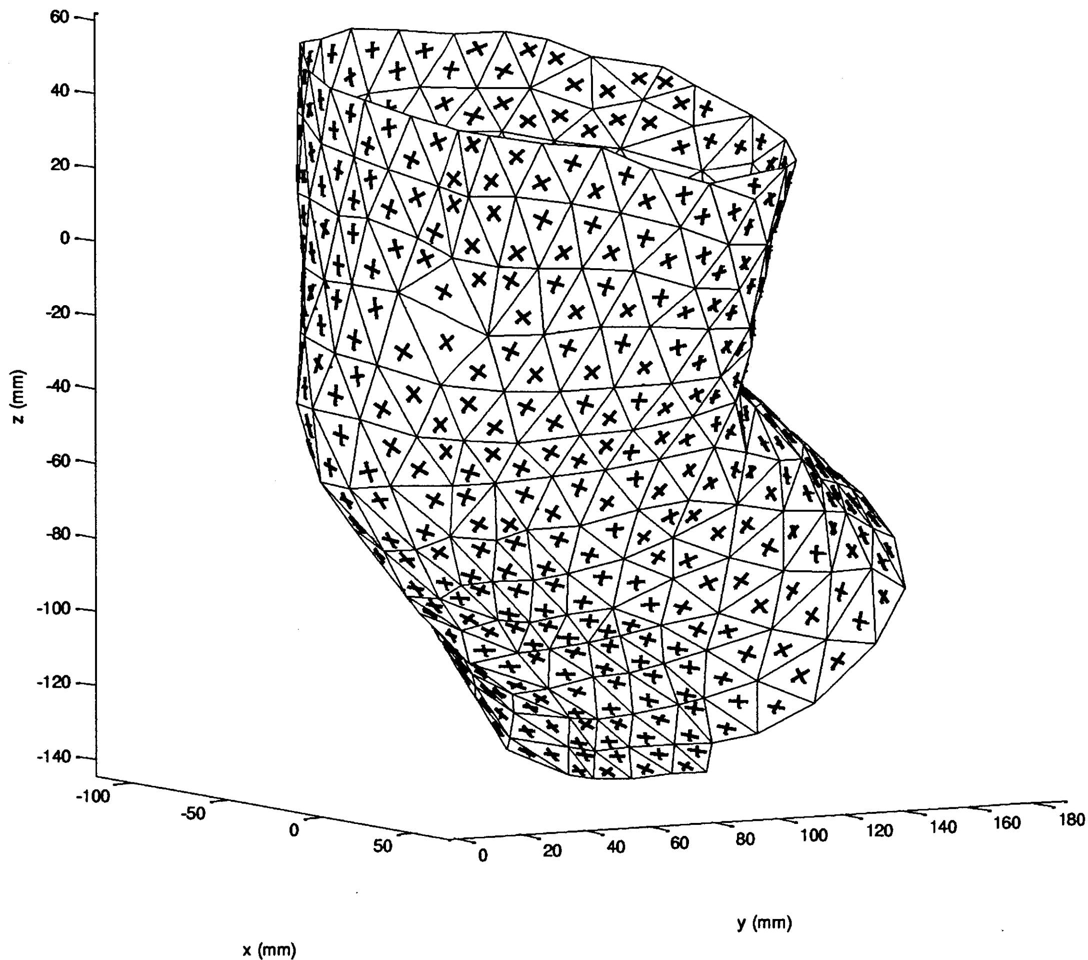


Figure 4.11: Strain field of a human knee during flexion. The red vectors indicate the larger of the two strains and the blue represents the smaller strain. Shear strain is represented by the angle between the two vectors.

The principal strain field shows the maximum and minimum normal strains of each triangular element. In the principal frame of each element no shear strains exist. Figure 4.12 shows the principal strain field of the human knee data set. This plot confirms the intuition behind how skin stretches over the body as posture is varied. At the patella and just proximal to the patella, the maximum strain is longitudinal. Here, the longitudinal direction is along the long axis of the femur. The skin must stretch over the joint in the flexed configuration, depicted by detail A in figure 4.12. The skin stretch is a minimum in the latitudinal directions in this region as the skin is not required to span any greater distance in this direction. The same arguments apply to the behavior of the skin on the muscle bellies. In this region, maximum skin stretch occurs in the circumferential direction, accommodating the expansion of the muscles during knee flexion. Above the knee the contractions of the hamstring and quadriceps serve to stretch the skin circumferentially, indicated by the latitudinal maximum strain vectors above the knee. Similarly, the contraction of the gastrocnemius muscle below the knee joint dilates the skin circumferentially as shown by detail B in figure 4.12. The principal strain field provides a clear view of how the skin stretches in two dimensions.

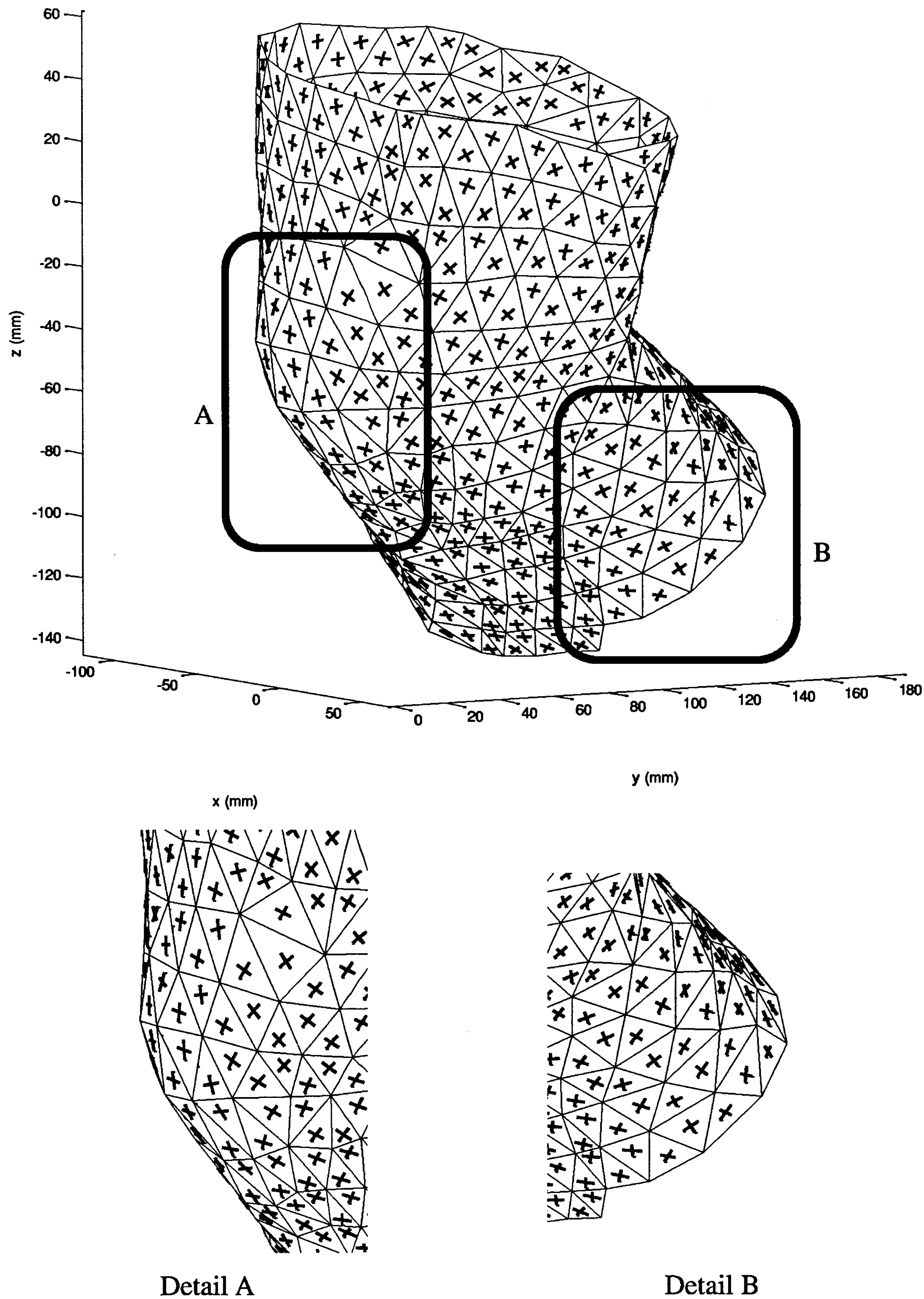


Figure 4.12: Principal strain field of a flexed human knee. Note how the larger (red) strain field is nearly horizontal proximal and distal to the knee joint as the skin stretches circumferentially (Detail B). Further, the larger strains are highly longitudinal at the knee patella as the skin stretches over the knee during flexion (Detail A).

The zero strain field is also computed by this strain analysis toolbox. Figures 4.13 and 4.14 show two views of the zero strain field for the case of a human knee joint. Not all of the elements have zero strain vectors as the skin in these areas either expands in both principal direction or contracts in both. Overall, most areas experience a combination of positive and negative strains as the skin deforms and settles at a minimum strain energy configuration as joint posture is changed. The pairs of zero strain, or non extension directions for each element lie circumferentially at the back of the knee in figure 4.13.

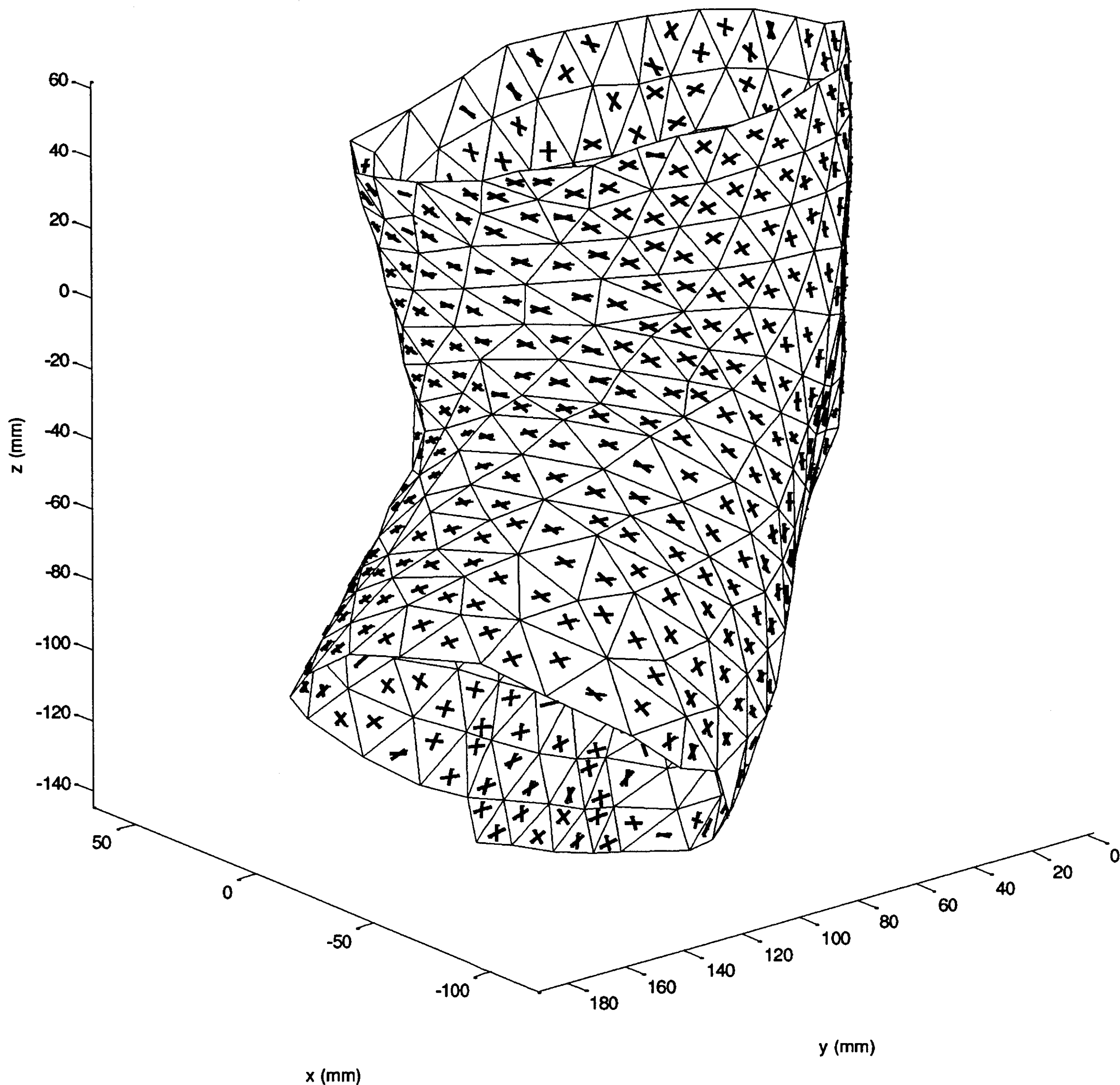


Figure 4.13: Posterior view of the zero strain field of a flexed human knee.

The relatively small angles of the vector pairs with respect to one another indicate the ratio of the maximum strain to the minimum strain is large. This means that the directions of non extension are angled just above and below the circumferential maximum strain vectors. On the anterior surface of the knee shown in figure 4.14, the angles tend to be larger as the principal strains are more similar in magnitude. The curves of non extension tend to open up on the anterior surface, allowing more uniform dilation of the skin.

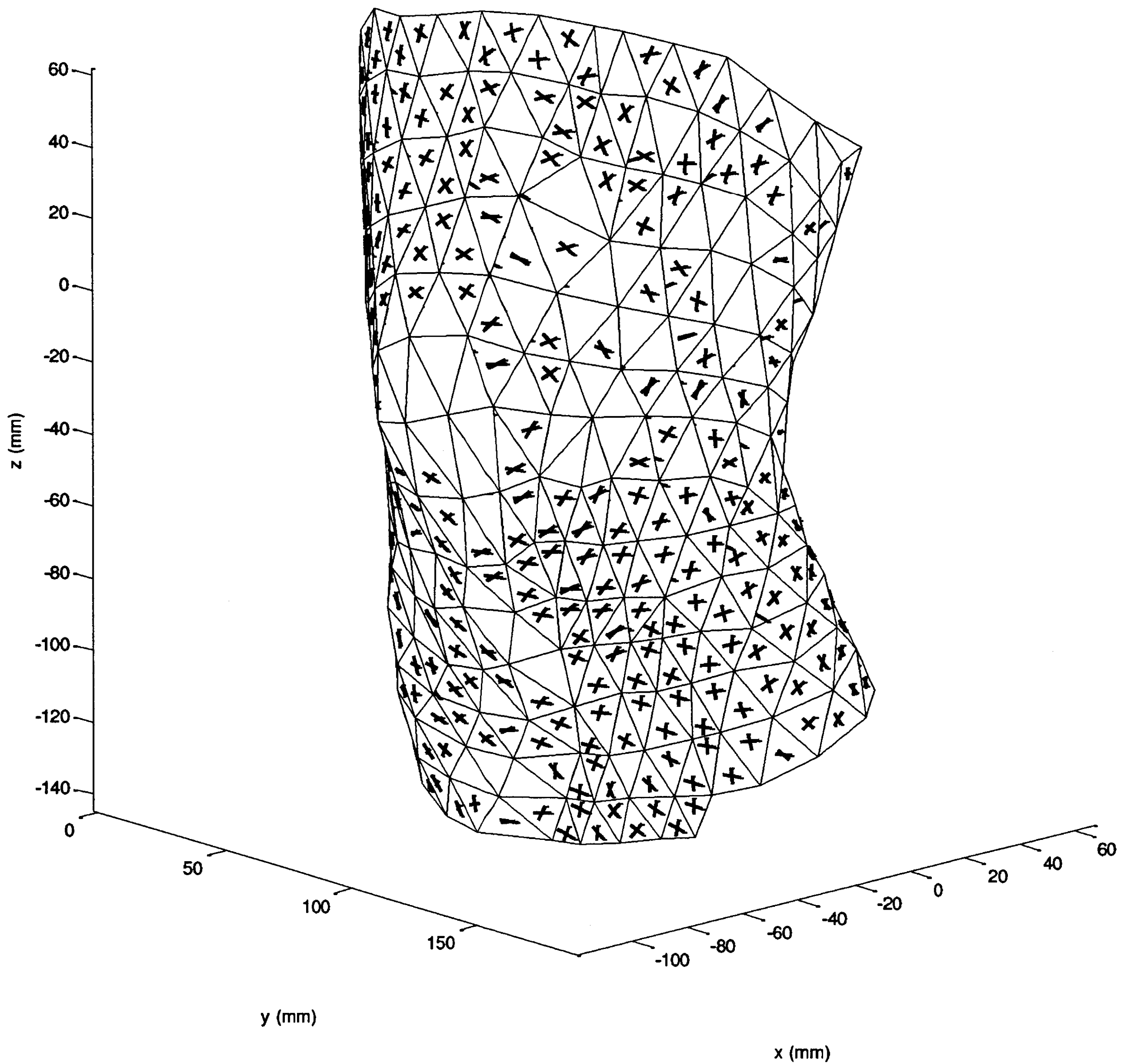


Figure 4.14: Anterior view of the zero strain field of a flexed human knee.

5. Results

In addition to the human knee test case described throughout the experimental methodology section, several other studies confirmed the validity of these methods and further explored the mechanical behavior of human skin in vivo.

5.1 Validation

Two validation tests prove the accuracy of the photogrammetric methods and the strain software. The first uses a piece of graph paper to model a zero strain case. Whether the paper is flat or curled, the strain along the surface should be approximately zero. The second test case analyzes the strain behavior of a section of latex tubing. Latex exhibits linear, isotopic, elastic mechanical behavior and has a Poisson ratio of approximately 0.5. Since its strain response to a step change in length is regular and predictable, the accuracy of all the methods in the toolbox can be assessed.

5.1.1 Zero Strain Test Case

The zero strain case is simulated using a piece of graph paper with a dot pattern. If a flat piece of paper is curled into any smooth shape, its surface should experience no strain. The triangular surface elements will translate and rotate in space but not change size. Figure 5.1 shows the two paper configurations in this study. A flat reference pose is used along with a bell curve shape held in position by a couple weights.

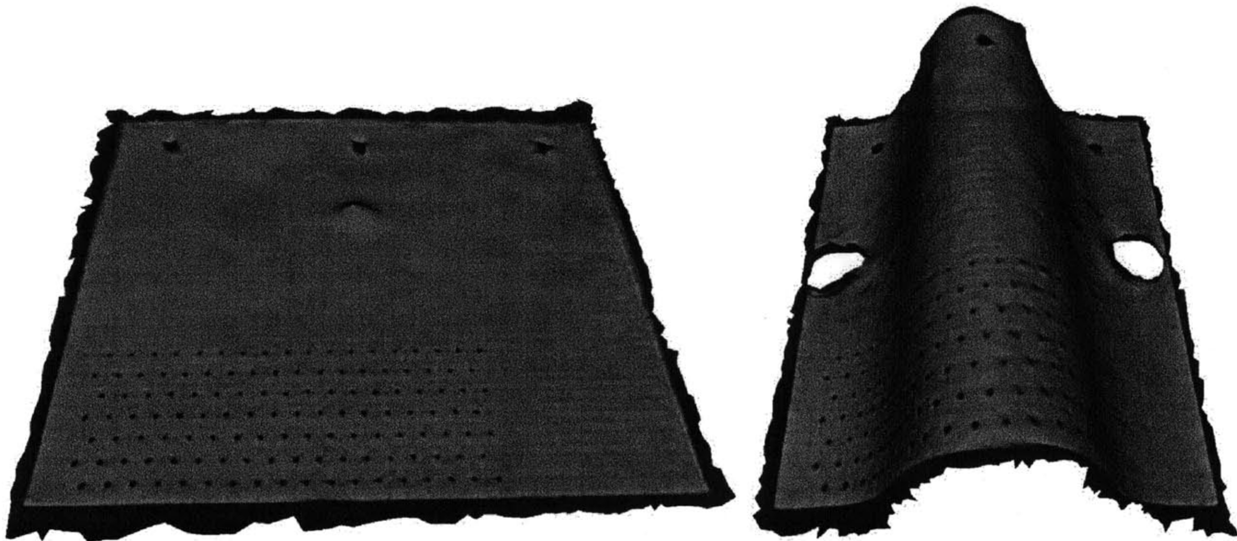


Figure 5.1: 3D photogrammetric models of the two graph paper configurations.

The mean equivalent strain of the elements on the graph paper was 1.0% with a standard deviation of 2.0%. Figure 5.2 shows the visualization of the equivalent strain of the deformed graph paper triangulation. This is consistent with the predicted average strain of 0%. The small errors are likely due to the nature of the photogrammetric 3D reconstruction process. Attempting to correspond, triangulate, and construct a 3D model from 2D images inherently involves an error minimizing optimization to produce the most accurate model.

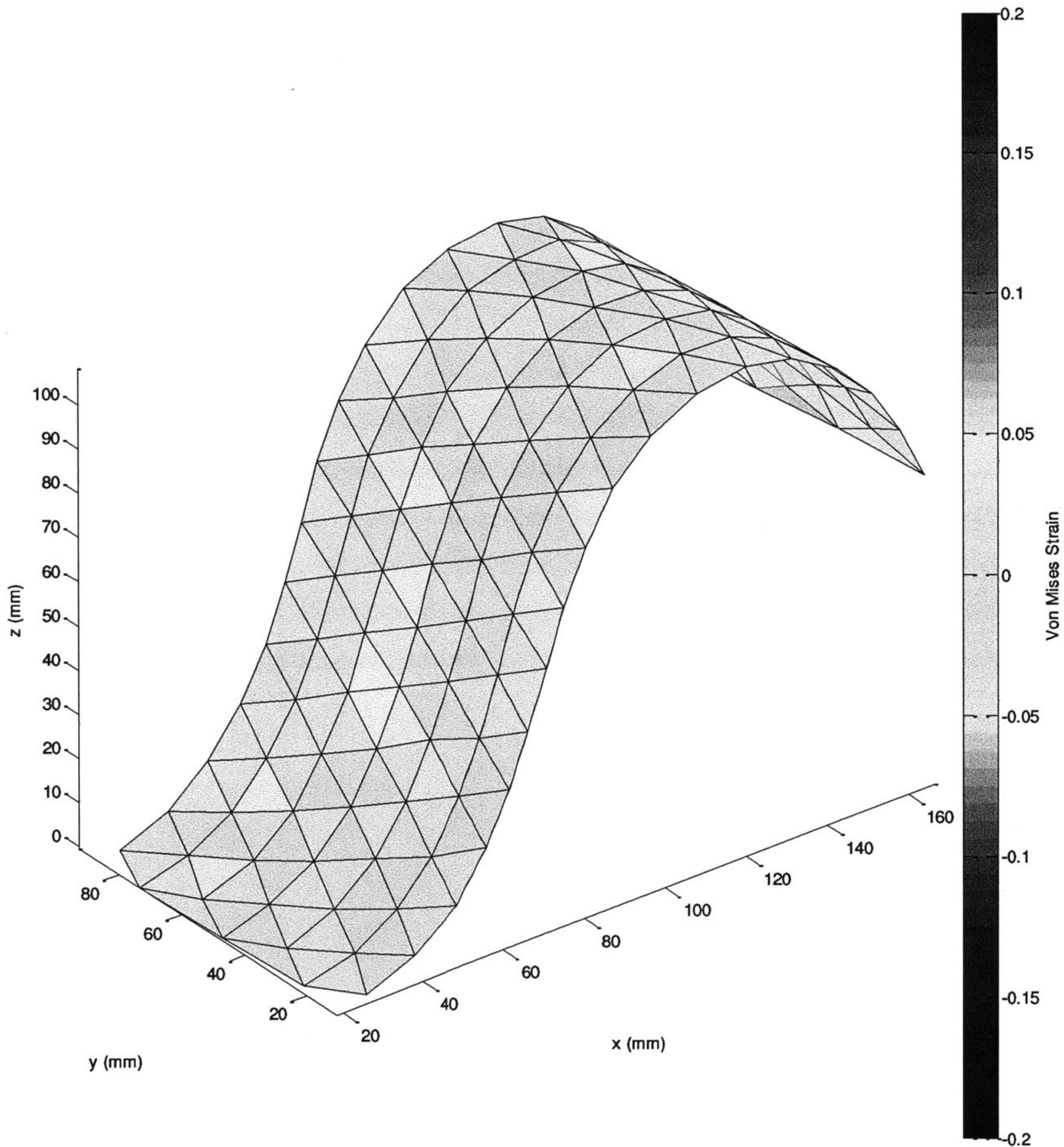


Figure 5.2: Equivalent strain plot of the deformed graph paper test case.

The strain field is also consistent with the predictions, showing nearly zero strain. Figure 5.3 shows the strain field for this test case. Here, the strain vectors are scaled with respect to one another to have an average length that is approximately half of the median triangle side length for visibility. The mean of the maximum principal strains is 3.4% with a standard deviation of 2.6%. The mean of the minimum principal strains is -1.8% with a standard deviation of 2.8%. The strains are close to zero, again demonstrating the accuracy of this system.

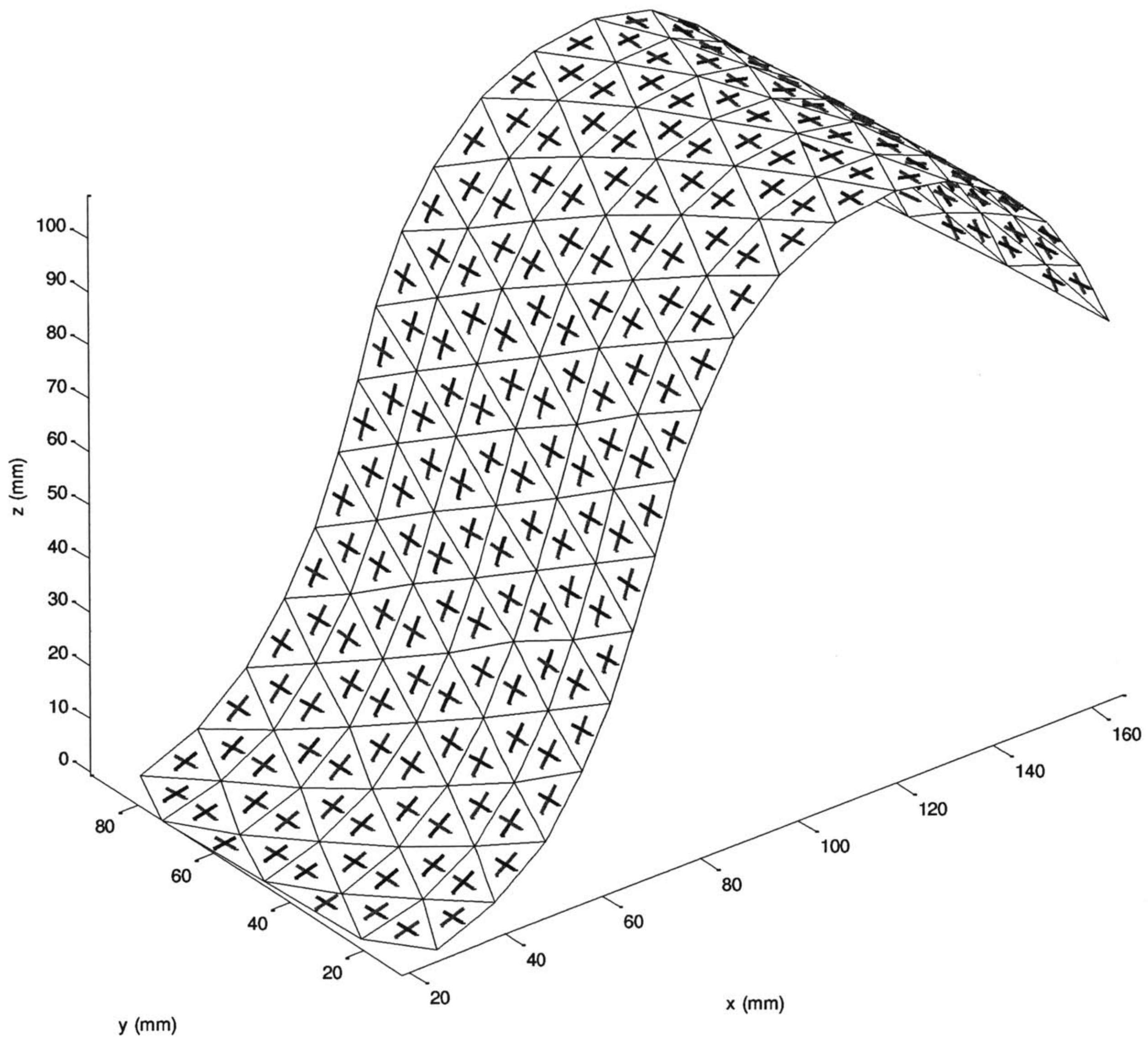


Figure 5.3: Strain field of the deformed graph paper. Note how the strain field vectors are approximately the same length and lie along the x and y axes to which the paper is also aligned.

5.1.2 Uniaxial Tension Test Case

The second test case of this automatic strain computation software analyzed the strain behavior of latex tubing. Latex is a linear, isotropic, elastic material that behaves like a rubber material with a Poisson ratio of 0.5. Photogrammetric models were constructed for the tubing in its relaxed state and in a state of constant uniaxial tension. Figure 5.4 shows the two reconstructed models of the latex tubing.

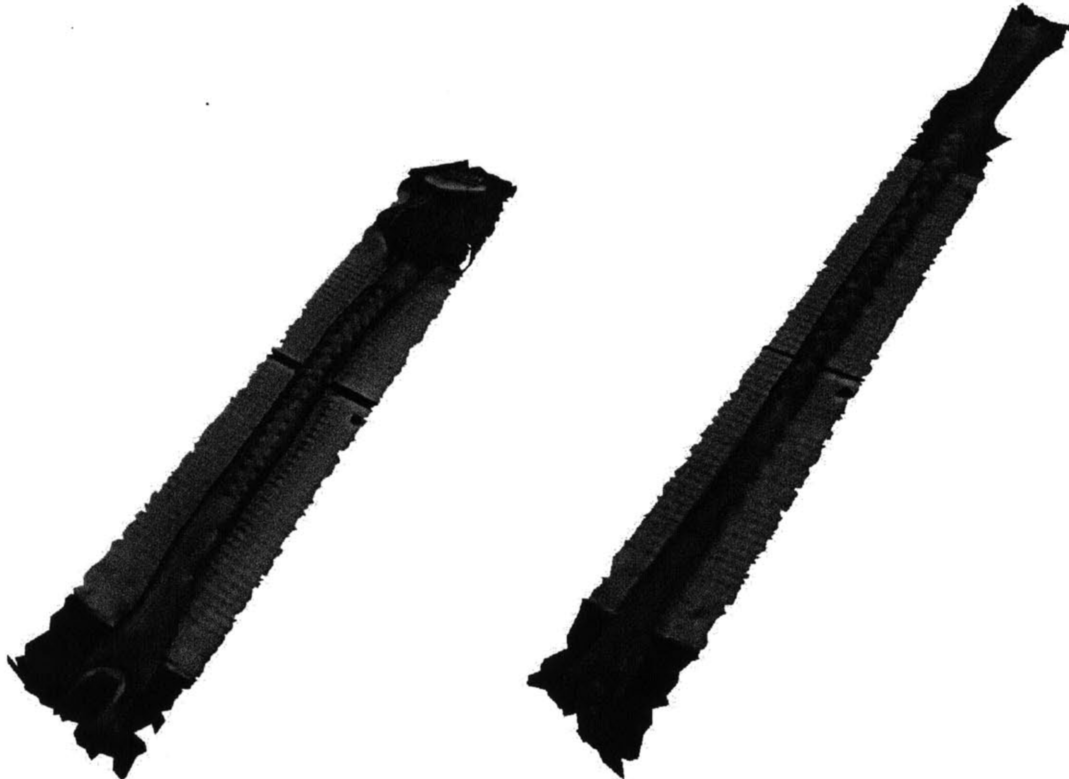


Figure 5.4: 3D photogrammetric models of the relaxed (left) and stretched (right) configurations.

The mean of the finite element equivalent strains was 46.3% with a standard deviation of 5.3%. Figure 5.5 depicts the equivalent strains present during this uniaxial tension test. While there appears to be uniform stretch throughout the latex, the strain field indicates whether the results are accurate. The finite strain calculations show a mean maximum strain of 68.6% with a standard deviation of 5.9%. The mean minimum strain is -23.3% with a standard deviation of 3.8%. Figure 5.6 displays the principal strain field for the stretched tubing. These experimental

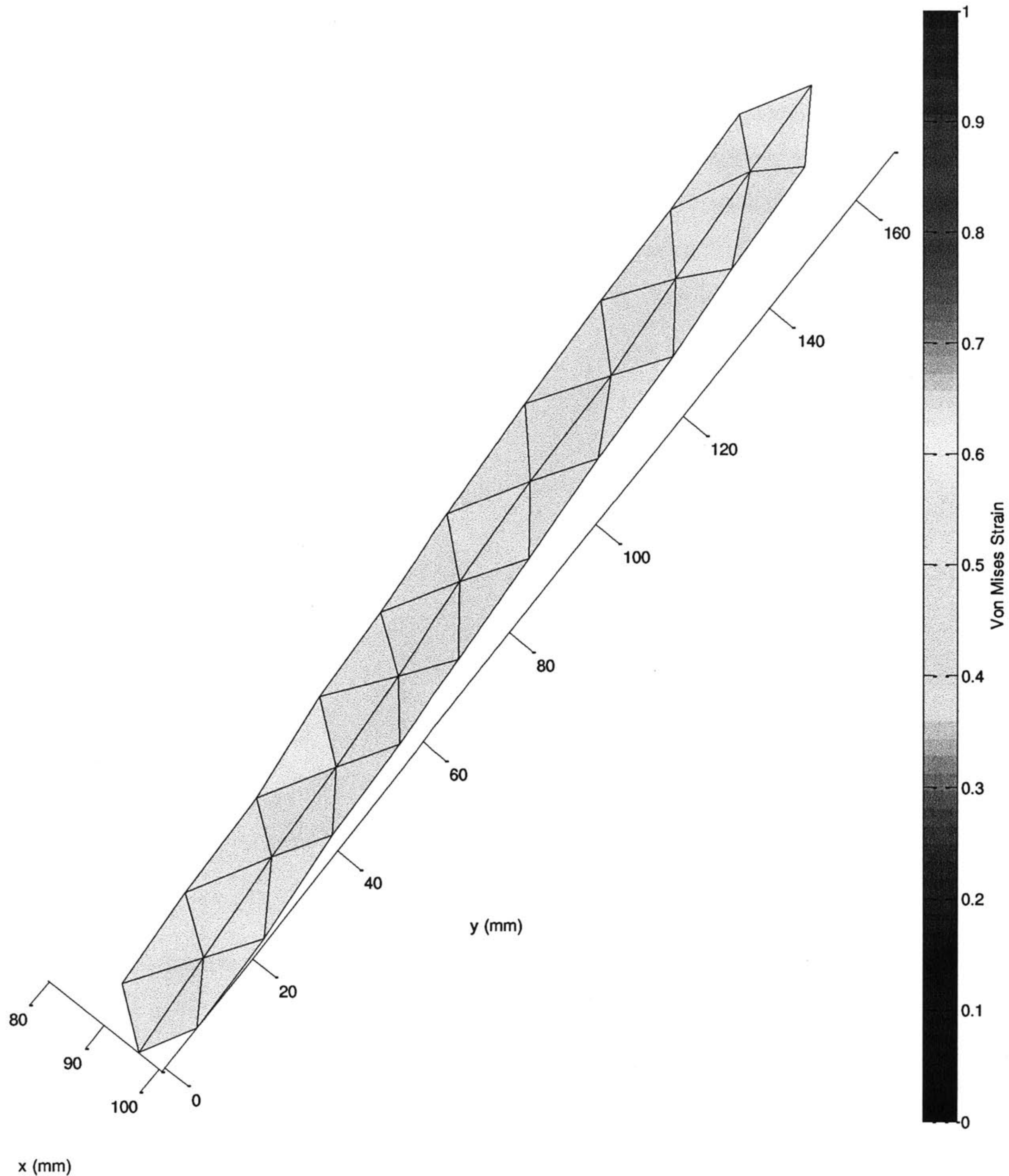


Figure 5.5: Equivalent strain plot of the latex tubing in uniaxial tension.

values must be compared to the expected strain behavior of latex tubing to assess this method's validity. A uniaxial tension test generated a force-deflection plot for the latex tubing using five data points. The elastic modulus was determined to be 903 MPa with an R^2 value of 0.997. In

the stretched state shown in Figure 5.6, the tensile force was 306 N, which corresponds to a stress of 1.53 MPa. Using Hooke's law in terms of the elastic modulus and Poisson's ratio, the ideal strain behavior of the latex tubing was determined. The expected axial strain for this loading situation was 69.4% and the associate lateral strain was -15.3%. While the actual and theoretical axial strains agreed closely, there is some discrepancy between the lateral values. This is due to the geometrical errors of the coarse triangulated surface that simplifies the curved surface into a set of faces.

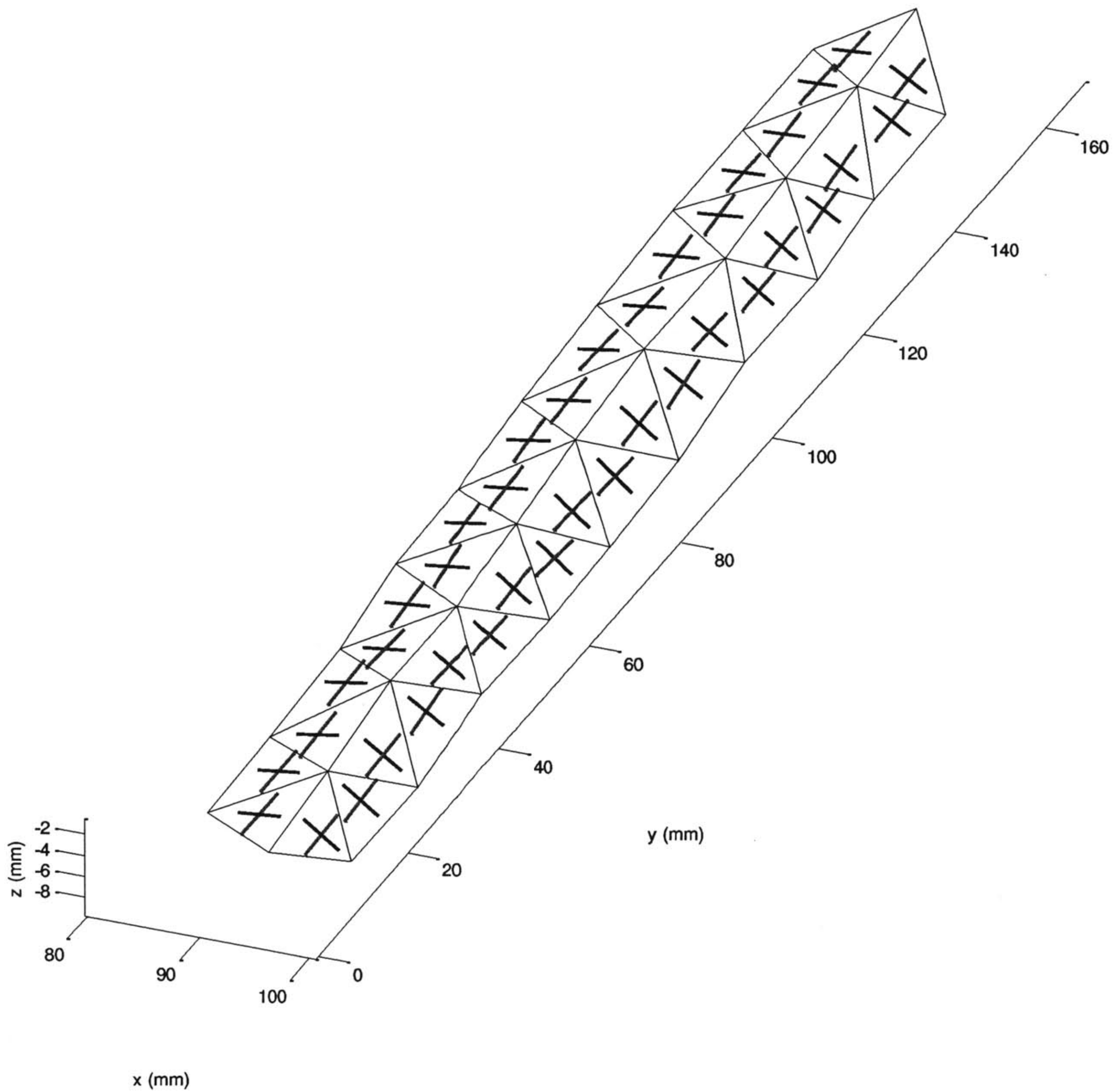


Figure 5.6: Principal strain field of the latex tubing in uniaxial tension.

5.2 Transtibial Amputation Study

In addition to the knee study described in previous sections, the knee of a transtibial amputee was also analyzed. A higher resolution dot matrix was used in this case at approximately 4 dots per cm^2. Three poses were captured for this study, with the knee fully extended, flexed to approximately 45 degrees, and flexed further to approximately 90 degrees. Figure 5.7 shows the reconstructed 3D models of the three poses and figure 5.8 shows their triangulations.

Figure 5.7: Three poses of a transtibial residual limb are shown each corresponding to a particular knee flexion angle.

The progressive increase in the equivalent strain of the surface elements is visible on the patella as the knee flexes in figure 5.9. In the partially flexed pose, the peak equivalent strain at the patella is approximately 20%, while it peaks at over 40% in the 90 degree posture. Strain fields similar to the full bodied knee study are generated here as well. Figure 5.10 shows the strain field for the 90 degree knee flexion, which confirms the claims made in the full bodied knee case. This strain information may prove to be useful in changing how we design prosthetic liners and sockets. The implications of this research are discussed in the subsequent future work section.

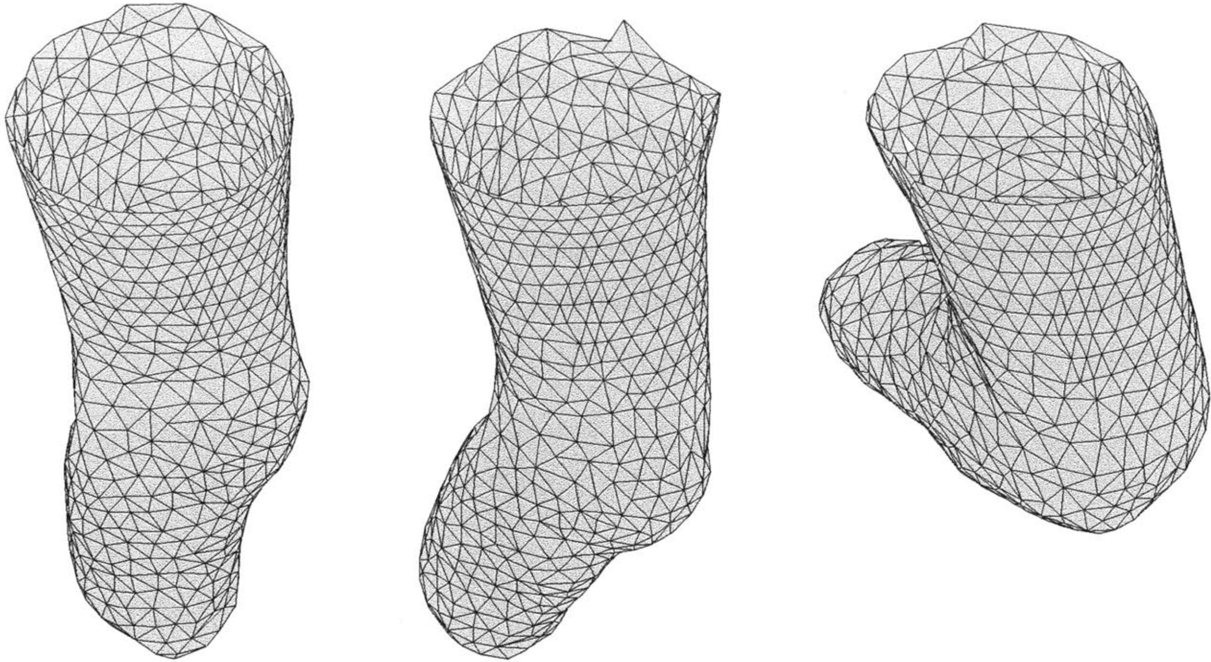


Figure 5.8: The coordinate information from three triangulated poses of a transtibial residual limb are used to compute the strain transforms.

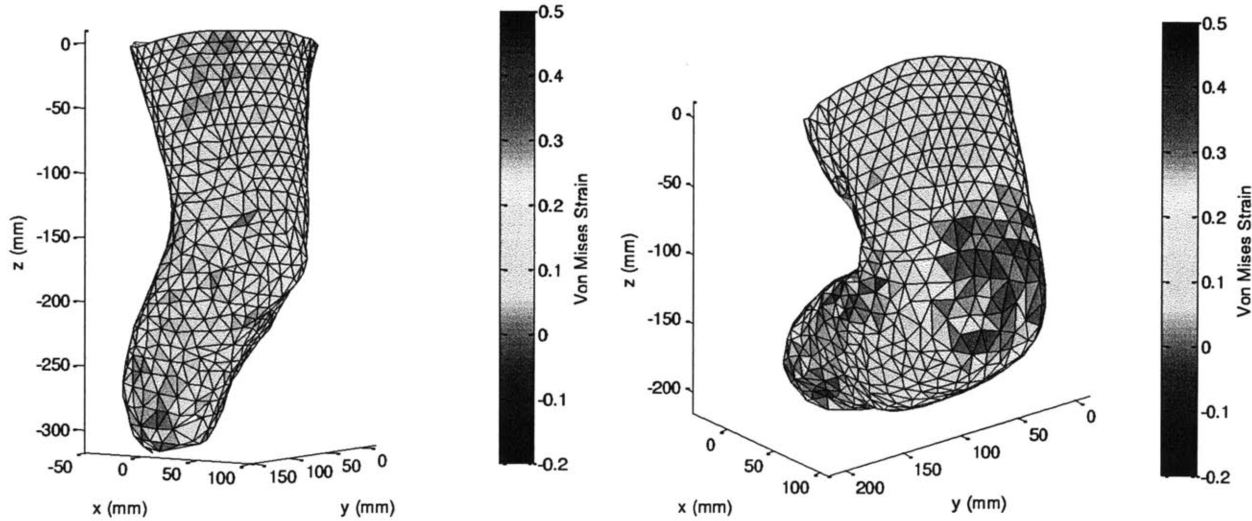
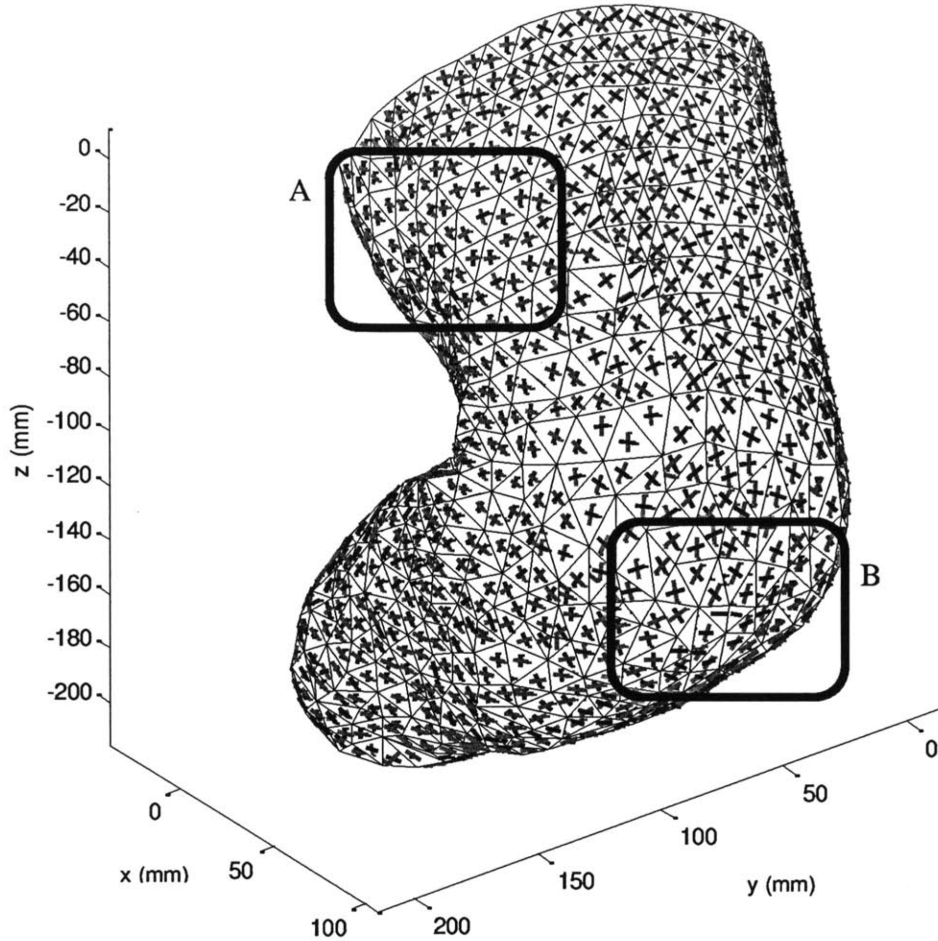


Figure 5.9: The average strain of each triangular face is analyzed and mapped to a color. Skin strain levels are shown for the partially flexed pose (left plot) and the fully flexed pose (right plot). Here higher average strain is shown around the knee patella due to the right pose's increased knee flexion.



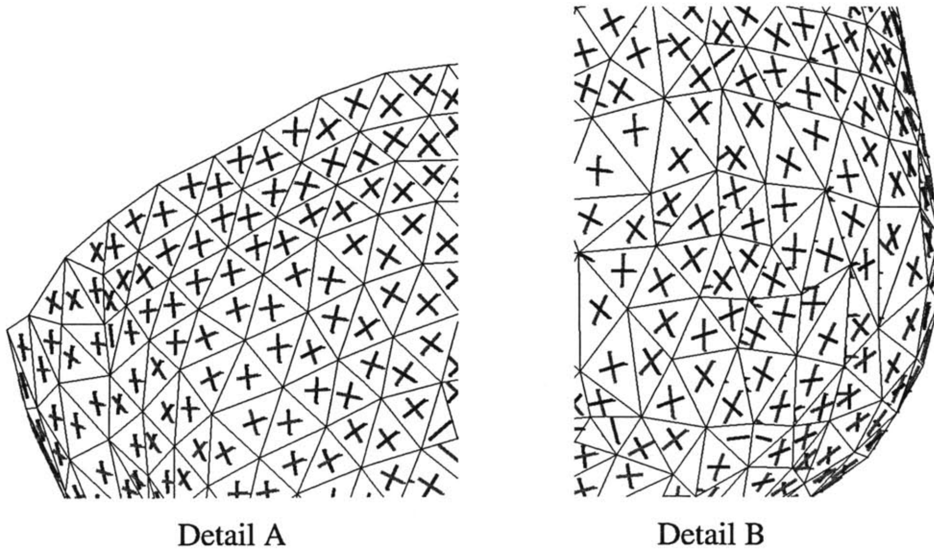


Figure 5.10: The strain field of the knee flexed to approximately 90 degrees. Note how the circumferential strains are dominant at muscle bellies (detail A) and longitudinal strains are greater at the patella (detail B). Here, the longitudinal direction is along the long axis of the femur.

6. Conclusions

An automatic method of assessing human skin strain is presented here which uses photogrammetric methods to understand how skin on the body deforms as joint posture is varied. The computational transparency and flexibility of this pipeline is one of its key strengths. The photogrammetric model is reconstructed from simple digital photographs, allowing the investigator to choose both the scope and level of detail of the model. For the case of analyzing skin strain, one can freely adjust the dot resolution and the range of body parts captured for reconstruction. The dots may be placed in closer proximity to one another in high interest areas, such as on muscle bellies or joints. The software completes the rest of the analysis procedure, identifying markers, solving the correspondence problem, and computing the various strain measures with a toolbox based on the fundamental principles of linear, isotropic, elastic strain mechanics. The methods proposed here require significantly less user input and offer increased flexibility compared to previous skin strain analysis procedures.

6.1 Applications

The flexibility of this toolbox means that its use should not be limited to analyzing human skin strain behaviors. It can be used for the analysis of the crust of any objects, especially non-rigid bodies that experience large deformations. The primary application of this software is the analysis of skin strain for the design of prosthetic and exoskeletal interfaces. Additional potential uses include the design of more form fitting clothing and improving the realism of computer graphics and animations of the human body, especially the face.

The primary application of this toolbox is to gather information to design a more comfortable interface for prostheses and exoskeletons. Skin deformation is one critical aspect of mechanical interface design, which directly affects the comfort of the device. Using conventional prosthetic socket technology, an amputee typically wears a liner that is rolled across the residual limb. By making the coefficient of static friction high between the skin and the liner materials, designers have effectively lowered the relative movement at that interface, reducing uncomfortable rubbing and chaffing. However, current liner technology only utilizes a single, isotropic material at the skin interface. This is insufficient to reproduce the wide range of mechanical strain behaviors that human skin exhibits, as informed by the previous sections. For example, in the case of a transtibial amputation, inflexibility in the liner in high strain regions,

such as on the patella and proximal knee areas, cause skin discomfort. This discomfort increases significantly when an amputee sits with knees flexed for an extended period of time.

To achieve the goal of creating a more comfortable liner, we propose design that minimizes total shear stress on the skin as the biological segment changes posture. The mechanical strain energy stored in the liner should be minimized when the biological limb is moved to a pose with large skin strains. A purely mathematical approach using the strain field can predict the desired mechanical properties of the interface at every location. By continuously adjusting the tensile viscoelastic properties of the material spatially across the surface, a dynamic liner that responds properly to changes in posture can be created. The numerical relationship could be linear or nonlinear depending on the type of mechanical interface, the region of the body for which an interface is to be constructed, and the specific needs of the user. One possible embodiment that may be adequate to mirror the behavior of skin uses a simple inverse proportional relationship between strain and tensile stiffness. In this model, directions of maximum strain should have a lower stiffness, while directions of lower strain should have higher effective liner stiffness.

Another possible mapping scheme determines the stiffness of a particular finite element by its equivalent strain relative to the rest of the elements. This one dimensional configuration provides the elasticity required by each element to stretch as much as the skin below it. Smoothing the stiffness transitions between adjacent elements may also provide a more continuous and natural mechanical behavior.

There is an opportunity to use the zero strain information to complement the design of the second skin liner. The zero strain field map indicates curves along which there is zero strain, or non-extension. Flexible bands of high elastic modulus material may be placed in these regions to restrict the area's extension in the desired direction but still allow shape changes in other directions.

This stiffness can be a function of any number of properties, including material selection and geometry. Tensile stiffness is proportional to elastic modulus and cross sectional area. Varying the material properties of a surface is possible by bonding adjacent materials together to create a patchwork where the proper materials are used locally to deform like the skin. One emerging technology is multi-material 3D printing. Objet Geometries is a pioneer in the 3D printing of variable stiffness and durometer materials [12]. The Connex family of printers has a

digital materials mode in which the two materials in the printer's reservoir can be combined in different ratios to produce ABS like rubber materials with a wide range of mechanical properties. The printed part can contain an entire continuum of durometers and elastic moduli in any conceivable form. Adjusting the geometry of the surface is another way of affecting its stiffness. Changing the cross sectional area will directly affect tensile stiffness. The easily configurable dimension of a liner is its thickness. For a given liner material, its thickness at a given finite element can be inversely proportional to strain. Thinner areas allow for greater deformation, while thicker ones are stiffer and deform less under similar loads. Varying both the material and its thickness provides an even wider range of effective stiffnesses for the liner.

All of these methods can be combined to create a new second skin interface for the attachment of prosthetic sockets and other external devices. Furthermore, this is a completely computational method of developing an interface that is tuned to the individual's body, a task usually reserved for professionals, such as prosthetists. Once a design cost function is selected, the process can be fully automated, similar to the rest of the strain analysis process.

Another application of this strain analysis software is the design of clothing, both for performance and comfort. For spandex performance clothing, the same approach as the prosthetic liner can be used. While spandex itself conforms to the body fairly well, as joint posture is changed, the material stretches and bunches up in different regions. Implementing the same stiffness mapping from skin strain can create even better fitting skin tight clothing.

This software can also provide design guidance for more stiff and inextensible materials. The skin strain map can show the level of stretch on the surface of the body throughout a range of joint postures. For any finite element, the maximum stretch throughout the range of motion can be identified. A program can generate a map of minimum material areas for every patch on the surface of the body. Therefore, for an inextensible cloth to wrap over the body without restricting its movement, every region of the clothing must have a surface area greater than or equal to the corresponding value on the minimum surface area map. This design rule can be used to create perfect fitting, custom clothing, such as denim jeans.

This strain analysis tool may also be useful for improving computer graphics animations of the human body, especially the face. Much of the realism of facial computer animations is

determined by how natural the skin movement is simulated. However, modeling and reproducing the biological movements of the body in a computer graphics program is extremely difficult. The intricate and coupled movements of bones, muscle, skin and other body tissues are not yet characterized in animation software. This optical analysis toolbox can catalog and elucidate how the skin on the face deforms depending on different facial feature postures. While it does not indicate the trajectories of bones or muscles, it does improve our understanding of the surface of the body, which is often the desired result for computer animations. This software could be useful in both the movie and video game industry to improve human skin animations.

6.2 Future Work

The next steps of this human skin strain study include modifying the procedure and code to process video to capture a continuum of movement, rather than static poses. The goal is to produce strain maps for an entire range of motion to identify how skin deforms. This would provide an understanding of the path of each finite element's deformation, in contrast to the current system that only shows deformation states for the photographed poses.
This can be accomplished through the use of several digital cameras placed on tripods around the targeted body part. Video captured at sixty frames per second or greater should provide adequate, clear views of the target. The videos need to be synchronized, either through the use of synchronized cameras or some other external visual or auditory cue. Selected frames can be now put through the analysis pipeline, from the photogrammetric reconstruction to the strain computations. If enough frames are processed, a video of the body's deformation can be constructed. Such outputs may include a video of the equivalent strain plot over time or the strain field over time as body poses changes all of which would improve our understanding of the biomechanics of movement.

Bibliograhy

[1] Arc 3D Webservice. http://homes.esat.kuleuven.be/~visit3d/webservice/v2/.

[2] Autodesk 123D Catch. http://www.123dapp.com/catch.

[3] Shitu Bala and Gianetan S. Sekhon. A Comparative Study of Surface Reconstruction Algorithms based on 3D Cloud Points Delaunay and Crust Triangulation. *International Journal of Computer Science and Technology*, 2(4): 327-329, Dec. 2011

[4] Kristen Bethke. The Second Skin Approach: Skin Strain Field Analysis and Mechanical Counter Pressure Prototyping for Advanced Spacesuit Design. MS Thesis, Massachusetts Institute of Technology, 2005.

[5] Bundler: Structure from Motion (SfM) for Unordered Image Collections. http://phototour.cs.washington.edu/bundler/.

[6] Satyan L. Devadoss and Joseph O'Rourke. *Discrete and Computational Geometry.* New Jersey: Princeton University Press, 2011.

[7] Russell C. Hibbler. *Mechanics of Materials.* New Jersey: Pearson Prentice Hall, 2005.

[8] Arthur S. Iberall. The Experimental Design of a Mobile Pressure Suit. *Journal of Basic Engineering*, June 1970

[9] Arthur S. Iberall. The Use of Lines of Nonextension to Improve Mobility in Full-Pressure Suits. Rand Development Corporation, November 1964.

[10] Alan Liu. *Mechanics and Mechanisms of Fracture*. Ohio: ASM International, 2005.

[11] Sara S. P. Marreiros. Skin Strain Field Analysis of the Human Ankle Joint. MS Thesis, University of Lisbon, 2010.

[12] Objet Geometries. http://www.objet.com.

[13] Microsoft Photosynth. http://photosynth.net.

[14] Eric W. Weisstein. "Delaunay Triangulation." MathWorld- A Wolfram Web Resource. http://mathworld.wolfram.com/DelaunayTriangulation.html

[15] Eric W. Weisstein. "Voronoi Diagram." MathWorld- A Wolfram Web Resource. http://mathworld.wolfram.com/VoronoiDiagram.html

List of Appendices

Appendix A: Point Detection and Tracking 60
Appendix B: Correspondence Problem 62
Appendix C: Strain Computations and Plotting 82

Appendix A: Point Detection and Tracking

image_filtering.m

```
function image_filtering(file_in, file_out, dot_res)
% performs two dimensional median filtering to make the dots in the texture
% image more visible.  takes in an image file and an approximate dot
% resolution in pixels. returns the filtered image and saves to the
% file_out location.
close all
image_color = imread(file);

image = image_color;
image2(:,:,1) = medfilt2(image(:,:,1), [dot_res dot_res]);
image2(:,:,2) = medfilt2(image(:,:,2),[dot_res dot_res]);
image2(:,:,3) = medfilt2(image(:,:,3),[dot_res dot_res]);

image3 = image2-image;
image3_inv = 255-image3;
figure
imshow(image)
figure
imshow(image2)
figure
imshow(image3)
figure
imshow(image3_inv)
imwrite(image3_inv, file_out, 'jpg');
```

findpts.m

```
function ptcloud = findpts(file)
% file is the location of the .obj file

obj1 = loadawobj(file); % load obj file
obj1v = obj1.v'; % get vertices
obj1f = obj1.f3'; % get faces

n = length(obj1f);
i = 1;
connectedfaces = cell(n,1);
g = 0; % group variable
while i<=n
    if sum(cellfun(@(x)ismember(i,x),connectedfaces))
        i = i+1;
    else
        g = g+1;
        list = i;
        connectedfaces{g}=findneighbors(obj1f,i,list);
        i=i+1;
    end;
end;

% find average centers of groups
connectedfaces(cellfun(@isempty,connectedfaces))=[];
ptcloud = ptcloudcenter(connectedfaces,obj1f,obj1v);

% plot results
figure(1);
hold on;
trisurf(obj1f,obj1v(:,1),obj1v(:,2),obj1v(:,3),'facecolor','none','edgecolor','black');
plot3(ptcloud(:,1),ptcloud(:,2),ptcloud(:,3),'ro');
```

findneighbors.m

```
function tri_list = findneighbors(faces,tri_index,list)
% finds all faces connected to one another by at least two vertices.
% effectively locates surface patches
tri_list = list;
n = length(faces);
    tri = faces(tri_index,:);
    binarymembers1 = ismember(faces,tri(1));
    binarymembers2 = ismember(faces,tri(2));
    binarymembers3 = ismember(faces,tri(3));
    binarymembers = binarymembers1+binarymembers2+binarymembers3;
    binarysums = sum(binarymembers,2); % sum rows
    for j = 1:n
        if binarysums(j) == 2
            if sum(ismember(tri_list,j))
            else
                tri_list = [tri_list, j];
                tri_list = findneighbors(faces,j,tri_list);
            end;
        end;
    end;
```

ptcloudcenter.m

```
function groupcenters = ptcloudcenter(connectedfaces,faces,vertices)
% locates the geometric center of a group of faces.  averages the vertex
% locations so certain shared vertices are intentionally counted more than
% once.
n = length(connectedfaces);
xc = [0];
yc = [0];
zc = [0];
sumx = 0;
sumy = 0;
sumz = 0;
for i=1:n
    facelist = connectedfaces{i};
    m = length(facelist);
    for j=1:m
        facenumber = facelist(j);
        vertexlist = faces(facenumber,:);
        v1 = vertices(vertexlist(1),:);
        v2 = vertices(vertexlist(2),:);
        v3 = vertices(vertexlist(3),:);
        sumx = sumx+v1(1)+v2(1)+v3(1);
        sumy = sumy+v1(2)+v2(2)+v3(2);
        sumz = sumz+v1(3)+v2(3)+v3(3);
    end;
    xc(i) = sumx/(3*m);
    yc(i) = sumy/(3*m);
    zc(i) = sumz/(3*m);
    sumx = 0;
    sumy = 0;
    sumz = 0;
end;
groupcenters = [xc' yc' zc'];
```

Appendix B: Correspondence Problem

correspondence.m

```
function [cloud1_list cloud2_list, c_matrix] = correspondence(cloud1, cloud2, tri1)
% solves correspondence problem for two point clouds- inputs are the point
% clouds and the triangulation of cloud1
close all;
set(0, 'DefaultFigureSelectionHighlight', 'off');
plot1 = figure(1);
set(gcf, 'Pointer', 'crosshair');
hold on;
title('Original Pose, Triangulated');
trisurf(tri1,cloud1(:,1),cloud1(:,2),cloud1(:,3),'facecolor','white','edgecolor','blue');

label1 = plot_ptlabels(cloud1(:,1),cloud1(:,2),cloud1(:,3));
axis equal;
plot2 = figure(2);
set(gcf, 'Pointer', 'crosshair');
hold on;
axis equal;
title('Deformed Pose, Attempted Triangulation');
trisurf(tri2,cloud2(:,1),cloud2(:,2),cloud2(:,3),'facecolor','white','edgecolor','blue');
label2 = plot_ptlabels(cloud2(:,1),cloud2(:,2),cloud2(:,3));

label_visible = 1;
done = 0;
target = [];
cp1 = [];
cp2 = [];
correspond_manual = [];
correspond_manual_save = [];
correspond_auto = 0;
correspond_auto_solved = [];
correspond_auto_errors = [];
cp_index = 1;
highlight_color = 1;
label1_index = 1;
label2_index = 1;
while done == 0
    resp1 = input('\nKeyboard Commands:\np to find most recently clicked point.\nc to correspond
last 2 points.\nr to recalculate corresponding triangulation.\na to autosolve correspondence.\nl
to toggle labels.\ns to save correspondence data.\nq to quit.\n','s');
    if strcmp(resp1,'p')==1
        cursor_pt = select3d;
        if ~isempty(cursor_pt)
            target = get(0,'CurrentFigure');
            if target==1 || target==2
                if target==1
                    [index1, point1] = closest_pt(cursor_pt, cloud1);
                    if ~isempty(index1)
                        cp1 = index1
                        point1
                        plot1;
                        label1_cpts(label1_index) = highlight_pt(point1);
                        label1_index = label1_index+1;
                    end;
                end;
                if target==2
                    [index2, point2] = closest_pt(cursor_pt, cloud2);
                    if ~isempty(index2)
                        cp2 = index2
                        point2
                        plot2;
                        label2_cpts(label2_index) = highlight_pt(point2);
                        label2_index = label2_index+1;
                    end;
```

```
            end;
        else
            fprintf('Error: Try to select a point again. \n');
        end;
    end;
end;
if strcmp(resp1,'c')==1
    if isempty(cp1) || isempty(cp2)
        fprintf('Error: Select a point from each plot. \n');
    else
        if isempty(correspond_manual)
            correspond_manual(cp_index,:)=[cp1, cp2]
            fprintf('Correspondence added\n');
            cp_index = cp_index+1;
            cp1 = [];
            cp2 = [];
        else
            dim_c = size(correspond_manual);
            n = dim_c(1);
            match1 = 0;
            match2 = 0;
            for i=1:n
                if correspond_manual(i,1)==cp1
                    match1=i;
                end;
                if correspond_manual(i,2)==cp2
                    match2=i;
                end;
            end;
            if match1==0 && match2==0
                correspond_manual(cp_index,:)=[cp1, cp2]
                fprintf('Correspondence added\n');
                cp_index = cp_index+1;
                cp1 = [];
                cp2 = [];
            end;
            if match1~=0 && match2~=0
                m1 = num2str(match1);
                m2 = num2str(match2);
                fprintf(['Previous correspondences with cloud 1 point ' m1 ' and cloud 2
point ' m2 ' found.\n']);
                resp2 = input('Confirm correspondence replacement (y/n)\n','s');
                if strcmp(resp2,'y')==1
                    correspond_manual(match1,:)=[cp1, cp2]
                    fprintf('Correspondence modified\n');
                    cp1 = []; cp2 = [];
                    if match1~=match2
                        correspond_manual(match2,:)=[]
                        cp_index = cp_index-1;
                        fprintf('Correspondence removed\n');
                    end;
                else fprintf('Correspondence operation cancelled\n');
                end;
            end;
            if match1~=0 && match2==0
                m1 = num2str(match1);
                fprintf(['Previous correspondence with cloud 1 point ' m1 ' found.\n']);
                resp3 = input('Confirm correspondence replacement (y/n)\n','s');
                if strcmp(resp3,'y')==1
                    correspond_manual(match1,:)=[cp1, cp2]
                    fprintf('Correspondence modified\n');
                    cp1 = []; cp2 = [];
                else fprintf('Correspondence operation cancelled\n');
                end;
            end;
            if match2~=0 && match1==0
                m2 = num2str(match2);
                fprintf(['Previous correspondence with cloud 2 point ' m2 ' found.\n']);
                resp4 = input('Confirm correspondence replacement (y/n)\n','s');
                if strcmp(resp4,'y')==1
                    correspond_manual(match2,:)=[cp1, cp2]
```

```
                    fprintf('Correspondence modified\n');
                    cp1 = []; cp2 = [];
                else fprintf('Correspondence operation cancelled\n');
                end;
            end;
        end;
        % Recolor selected control points
        figure(1);
        dim_c = size(correspond_manual);
        n = dim_c(1);
        for i=1:(label1_index-1)
            set(label1_cpts(:),'visible','off')
        end;
        highlight_color = 1;
        for i=1:n
            pt_index = correspond_manual(i,1);
            label1_cpts(i)=highlight_pt(cloud1(pt_index,:));
        end;
        label1_index = n+1;
        figure(2);
        highlight_color = 1;
        for i=1:(label2_index-1)
            set(label2_cpts(i),'visible','off')
        end;
        for i=1:n
            pt_index = correspond_manual(i,2);
            label2_cpts(i)=highlight_pt(cloud2(pt_index,:));
        end;
        label2_index = n+1;
    end;
end;
if strcmp(resp1,'r')==1
    if ~isempty(correspond_manual)
        fprintf('Calculating correspondences \n');
        dim_mc = size(correspond_manual);
        mc = dim_mc(1);
        for i=1:mc
            [tri2, newcloud2, corr_auto, stop] =
correspond_clouds(cloud1,cloud2,tri1,tri2,correspond_manual(i,:),correspond_auto);
            if stop==1
                % Rename single point alone
                newcloud2 = zeros(length(cloud2),3);
                newcloud2(correspond_manual(i,1),:)=cloud2(correspond_manual(i,2),:);
                index_rebuild = 1;
                cloud2_subset = del_tri(cloud2,correspond_manual(i,2));
                for p=1:length(cloud2)
                    if sum(ismember(correspond_manual(i,1),p))==0
                        newcloud2(p,:)=cloud2_subset(index_rebuild,:);
                        index_rebuild = index_rebuild+1;
                    end;
                end;
                if correspond_auto==0
                    correspond_auto = [];
                end;
                correspond_auto2 = [correspond_auto, correspond_manual(i,1)];
                correspond_auto = unique(correspond_auto2)
                fprintf('Error: Cannot create correspondence group. Single correspondence
created\n');
            else
                cloud2 = newcloud2;
                correspond_auto = corr_auto
                figure(2);
                if exist('label3','var')
                    set(label3,'visible','off')
                    clear label3;
                end;
                set(label2,'visible','off')
                clear label1 label2;
                label2 = plot_ptlabels(cloud2(:,1),cloud2(:,2),cloud2(:,3));
                autocoords2 = vertices2coords(cloud2,correspond_auto);
                label3 = highlight_autopts(autocoords2);
```

```
                    figure(1);
                    label1 = plot_ptlabels(cloud1(:,1),cloud1(:,2),cloud1(:,3));
                    autocoords1 = vertices2coords(cloud1,correspond_auto);
                    label1_3 = highlight_autopts(autocoords1);
                    fprintf('Replotting cloud 1 and cloud 2 labels\n');
                    correspond_manual(i,2)=correspond_manual(i,1);
                end;
            end;
        else
            fprintf('Error: Input manual correspondences first. \n');
        end;
    end;
    if strcmp(resp1,'a')==1
        fprintf('Attempting to autosolve correspondence\n');
        startpoint = correspond_manual(1,1);
        corr_done = 0;
        counter = 0;
        while corr_done == 0
            if length(correspond_auto)==length(cloud1)
                corr_done = 1;
            else
                if isempty(correspond_auto_solved)
                    correspond_auto_solved = startpoint;
                end;

                % Find unique unsolved points
                clear unsolved_auto;
                x_index = 1;
                x_counter=1;
                while x_counter <= length(correspond_auto)
                    if sum(ismember(correspond_auto_solved,correspond_auto(x_counter)))==0
                        unsolved_auto(x_index)=correspond_auto(x_counter);
                        x_index = x_index+1;
                    end;
                    x_counter = x_counter+1;
                end;

                if exist('unsolved_auto_updated', 'var')
                    if ~exist('unsolved_auto','var')
                        unsolved_auto = unsolved_auto_updated;
                    else
                        unsolved_auto = unique([unsolved_auto, unsolved_auto_updated]);
                    end;
                    fprintf('Updated unsolved vertices\n');
                end;
                unsolved_auto;

                auto_iterations = length(unsolved_auto);
                k = length(correspond_auto);
                i=1;
                inner_loop=0;
                while i<=auto_iterations && inner_loop==0
                    [tri2_new, newcloud2, corr_auto, stop] =
correspond_clouds(cloud1,cloud2,tri1,tri2,[unsolved_auto(i), unsolved_auto(i)],correspond_auto);
                    if stop~=0
                        if stop == 1
                            if sum(ismember(correspond_auto_solved,unsolved_auto(i)))==1
                                correspond_auto_errors_aug = [correspond_auto_errors,
unsolved_auto(i)];

                                correspond_auto_errors = unique(correspond_auto_errors_aug);
                                i=i+1;
                                fprintf('Stop 1: ismember\n');
                            else
                                correspond_auto_errors_aug = [correspond_auto_errors,
unsolved_auto(i)];

                                correspond_auto_errors = unique(correspond_auto_errors_aug);
                                i=i+1;
                                fprintf('Stop 1: notmember\n');
                            end;
                        else % stop == 2
                            if sum(ismember(correspond_auto,unsolved_auto(i)))==0
```

```
                                % Rename single point alone
                                newcloud2 = zeros(length(cloud2),3);

newcloud2(correspond_manual(i,1),:)=cloud2(correspond_manual(i,2),:);
                                index_rebuild = 1;
                                cloud2_subset = del_tri(cloud2,correspond_manual(i,2));
                                for p=1:length(cloud2)
                                    if sum(ismember(correspond_manual(i,1),p))==0
                                        newcloud2(p,:)=cloud2_subset(index_rebuild,:);
                                        index_rebuild = index_rebuild+1;
                                    end;
                                end;
                                if correspond_auto==0
                                    correspond_auto = [];
                                end;
                                correspond_auto2 = [correspond_auto, correspond_manual(i,1)];
                                correspond_auto = unique(correspond_auto2);
                                fprintf('Error: Cannot create correspondence group. Single
correspondence created\n');
                                i=i+1;
                                fprintf('Stop 2: notmember\n');
                            else
                                correspond_auto_errors_aug = [correspond_auto_errors,
unsolved_auto(i)];
                                correspond_auto_errors = unique(correspond_auto_errors_aug);
                                i=i+1;
                                fprintf('Stop 2: ismember\n');
                            end;
                        end;

                    else
                        % Check correspondence and error lists, remove as errors as they are
                        % solved
                        if ~isempty(correspond_auto_errors)
                            z_index = 1;
                            while z_index <= length(correspond_auto_errors)
                                if
sum(ismember(correspond_auto,correspond_auto_errors(z_index)))==1
                                    correspond_auto_errors(z_index)=[];
                                    fprintf('Error solved!\n');
                                else
                                    z_index = z_index+1;
                                end;
                            end;
                        end;
                        tri2 = tri2_new;
                        cloud2 = newcloud2;
                        correspond_auto = corr_auto;
                        correspond_auto_solved2 = [correspond_auto_solved, unsolved_auto(i)];
                        correspond_auto_solved = unique(correspond_auto_solved2);
                        j = length(correspond_auto);
                        if j==k
                            i=i+1;
                        else
                            k=j;
                        end;
                        fprintf('-----------------------------------------\n');
                        num_n = num2str(j);
                        tot_n = num2str(length(cloud1));
                        fprintf([num_n '/' tot_n ' points identified\n']);
                        s = length(correspond_auto_solved);
                        num_s = num2str(s);
                        fprintf([num_s '/' num_n ' correspondences fully solved\n']);
                        figure(2);
                        if label_visible == 1
                            set(label2,'visible','off')
                            clear label2;
                            label2 = plot_ptlabels(cloud2(:,1),cloud2(:,2),cloud2(:,3));
                        end;
                        autocoords2 = vertices2coords(cloud2,correspond_auto);
                        label3_2 = highlight_autopts(autocoords2);
```

```
                    solved_autocoords2 = vertices2coords(cloud2,correspond_auto_solved);
                    label3_2s = highlight_solvedautopts(solved_autocoords2);
                    figure(1);
                    autocoords1 = vertices2coords(cloud1,correspond_auto);
                    solved_autocoords1 = vertices2coords(cloud1,correspond_auto_solved);
                    label3_1 = highlight_autopts(autocoords1);
                    label3_1s = highlight_solvedautopts(solved_autocoords1);
                    fprintf('Replotting cloud labels\n');
                    unsolved_auto = unique([unsolved_auto, correspond_auto]);
                    unsolved_auto(ismember(unsolved_auto, correspond_auto_solved)) = [];
                    if exist('unsolved_auto_updated','var')
                        unsolved_auto = unique([unsolved_auto, unsolved_auto_updated]);
                    end;

                    if ~exist('unsolved_auto','var')
                        inner_loop=1;
                    else
                        auto_iterations = length(unsolved_auto);
                    end;
                end;
            end;
            autocoords2 = vertices2coords(cloud2,correspond_auto);
            solved_autocoords2 = vertices2coords(cloud2,correspond_auto_solved);
            autocoords1 = vertices2coords(cloud1,correspond_auto);
            solved_autocoords1 = vertices2coords(cloud1,correspond_auto_solved);
            figure(1);
            hold off;
            trisurf(tri1, cloud1(:,1), cloud1(:,2),
cloud1(:,3),'facecolor','white','edgecolor','blue');
            axis equal;
            hold on;
            label1 = plot_ptlabels(cloud1(:,1),cloud1(:,2),cloud1(:,3));
            label3_1 = highlight_autopts(autocoords1);
            label3_1s = highlight_solvedautopts(solved_autocoords1);
            grid off;
            figure(2);
            hold off;
            trisurf(tri2, cloud2(:,1), cloud2(:,2),
cloud2(:,3),'facecolor','white','edgecolor','blue');
            axis equal;
            hold on;
            label2 = plot_ptlabels(cloud2(:,1),cloud2(:,2),cloud2(:,3));
            label3_2 = highlight_autopts(autocoords2);
            label3_2s = highlight_solvedautopts(solved_autocoords2);
            grid off;

            % check if done...
            if length(correspond_auto)~=length(cloud1)
                fprintf('Attempting deformed triangulation repairs\n');
                for iteration=1:1
                    [cloud2, tri2, correspond_auto, correspond_auto_solved] =
retriangulate(cloud1, cloud2, tri1, tri2, correspond_auto, correspond_auto_solved);
                    correspond_auto = unique([correspond_auto, correspond_auto_solved]);
                end;
                fprintf('-----------------------------------------\n');
                l_ca = length(correspond_auto);
                num_n = num2str(l_ca);
                tot_n = num2str(length(cloud1));
                fprintf([num_n '/' tot_n ' points identified\n']);
                l_ca_s = length(correspond_auto_solved);
                num_s = num2str(l_ca_s);
                fprintf([num_s '/' num_n ' correspondences fully solved\n']);

                % Find new unsolved
                unsolved_auto_updated = find_unsolved_verts(tri1, tri2,
correspond_auto_solved);
                unsolved_auto_updated

                % Replot all figures
                autocoords2 = vertices2coords(cloud2,correspond_auto);
                solved_autocoords2 = vertices2coords(cloud2,correspond_auto_solved);
```

```
                    autocoords1 = vertices2coords(cloud1,correspond_auto);
                    solved_autocoords1 = vertices2coords(cloud1,correspond_auto_solved);
                    figure(1);
                    hold off;
                    trisurf(tri1, cloud1(:,1), cloud1(:,2),
cloud1(:,3),'facecolor','white','edgecolor','blue');
                    axis equal;
                    hold on;
                    grid off;
                    label1 = plot_ptlabels(cloud1(:,1),cloud1(:,2),cloud1(:,3));
                    label3_1 = highlight_autopts(autocoords1);
                    label3_1s = highlight_solvedautopts(solved_autocoords1);
                    figure(2);
                    hold off;
                    trisurf(tri2, cloud2(:,1), cloud2(:,2),
cloud2(:,3),'facecolor','white','edgecolor','blue');
                    axis equal;
                    hold on;
                    grid off;
                    label2 = plot_ptlabels(cloud2(:,1),cloud2(:,2),cloud2(:,3));
                    label3_2 = highlight_autopts(autocoords2);
                    label3_2s = highlight_solvedautopts(solved_autocoords2);
                    figure(3);
                    hold off;
                    trisurf(tri2, cloud2(:,1), cloud2(:,2),
cloud2(:,3),'facecolor','white','edgecolor','blue');
                    axis equal;
                    hold on;
                    grid off;
                    plot3(cloud2(:,1),cloud2(:,2),cloud2(:,3),'ko');
                    label3 = plot_ptlabels(cloud2(:,1),cloud2(:,2),cloud2(:,3));
                    solved_autocoords3 = vertices2coords(cloud2,correspond_auto_solved);
                    label3_3s = highlight_solvedautopts(solved_autocoords3);
                    autocoords3 = vertices2coords(cloud2, unsolved_auto_updated);
                    label3_3 = highlight_autopts(autocoords3);
                else
                    fprintf('------Correspondence Problem Fully Solved------\nCloud 2
reogranized, saved to workspace.\n');
                    cloud2_list = cloud2;
                end;
            end;
        end;
    end;
    if strcmp(resp1,'l')==1
        if label_visible == 1
            label_visible = 0;
            fprintf('Labels off\n');
            set(label1,'visible','off')
            set(label2,'visible','off')
        else label_visible = 1;
            fprintf('Labels on\n');
            set(label1,'visible','on')
            set(label2,'visible','on')
        end;
    end;
    if strcmp(resp1,'s')==1
        fprintf('Saving data to workspace');
        cloud1_list = cloud1;
        cloud2_list = cloud2;
        c_matrix = correspond_auto;
    end;
    if strcmp(resp1,'q')==1
        fprintf('Exiting Program\n');
        done = 1;
    end;
end;
```

correspond_clouds.m

```
function [tri2_new, cloud2new, corr_auto, stop_error] = correspond_clouds(cloud1, cloud2, tri1,
tri2, correspond_manual, correspond_auto)
% % Determines correspondences between point clouds.

% For the manual correspondence
% First calculate adjacent triangles and vertices in original cloud

pt_index1 = correspond_manual(1,1);
adj_vertices1 = find_adj_vertices(pt_index1, tri1);
pt_index2 = correspond_manual(1,2);
adj_vertices2 = find_adj_vertices(pt_index2, tri2);
data1 = vertices2coords(cloud1,adj_vertices1);
data2 = vertices2coords(cloud2,adj_vertices2);

% Align clouds according to known center point. Pure translation
data2_moved(:,1) = data2(:,1)+(data1(1,1)-data2(1,1));
data2_moved(:,2) = data2(:,2)+(data1(1,2)-data2(1,2));
data2_moved(:,3) = data2(:,3)+(data1(1,3)-data2(1,3));

% Rotate second group according to knowledge about other correspondences
if length(adj_vertices1)==length(adj_vertices2)
    adj_vertices1b = adj_vertices1(2:end);
    adj_vertices2b = adj_vertices2(2:end);

    index_1b = 0;
    index_2b = 0;
    d = 1;
    cont = 0;
    while cont==0
        if sum(ismember(adj_vertices1b,adj_vertices2b(d)))==1 &&
sum(ismember(correspond_auto,adj_vertices2b(d)))==1
            index_2b = d;
            cont = 1;
        else
            if d<length(adj_vertices1b)
                d = d+1;
            else
                cont = 1;
            end;
        end;
    end;
    if index_2b~=0
        point2b = adj_vertices2b(d);
        coords2b = vertices2coords(cloud2,point2b)+[(data1(1,1)-data2(1,1)), (data1(1,2)-
data2(1,2)), (data1(1,3)-data2(1,3))];
        g = 1;
        cont = 0;
        while cont==0
            if adj_vertices1b(g)==point2b
                index_1b = g;
                cont = 1;
            else
                if g<length(adj_vertices1b)
                    g = g+1;
                else
                    cont = 1;
                end;
            end;
        end;
        point1b = adj_vertices1b(g);
        coords1b = vertices2coords(cloud1,point1b);

        % Move both to origin
        data1_trans(:,1) = data1(:,1)-data1(1,1);
        data1_trans(:,2) = data1(:,2)-data1(1,2);
        data1_trans(:,3) = data1(:,3)-data1(1,3);

        data2_trans(:,1) = data2_moved(:,1)-data1(1,1);
```

```
        data2_trans(:,2) = data2_moved(:,2)-data1(1,2);
        data2_trans(:,3) = data2_moved(:,3)-data1(1,3);

        coords1bt = coords1b-[data1(1,1), data1(1,2), data1(1,3)];
        coords2bt = coords2b-[data1(1,1), data1(1,2), data1(1,3)];

        ray1 = coords1bt;
        ray2 = coords2bt;

        rot = vrrotvec(ray2,ray1);

        rot_matrix = R3d(rot(4),rot(1:3));
        for h=1:length(adj_vertices1)
            data2_trans_rot(h,:)=rot_matrix*data2_trans(h,:)';
        end;
        data1 = data1_trans;
        data2_moved = data2_trans_rot;
    end;
end;

[R,t,cor,error,data2f] = icp2(data1',data2_moved',15);

% Error checks

c_size = size(cor);
c_length = c_size(1);
if c_length~=length(adj_vertices1) || c_length~=length(adj_vertices2) || error>0.28

    % Attempt final check
    % Change orientation of points depending on known correspondences

    if sum(ismember(correspond_auto, adj_vertices1))==length(adj_vertices1) &&
sum(ismember(correspond_auto, adj_vertices2))==length(adj_vertices2)
        % all solved, quit
        stop_error = 1;
        fprintf('All adjacent vertices solved\n');
        correspond_manual
        corr_auto = correspond_auto;
        cloud2new = cloud2;
        tri2_new = tri2;
    else
        common_verts = unique([adj_vertices1, adj_vertices2]);
        common_verts(~ismember(common_verts, correspond_auto))=[];
        if length(common_verts)==length(adj_vertices1)+1 ...
                && length(adj_vertices1)==length(adj_vertices2) ...
                &&
sum(ismember(correspond_auto,adj_vertices1))==sum(ismember(correspond_auto,adj_vertices2))
            common_verts
            adj_vertices1
            adj_vertices2
            correspond_auto
            correspond_manual

            for w=1:length(adj_vertices1)
                if sum(ismember(adj_vertices1,adj_vertices2(w)))==1
                    data2_moved(w,:)=vertices2coords(cloud1,adj_vertices2(w));
                else
                    v1_pos = [];
                    for u=1:length(adj_vertices1)
                        if sum(ismember(correspond_auto,adj_vertices1(u)))==0
                            v1_pos = u;
                        end;
                    end;
                    data2_moved(w,:)=vertices2coords(cloud1,adj_vertices1(v1_pos));
                end;
            end;

            [R,t,cor,error,data2f] = icp2(data1',data2_moved',15);

            c_size = size(cor);
            c_length = c_size(1);
```

```
if c_length~=length(adj_vertices1) || c_length~=length(adj_vertices2) || error>0.25
    stop_error = 1;
    if c_length~=length(adj_vertices1) || c_length~=length(adj_vertices2)
        fprintf('Triangulation dimension mismatch\n');
    else
        fprintf('Error limit exceeded\n');
    end;
    error
    corr_auto = correspond_auto;
    cloud2new = cloud2;
    tri2_new = tri2;
else
    stop_error = 0;
    if correspond_auto==0
        correspond_auto=[];
    end;
    corr_auto = [correspond_auto, adj_vertices1];
    corr_auto = unique(corr_auto);
    corr_auto2 = [correspond_auto, adj_vertices2];
    corr_auto2 = unique(corr_auto2);

    dim_m = size(adj_vertices1);
    m = dim_m(2);
    cloud2new = zeros(length(cloud2),3);
    for j=1:m
        index_1 = cor(j,1);
        index_2 = cor(j,2);
        vertex1 = adj_vertices1(index_1);
        vertex2 = adj_vertices2(index_2);
        cloud2new(vertex1,:)=cloud2(vertex2,:);
        cloud2new(vertex2,:)=cloud2(vertex1,:);
    end;
    index_rebuild = 1;
    cloud2_subset = del_tri(cloud2,unique([corr_auto2, adj_vertices1]));
    for p=1:length(cloud2)
        if sum(ismember(unique([corr_auto, adj_vertices2]),p))==0
            cloud2new(p,:)=cloud2_subset(index_rebuild,:);
            index_rebuild = index_rebuild+1;
        else
            if sum(ismember(adj_vertices1,p))==0 && sum(ismember(adj_vertices2,p))==0
                cloud2new(p,:)=cloud2(p,:);
            end;
        end;
    end;

    % Check against known previous cloud points if any, look for
    % discrepancies: cloud2new vs cloud2, corr_auto vs correspond_auto
    for i=1:length(correspond_auto)
        if sum(ismember(corr_auto,correspond_auto(i)))==1
            if cloud2new(correspond_auto(i))~=cloud2(correspond_auto(i))
                stop_error = 2;
            end;
        end;
    end;

    % Renumber triangles in tri2 accordingly
    tri2_new = tri2;
    for i=1:length(adj_vertices1)
        vert = adj_vertices2(cor(i,2));
        vert_move = adj_vertices1(cor(i,1));
        for z=1:length(tri2)
            triangle = tri2(z,:);
            for x=1:3
                if triangle(x)==vert
                    tri2_new(z,x) = vert_move;
                else
                    if triangle(x)==vert_move;
                        tri2_new(z,x) = vert;
                    end;
                end;
```

```
                    end;
                end;
            end;
            tri2_new = sort(tri2_new, 2);
            % tri2_new = sort(tri2_new, 1);

            if stop_error == 2
                fprintf('Correspondence mismatch... reverting\n');
                corr_auto = correspond_auto;
                cloud2new = cloud2;
                tri2_new = tri2;
            end;

        end;
    else
        stop_error = 1;
        correspond_manual
        if c_length~=length(adj_vertices1) || c_length~=length(adj_vertices2)
            fprintf('Triangulation dimension mismatch\n');
        else
            fprintf('Error limit exceeded\n');
        end;
        error
        tri2_new = tri2;
        corr_auto = correspond_auto;
        cloud2new = cloud2;
    end;
end;
else
    stop_error = 0;

    % Then find the corresponding adjacent triangles and vertices in the
    % deformed cloud according to the affine transform from the ICP
    % algorithm

    % Reorder the points in cloud 2 according to the correspondence matrix
    % cor

    % Add automatic correspondences to manual correspondences, remove redundant
    % correspondences
    if correspond_auto==0
        correspond_auto=[];
    end;
    corr_auto = [correspond_auto, adj_vertices1];
    corr_auto = unique(corr_auto);
    corr_auto2 = [correspond_auto, adj_vertices2];
    corr_auto2 = unique(corr_auto2);

    dim_m = size(adj_vertices1);
    m = dim_m(2);
    cloud2new = zeros(length(cloud2),3);
    for j=1:m
        index_1 = cor(j,1);
        index_2 = cor(j,2);
        vertex1 = adj_vertices1(index_1);
        vertex2 = adj_vertices2(index_2);
        cloud2new(vertex1,:)=cloud2(vertex2,:);
        cloud2new(vertex2,:)=cloud2(vertex1,:);
    end;
    index_rebuild = 1;
    cloud2_subset = del_tri(cloud2,unique([corr_auto2, adj_vertices1]));
    for p=1:length(cloud2)
        if sum(ismember(unique([corr_auto, adj_vertices2]),p))==0
            cloud2new(p,:)=cloud2_subset(index_rebuild,:);
            index_rebuild = index_rebuild+1;
        else
            if sum(ismember(adj_vertices1,p))==0 && sum(ismember(adj_vertices2,p))==0
                cloud2new(p,:)=cloud2(p,:);
            end;
        end;
    end;
```

```
    if length(unique(cloud2new,'rows'))~=length(cloud2)
        fprintf('Correspondence error: mismatched points in cloud\n')
        cm = correspond_manual;
        ca =corr_auto;
        ca2= corr_auto2;
        a1 = adj_vertices1;
        a2 = adj_vertices2;
        cl2 = cloud2;
        clnew = cloud2new;
        stop_error = 2;
    end;

    % Check against known previous cloud points if any, look for
    % discrepancies: cloud2new vs cloud2, corr_auto vs correspond_auto
    for i=1:length(correspond_auto)
        if sum(ismember(corr_auto,correspond_auto(i)))==1
            if cloud2new(correspond_auto(i))~=cloud2(correspond_auto(i))
                stop_error = 2;
            end;
        end;
    end;

    % Renumber triangles in tri2 accordingly
    tri2_new = tri2;
    for i=1:length(adj_vertices1)
        vert = adj_vertices2(cor(i,2));
        vert_move = adj_vertices1(cor(i,1));
        for z=1:length(tri2)
            triangle = tri2(z,:);
            for x=1:3
                if triangle(x)==vert
                    tri2_new(z,x) = vert_move;
                else
                    if triangle(x)==vert_move;
                        tri2_new(z,x) = vert;
                    end;
                end;
            end;
        end;
    end;
    tri2_new = sort(tri2_new, 2);

    if stop_error == 2
        fprintf('Correspondence mismatch... reverting\n');
        corr_auto = correspond_auto;
        cloud2new = cloud2;
        tri2_new = tri2;
    end;
end;
```

retriangulate.m

```
function [cloud2, tri2, c_auto, c_auto_solved] = retriangulate(cloud1, cloud2, tri1, tri2,
c_auto, c_auto_solved)
% Automatically repairs tri2 with information from clouds 1 and 2, tri1,
% and the lists of correspondences
v1_unique = [];
v2_unique = [];
v2_verified = [];
tri1_verts = [];
tri2_verts = [];


for m=1:length(tri1)
    % Check if a triangle contains exactly 2 verified (solved)
```

```
    % correspondences.
    clear tri1_verts;
    tri1_verts = tri1(m,:);
    for n=1:length(tri2)
        clear tri2_verts;
        tri2_verts = tri2(n,:);
        if sum(ismember(tri1_verts,tri2_verts))>=1 && sum(ismember(c_auto_solved, tri2_verts))>=2
            % Find mismatched vertex
            if sum(ismember(tri1_verts,tri2_verts))==3
                % Confirm the vertex that is not in the solved array
                clear v1_unique;
                v1_unique = [];
                for z=1:3
                    if sum(ismember(c_auto_solved, tri1_verts(z)))==0
                        v1_unique = tri1_verts(z);
                    end;
                end;
                % Confirm correspond auto. Add to correspond auto solved
                % array
                if ~isempty(v1_unique) && sum(ismember(c_auto_solved, v1_unique))==0
                    clear adj_verts1 adj_verts2 av1_saved av2_saved av1_errors av2_errors stop1
                    adj_verts1 = find_adj_vertices(v1_unique, tri1);
                    adj_verts2 = find_adj_vertices(v1_unique, tri2);
                    adj_verts1(~ismember(adj_verts1, c_auto_solved))=[];
                    adj_verts2(~ismember(adj_verts2, c_auto_solved))=[];
                    av1_saved = adj_verts1;
                    av2_saved = adj_verts2;
                    adj_verts1(ismember(adj_verts1, adj_verts2))=[];
                    av1_errors = unique(adj_verts1);
                    adj_verts1 = av1_saved;
                    adj_verts2 = av2_saved;
                    adj_verts2(ismember(adj_verts2, adj_verts1))=[];
                    av2_errors = unique(adj_verts2);
                    stop1 = 0;
                    if ~isempty(av1_errors) || ~isempty(av2_errors)
                        stop2 = 0;
                        if stop2 == 0
                            fprintf('Triangulation correspondence inconsistency\n');
                            v1_unique
                            tri1_verts
                            tri2_verts
                            av1_errors
                            av2_errors
                            stop1 = 1;
                        end;
                    end;
                    if stop1==0
                        c_auto_solved = unique([c_auto_solved, v1_unique]);
                        c_auto = unique([c_auto, v1_unique]);
                        fprintf('Added to correspond auto solved\n');
                        v1_unique
                        tri1_verts
                        tri2_verts
                        fprintf('--------\n');
                    end;
                end;
            end;
        end;
    end;
end;

% Remove any solved correspondences that have triangulation
% inconsistencies
vert_remove = [];
for z=1:length(c_auto)
    clear adj_verts1 adj_verts2 av1_saved av2_saved av1_errors av2_errors stop1 vert
    vert = c_auto(z);
    adj_verts1 = find_adj_vertices(vert, tri1);
    adj_verts2 = find_adj_vertices(vert, tri2);
    adj_verts1(~ismember(adj_verts1, c_auto))=[];
    adj_verts2(~ismember(adj_verts2, c_auto))=[];
```

```
    av1_saved = adj_verts1;
    av2_saved = adj_verts2;
    adj_verts1(ismember(adj_verts1, adj_verts2))=[];
    av1_errors = unique(adj_verts1);
    adj_verts1 = av1_saved;
    adj_verts2 = av2_saved;
    adj_verts2(ismember(adj_verts2, adj_verts1))=[];
    av2_errors = unique(adj_verts2);
    remove = 0;
    if length(av1_errors)>1 && isempty(av2_errors)
        % remove
        remove = 1;
    elseif length(av2_errors)>1 && isempty(av1_errors)
        % remove
        remove = 1;
    elseif ~isempty(av1_errors) && ~isempty(av2_errors)
        if length(av1_errors)==1 && length(av2_errors)==1 && sum(ismember(c_auto, av1_errors))==1
&& sum(ismember(c_auto, av2_errors))==1
            % keep correspondence until triangulation is analyzed
            remove = 0;
        else remove = 1;
        end;
    else remove = 0;
    end;
    if remove == 1
        % remove correspondence
        vert;
        av1_errors;
        av2_errors;
        vert_remove = unique([vert_remove, vert]);
        c_auto_solved(ismember(c_auto_solved, vert))=[];
        c_auto(ismember(c_auto, vert))=0;
    end;
end;

vert_remove
c_auto(ismember(c_auto, 0))=[];
c_auto;
c_auto_solved;
pause

for m=1:length(tri1)
    % Check if a triangle contains exactly 2 verified (solved)
    % correspondences.
    clear tri1_verts;
    tri1_verts = tri1(m,:);
    for n=1:length(tri2)
        clear tri2_verts;
        tri2_verts = tri2(n,:);
        if sum(ismember(tri1_verts,tri2_verts))>=1 && sum(ismember(c_auto_solved, tri2_verts))>=2
            % Find mismatched vertex
            if sum(ismember(tri1_verts,tri2_verts))==3
                % Already done, do nothing for this case
            else
                if sum(ismember(c_auto,tri2_verts))==3 && sum(ismember(c_auto_solved,
tri1_verts))==3
                else
                    if sum(ismember(tri2, tri1_verts,'rows'))==1
                    else
                        if sum(ismember(c_auto, tri1_verts))==3 && sum(ismember(c_auto_solved,
tri2_verts))==2
                            fprintf('Condition 1\n');
                            verts1 = unique(tri2_verts(ismember(tri2_verts, tri1_verts)))
                            verts2 = unique(c_auto_solved(ismember(c_auto_solved,tri2_verts)))
                            if length(verts1)==2 && length(verts2)==2
                                if verts1(1)==verts2(1) && verts1(2)==verts2(2)

                                    for z=1:3
                                        if sum(ismember(tri1_verts, tri2_verts(z)))==0
                                            v2_unique = tri2_verts(z)
                                            tri1_verts
```

```
                                        tri2_verts

                                    end;
                                end;
                                % verify correct triangle is identified
                                % check if centroids are close to one
                                % another- indicates overlapping
                                % triangles
                                stop_correction = 0;

                                % get coordinates
                                tri1_coords_in2 = vertices2coords(cloud2, tri1_verts);
                                tri2_coords = vertices2coords(cloud2, tri2_verts);
                                % get centroids
                                t1_in2_centroid =
tri_centroid(tri1_coords_in2(1,:),tri1_coords_in2(2,:),tri1_coords_in2(3,:));
                                t2_centroid =
tri_centroid(tri2_coords(1,:),tri2_coords(2,:),tri2_coords(3,:));
                                % side lengths
                                t1_2_lengths =
tri_lengths(tri1_coords_in2(1,:),tri1_coords_in2(2,:),tri1_coords_in2(3,:))
                                t2_lengths =
tri_lengths(tri2_coords(1,:),tri2_coords(2,:),tri2_coords(3,:))
                                mean_lengths = mean([t1_2_lengths, t2_lengths])
                                % if euclidean distance between
                                % centroids is smaller than the average
                                % leg length, triangles verified
                                distance = ((t1_in2_centroid(1)-
t2_centroid(1))^2+(t1_in2_centroid(2)-t2_centroid(2))^2+(t1_in2_centroid(3)-
t2_centroid(3))^2)^0.5;
                                distance
                                if distance>mean_lengths*0.4 % compare to nearby centroids
instead
                                    fprintf('Wrong triangle identified\n');
                                    stop_correction = 1;
                                end;
                                if sum(ismember(tri2,tri1_verts,'rows'))==0 &&
stop_correction==0
                                    c_auto
                                    % Save original tri2
                                    tri2_saved = tri2;
                                    fprintf('Correcting triangulation: Using tri1\n');
                                    tri2(n,:) = tri1_verts;

                                    % add second mistriangulated element
                                    common_verts = unique(tri2_verts(ismember(tri2_verts,
tri1_verts)));
                                    common_verts;
                                    tri2_common = []; % two common vertices
                                    tri2_other_list = []; % single uncorresponded vertex
                                    tri2_other_index_list = []; % index of other triangle
                                    adj_tri = [];
                                    for z=1:length(tri2)
                                        tri2_candidate = tri2(z,:);
                                        if sum(ismember(tri2_verts,tri2_candidate))==2 &&
sum(ismember(c_auto, tri2_candidate))>=2 && sum(ismember(tri1_verts, tri2_candidate))==2 && z~=n
                                            % check that 2 solved vertices
                                            % are in common
                                            verts1 =
tri1_verts(ismember(tri1_verts,tri2_candidate));
                                            verts2 = c_auto(ismember(c_auto,tri2_candidate));
                                            if sum(ismember(verts1, verts2))==2
%verts1(1)==verts2(1) && verts1(2)==verts2(2)
                                                tri2_common =
unique(tri1_verts(ismember(tri1_verts,tri2_candidate)));
                                                adj_tri = [adj_tri; tri2_candidate];
                                                for x=1:3
                                                    if
sum(ismember(tri2_common,tri2_candidate(x)))==0
                                                        tri2_other = tri2_candidate(x);
                                                        tri2_other_index = z;
```

```
                                                            tri2_other_list = [tri2_other_list,
tri2_other];
                                                            tri2_other_index;
                                                            tri2_other_index_list
=[tri2_other_index_list, tri2_other_index];
                                                        end;
                                                    end;
                                                end;
                                            end;
                                        end;

                                        % Pick out closest triangle
                                        % from list- compare to average
                                        % centroid of known input
                                        % triangles
                                        adj_tri;
                                        t_v1 = vertices2coords(cloud2, tri1_verts);
                                        t_v2 = vertices2coords(cloud2, tri2_verts);
                                        t_cen1 = tri_centroid(t_v1(1,:),t_v1(2,:),t_v1(3,:));
                                        t_cen2 = tri_centroid(t_v2(1,:),t_v2(2,:),t_v2(3,:));
                                        %center = [mean([t_cen1(1), t_cen2(1)]), mean([t_cen1(2),
t_cen2(2)]), mean([t_cen1(3), t_cen2(3)])];
                                        t_cen1;
                                        t_cen2;
                                        d_list = [];
                                        for z=1:length(tri2_other_index_list)
                                            t_v = adj_tri(z,:);
                                            t_v_pts = vertices2coords(cloud2, t_v);
                                            t_cen =
tri_centroid(t_v_pts(1,:),t_v_pts(2,:),t_v_pts(3,:));
                                            t_cen;
                                            dist = ((t_cen1(1)-t_cen(1))^2+(t_cen1(2)-
t_cen(2))^2+(t_cen1(3)-t_cen(3))^2)^0.5+((t_cen2(1)-t_cen(1))^2+(t_cen2(2)-
t_cen(2))^2+(t_cen2(3)-t_cen(3))^2)^0.5;
                                            d_list = [d_list, dist];
                                        end;
                                        [m_val, m_index] = min(d_list);
                                        tri2_other_index = tri2_other_index_list(m_index);
                                        tri2_other = tri2_other_list(m_index);
                                        tri2_other_list;
                                        d_list;
                                        tri2_other
                                        tri2_other_index;


                                        tri1_common = [];
                                        for z=1:length(tri1)
                                            tri1_candidate = tri1(z,:);
                                            tri2_othertri = tri2(tri2_other_index,:);
                                            if sum(ismember(tri1_verts,tri1_candidate))==2 &&
sum(ismember(c_auto, tri1_candidate))>=2 && sum(ismember(tri2_verts, tri1_candidate))>=1 &&
sum(ismember(tri2_othertri,tri1_candidate))>=1 && z~=tri2_other_index
                                                % check that 2 solved vertices
                                                % are in common
                                                tri2_othertri;
                                                tri1_com =
unique(tri1_verts(ismember(tri1_verts,tri1_candidate)));
                                                tri_new = unique([tri1_com, tri2_other]);
                                                tri2_saved = sort(tri2_saved, 2);
                                                if sum(ismember(tri2_saved, tri_new, 'rows'))==0
                                                    tri1_common = tri1_com;
                                                end;
                                            end;
                                        end;
                                        tri1_common
                                        %tri2_other
                                        tri2_second = unique([tri1_common, tri2_other])
                                        if isempty(tri1_common)
                                            fprintf('Mismatched...\n');
                                        else
```

```
                                        fprintf('Correcting triangulation: Second
triangle\n');
                                        tri2(tri2_other_index,:)=tri2_second;
                                        tri2 = sort(tri2, 2);
                                        %tri2 = sort(tri2, 1);
                                        fprintf('--------\n');
                                        % pause
                                    end;
                                end;
                            end;
                        end;
                    else
                        if sum(ismember(c_auto, tri1_verts))==2 && sum(ismember(c_auto,
tri2_verts))==3
                            fprintf('Condition 2\n');
                            tri1_verts
                            tri2_verts
                            verts1 = unique(tri2_verts(ismember(tri2_verts, tri1_verts)))
                            verts2 = unique(c_auto(ismember(c_auto,tri1_verts)))
                            if length(verts1)==2 && length(verts2)==2
                                if verts1(1)==verts2(1) && verts1(2)==verts2(2)

                                    % verify correct triangle is identified
                                    % check if centroids are close to one
                                    % another- indicates overlapping
                                    % triangles
                                    stop_correction = 0;

                                    % get coordinates
                                    tri1_coords_in1 = vertices2coords(cloud1, tri1_verts);
                                    tri2_coords = vertices2coords(cloud1, tri2_verts);
                                    % get centroids
                                    t1_in2_centroid =
tri_centroid(tri1_coords_in1(1,:),tri1_coords_in1(2,:),tri1_coords_in1(3,:));
                                    t2_centroid =
tri_centroid(tri2_coords(1,:),tri2_coords(2,:),tri2_coords(3,:));
                                    % side lengths
                                    t1_2_lengths =
tri_lengths(tri1_coords_in1(1,:),tri1_coords_in1(2,:),tri1_coords_in1(3,:));
                                    t2_lengths =
tri_lengths(tri2_coords(1,:),tri2_coords(2,:),tri2_coords(3,:));
                                    mean_lengths = mean([t1_2_lengths, t2_lengths])
                                    % if euclidean distance between
                                    % centroids is smaller than the average
                                    % leg length, triangles verified
                                    distance = ((t1_in2_centroid(1)-
t2_centroid(1))^2+(t1_in2_centroid(2)-t2_centroid(2))^2+(t1_in2_centroid(3)-
t2_centroid(3))^2)^0.5;
                                    distance
                                    if distance>mean_lengths*0.4 % compare to nearby
centroids instead
                                        fprintf('Wrong triangle identified\n');
                                        fprintf('--------\n');
                                        stop_correction = 1;
                                    end;
                                    if sum(ismember(tri2,tri1_verts,'rows'))==0 &&
stop_correction==0 && sum(ismember(tri1, tri2_verts,'rows'))==0
                                        c_auto

                                        common_verts = unique(tri2_verts(ismember(tri2_verts,
tri1_verts)));
                                        % fix mistriangulated elements
                                        common_verts;
                                        tri2_common = []; % two common vertices
                                        tri2_other_list = []; % single uncorresponded vertex
                                        tri2_other_index_list = []; % index of other triangle
                                        adj_tri = [];
                                        for z=1:length(tri2)
                                            tri2_candidate = tri2(z,:);
                                            if sum(ismember(tri2_verts,tri2_candidate))==2 &&
sum(ismember(c_auto, tri2_candidate))>=2 && sum(ismember(tri1_verts, tri2_candidate))>=1 && z~=n
```

```
                                                        % check that 2 solved vertices
                                                        % are in common
                                                        tri2_candidate;
                                                        verts1 =
sort(tri2_verts(ismember(tri2_verts,tri2_candidate)));
                                                        verts2 =
sort(c_auto(ismember(c_auto,tri2_candidate)));
                                                        verts1;
                                                        verts2;
                                                        if verts1(1)==verts2(1) &&
verts1(2)==verts2(2)
                                                            tri2_common =
unique(tri2_verts(ismember(tri2_verts,tri2_candidate)));
                                                            adj_tri = [adj_tri; tri2_candidate];
                                                            for x=1:3
                                                                if sum(ismember(tri2_common,
tri2_candidate(x)))==0 %sum(ismember(c_auto,tri2_candidate(x)))==0
                                                                    tri2_other = tri2_candidate(x);
                                                                    tri2_other_index = z;
                                                                    tri2_other_list =
[tri2_other_list, tri2_other];
                                                                    tri2_other_index;
                                                                    tri2_other_index_list
=[tri2_other_index_list, tri2_other_index];
                                                                end;
                                                            end;
                                                        end;
                                                    end;
                                                end;

                                                % Pick out closest triangle
                                                % from list- compare to average
                                                % centroid of known input
                                                % triangles
                                                adj_tri
                                                t_v2 = vertices2coords(cloud2, tri2_verts);
                                                if sum(ismember(c_auto, tri1_verts))==3
                                                    t_v1 = vertices2coords(cloud2, tri1_verts);
                                                else
                                                    t_v1 = vertices2coords(cloud2, tri2_verts);
                                                end;
                                                t_cen1 = tri_centroid(t_v1(1,:),t_v1(2,:),t_v1(3,:));
                                                t_cen2 = tri_centroid(t_v2(1,:),t_v2(2,:),t_v2(3,:));
                                                center = [mean([t_cen1(1), t_cen2(1)]),
mean([t_cen1(2), t_cen2(2)]), mean([t_cen1(3), t_cen2(3)])];
                                                t_cen1
                                                t_cen2
                                                d_list = [];
                                                for z=1:length(tri2_other_index_list)
                                                    t_v = adj_tri(z,:);
                                                    t_v_pts = vertices2coords(cloud2, t_v);
                                                    t_cen =
tri_centroid(t_v_pts(1,:),t_v_pts(2,:),t_v_pts(3,:));
                                                    t_cen
                                                    dist = ((t_cen1(1)-t_cen(1))^2+(t_cen1(2)-
t_cen(2))^2+(t_cen1(3)-t_cen(3))^2)^0.5+((t_cen2(1)-t_cen(1))^2+(t_cen2(2)-
t_cen(2))^2+(t_cen2(3)-t_cen(3))^2)^0.5;
                                                    d_list = [d_list, dist];
                                                end;
                                                [m_val, m_index] = min(d_list);
                                                tri2_other_index = tri2_other_index_list(m_index);
                                                tri2_other = tri2_other_list(m_index);
                                                tri2_other_list
                                                d_list
                                                tri2_other
                                                tri2_other_index;
                                                tri2_common;
                                                fprintf('Correcting triangulation: First
triangle\n');
                                                tri2_first = unique([common_verts, tri2_other]);
                                                tri2_first
```

```
                                            tri2(n,:)=tri2_first;

                                            tri2_common2 = [];
                                            for z=1:length(tri1)
                                                tri1_candidate = tri1(z,:);
                                                if sum(ismember(tri1_verts,tri1_candidate))==2 &&
sum(ismember(c_auto, tri1_candidate))>=2 && sum(ismember(tri2_verts, tri1_candidate))>=2 && z~=m
% && sum(ismember(tri2(tri2_other_index,:),tri1_candidate))==1
                                                    % check that 2
                                                    % vertices- 1 solved
                                                    % are in common
                                                    tri2_com2 =
unique(tri2_verts(ismember(tri2_verts,tri1_candidate)));
                                                    tri2_new = unique([tri2_com2, tri2_other]);
                                                    tri2 = sort(tri2, 2);
                                                    if sum(ismember(tri2, tri2_new, 'rows'))==0
                                                        tri2_common2 = tri2_com2;
                                                    end;
                                                end;
                                            end;
                                            if isempty(tri2_common2)
                                                fprintf('Mismatched...\n');
                                            else
                                                tri2_second = unique([tri2_common2, tri2_other]);
                                                tri2_second
                                                fprintf('Correcting triangulation: Second
triangle\n');

                                                tri2(tri2_other_index,:)=tri2_second;
                                                tri2 = sort(tri2, 2);
                                                %tri2 = sort(tri2, 1);
                                                fprintf('--------\n');
                                                % pause
                                            end;
                                        end;
                                    end;
                                end;
                            end;
                        end;
                    end;
                end;
            end;
        end;
    end;
end;
tri2 = sort(tri2, 2);

% Remove any solved correspondences that have triangulation
% inconsistencies
vert_remove = [];
for z=1:length(c_auto)
    clear adj_verts1 adj_verts2 av1_saved av2_saved av1_errors av2_errors stop1 vert
    vert = c_auto(z);
    adj_verts1 = find_adj_vertices(vert, tri1);
    adj_verts2 = find_adj_vertices(vert, tri2);
    adj_verts1(~ismember(adj_verts1, c_auto))=[];
    adj_verts2(~ismember(adj_verts2, c_auto))=[];
    av1_saved = adj_verts1;
    av2_saved = adj_verts2;
    adj_verts1(ismember(adj_verts1, adj_verts2))=[];
    av1_errors = unique(adj_verts1);
    adj_verts1 = av1_saved;
    adj_verts2 = av2_saved;
    adj_verts2(ismember(adj_verts2, adj_verts1))=[];
    av2_errors = unique(adj_verts2);

    if ~isempty(av1_errors) || ~isempty(av2_errors)
        %if length(av1_errors)==1 && length(av2_errors)==1
            % keep correspondence until triangulation is analyzed
       % else
            % remove correspondence
            vert
```

```
            av1_errors
            av2_errors
            vert_remove = unique([vert_remove, vert]);
            c_auto_solved(ismember(c_auto_solved, vert))=[];
            c_auto(ismember(c_auto, vert))=0;
        %end;
    end;
end;
vert_remove
c_auto(ismember(c_auto, 0))=[];
c_auto;
c_auto_solved;
```

Appendix C: Strain Computations and Plotting

plot_mises_strain.m

```
function plot_mises_strain(tri,xi,yi,zi,xf,yf,zf)
% computes equivalent strains for every element in the triangulation, tri.
% uses the corresponded point clouds.
global tri_wstrain;

i = size(tri);
j = i(1);
clear i;
trisurf(tri,xf,yf,zf,'facecolor','none','edgecolor','b');
hold on;
tri_r = TriRep(tri,xf,yf,zf);
ic = incenters(tri_r);
for i=1:j
    [x_i,y_i,z_i]=tri_coords(tri,xi,yi,zi,i);
    [x_f,y_f,z_f]=tri_coords(tri,xf,yf,zf,i);
    strain_tr = tri_strain(x_i,y_i,z_i,x_f,y_f,z_f);
    fill3(x_f,y_f,z_f,strain_tr);
    tri_wstrain(i,:) = [tri(i,:) strain_tr];

end;
axis equal;
median_strain = median(tri_wstrain(:,4));
std_strain = std(tri_wstrain(:,4));
cbar = colorbar('location','eastoutside');
colormap(gray);
caxis([median_strain-4*std_strain,median_strain+4*std_strain]);
set(get(cbar,'ylabel'),'string','Von Mises Strain');
```

tri_strain.m

```
function strain_mises = tri_strain(x1,y1,z1,x2,y2,z2)
% computes equivalent strain for one triangular element.

p1 = [x1(1),y1(1),z1(1)];
p2 = [x1(2),y1(2),z1(2)];
p3 = [x1(3),y1(3),z1(3)];
p4 = [x2(1),y2(1),z2(1)];
p5 = [x2(2),y2(2),z2(2)];
p6 = [x2(3),y2(3),z2(3)];

normal1 = cross(p1'-p2', p1'-p3');
normal2 = cross(p4'-p5', p4'-p6');

rot1 = vrrotvec(normal1,[0 0 1]);
rot2 = vrrotvec(normal2,[0 0 1]);

tri1 = [p1' p2' p3'];
tri2 = [p4' p5' p6'];

if mod(rot1(4),pi)  == 0;
    rot_matrix1 = eye(3);
else rot_matrix1 = R3d(rot1(4),rot1(1:3));
end;

if mod(rot2(4),pi) == 0;
    rot_matrix2 = eye(3);
else rot_matrix2 = R3d(rot2(4),rot2(1:3));
end;

tri1_t = rot_matrix1*tri1;
```

```
tri2_t = rot_matrix2*tri2;

tform = maketform('affine',tri2_t(1:2,:)',tri1_t(1:2,:)');
[a,b,c] = svd(tform.tdata.Tinv(1:2,1:2)');
e1 = b(1,1);
e2 = b(2,2);
strain_mises = sqrt(0.5*((e1-e2)^2+e1^2+e2^2))-1;
%strain_mises = sqrt(0.5*((abs(e1-1)-abs(e2-1))^2+(e1-1)^2+(e2-1)^2));
%strain_mises = 1/3*sqrt(2*((e1-e2)^2+e1^2+e2^2));
%strain_mises = 2/3*sqrt(3/2*(e1^2+e2^2));
%strain_mises = sqrt(2/3*(e1^2+e2^2));
```

plot_strain_field.m

```
function plot_strain_field(dt_final,x1,y1,z1,x2,y2,z2)
% Plots minimum and maximum strain vectors on one triangular element

numtri = size(dt_final,1);
trisurf(dt_final,x2,y2,z2,'facecolor','white','edgecolor','black')
hold on;
title('Strain Field');
for i = 1:numtri
    v = dt_final(i,:);
    v1 = v(1); v2 = v(2); v3 = v(3);

    p1 = [x1(v1),y1(v1),z1(v1)];
    p2 = [x1(v2),y1(v2),z1(v2)];
    p3 = [x1(v3),y1(v3),z1(v3)];
    p4 = [x2(v1),y2(v1),z2(v1)];
    p5 = [x2(v2),y2(v2),z2(v2)];
    p6 = [x2(v3),y2(v3),z2(v3)];

    normal1 = cross(p1'-p2', p1'-p3');
    normal2 = cross(p4'-p5', p4'-p6');

    rot1 = vrrotvec(normal1,[0 0 1]);
    rot2 = vrrotvec(normal2,[0 0 1]);

    tri1 = [p1' p2' p3'];
    tri2 = [p4' p5' p6'];

    if mod(rot1(4),pi) == 0;
        rot_matrix1 = eye(3);
    else rot_matrix1 = R3d(rot1(4),rot1(1:3));
    end;

    if mod(rot2(4),pi) == 0;
        rot_matrix2 = eye(3);
        inv_r = eye(3);
    else rot_matrix2 = R3d(rot2(4),rot2(1:3));
        inv_r = R3d(-rot2(4),rot2(1:3));
    end;

    tri1_t = rot_matrix1*tri1;
    tri2_t = rot_matrix2*tri2;

    tform = maketform('affine',tri2_t(1:2,:)',tri1_t(1:2,:)');
    [a,b,c] = svd(tform.tdata.Tinv(1:2,1:2)');
    e1 = b(1,1);
    e2 = b(2,2);
    strain_mises = sqrt(0.5*((e1-e2)^2+e1^2+e2^2));

    icn = center(tri2_t(:,1),tri2_t(:,2),tri2_t(:,3),'incenter');
    icx = icn(1); icy = icn(2); icz = icn(3);

    % Scale vectors by size of triangle
    f = 5;
```

```
    d = a*b*c;

    princ1 = e1;
    princ2 = e2;

    % Principal direction 1
    % Angle between principal axis 1 and real axis 1
    th1 = atan2(d(2,1),d(1,1))-atan2(a(2,1),a(1,1));
    % Angle between principal axis 2 and real axis 2 ???
    th2 = atan2(d(2,2),d(1,2))-atan2(a(2,1),a(1,1));

    syms h;

    % Calculate new principal lengths (half length)
    j1 = eval((solve(1/(princ1/2)^2*(h*cos(th1))^2+1/(princ2/2)^2*(h*sin(th1))^2-1)));
    j2 = eval((solve(1/(princ1/2)^2*(h*cos(th2))^2+1/(princ2/2)^2*(h*sin(th2))^2-1)));

    % Define strain colors
    if j1(1)>=j2(1)
        c1 = 'red';
        c2 = 'blue';
    else c1 = 'blue';
        c2 = 'red';
    end;

    xf1 = icx+j1(1)*f*cos(atan2(d(2,1),d(1,1)));
    yf1 = icy+j1(1)*f*sin(atan2(d(2,1),d(1,1)));
    xf1a = icx-j1(1)*f*cos(atan2(d(2,1),d(1,1)));
    yf1a = icy-j1(1)*f*sin(atan2(d(2,1),d(1,1)));
    p1_vects = [xf1 xf1a; yf1 yf1a; icz icz];
    p1_vr = inv_r*p1_vects;
    plot3([p1_vr(1,1), p1_vr(1,2)],[p1_vr(2,1), p1_vr(2,2)],[p1_vr(3,1),
p1_vr(3,2)],c1,'LineWidth',2);

    % Principal direction 2
    xf2 = icx+j2(1)*f*cos(atan2(d(2,2),d(1,2)));
    yf2 = icy+j2(1)*f*sin(atan2(d(2,2),d(1,2)));
    xf2a = icx-j2(1)*f*cos(atan2(d(2,2),d(1,2)));
    yf2a = icy-j2(1)*f*sin(atan2(d(2,2),d(1,2)));
    p2_vects = [xf2 xf2a; yf2 yf2a; icz icz];
    p2_vr = inv_r*p2_vects;
    plot3([p2_vr(1,1), p2_vr(1,2)],[p2_vr(2,1), p2_vr(2,2)],[p2_vr(3,1),
p2_vr(3,2)],c2,'LineWidth',2);
    axis equal;
end;
```

plot_minmax_strain.m

```
function plot_minmax_strain(dt_final,x1,y1,z1,x2,y2,z2)
% Plots minimum and maximum strain vectors on one triangular element
global max_principalstrain;
global min_principalstrain;

numtri = size(dt_final,1);

trisurf(dt_final,x2,y2,z2,'facecolor','none','edgecolor','black')
hold on;
title('Principal Strain Field');
for i = 1:numtri
    v = dt_final(i,:);
    v1 = v(1); v2 = v(2); v3 = v(3);

    p1 = [x1(v1),y1(v1),z1(v1)];
    p2 = [x1(v2),y1(v2),z1(v2)];
    p3 = [x1(v3),y1(v3),z1(v3)];
    p4 = [x2(v1),y2(v1),z2(v1)];
    p5 = [x2(v2),y2(v2),z2(v2)];
    p6 = [x2(v3),y2(v3),z2(v3)];
```

```
    normal1 = cross(p1'-p2', p1'-p3');
    normal2 = cross(p4'-p5', p4'-p6');

    rot1 = vrrotvec(normal1,[0 0 1]);
    rot2 = vrrotvec(normal2,[0 0 1]);

    tri1 = [p1' p2' p3'];
    tri2 = [p4' p5' p6'];

    if mod(rot1(4),pi) == 0;
        rot_matrix1 = eye(3);
    else rot_matrix1 = R3d(rot1(4),rot1(1:3));
    end;

    if mod(rot2(4),pi) == 0;
        rot_matrix2 = eye(3);
        inv_r = eye(3);
    else rot_matrix2 = R3d(rot2(4),rot2(1:3));
        inv_r = R3d(-rot2(4),rot2(1:3));
    end;

    tri1_t = rot_matrix1*tri1;
    tri2_t = rot_matrix2*tri2;

    tform = maketform('affine',tri2_t(1:2,:)',tri1_t(1:2,:)');
    [a,b,c] = svd(tform.tdata.Tinv(1:2,1:2)');
    e1 = b(1,1);
    e2 = b(2,2);
    strain_mises = sqrt(0.5*((e1-e2)^2+e1^2+e2^2));

    icn = center(tri2_t(:,1),tri2_t(:,2),tri2_t(:,3),'incenter');

    icx = icn(1); icy = icn(2); icz = icn(3);

    % Scale vectors by size of triangle
    f = 5;

    % Principal direction 1
    princ1 = e1;
    xf1 = icx+1/2*princ1*f*cos(atan2(a(2,1),a(1,1)));
    yf1 = icy+1/2*princ1*f*sin(atan2(a(2,1),a(1,1)));
    xf1a = icx-1/2*princ1*f*cos(atan2(a(2,1),a(1,1)));
    yf1a = icy-1/2*princ1*f*sin(atan2(a(2,1),a(1,1)));

    % Rotate back to original normal on triangle face
    p1_vects = [xf1 xf1a; yf1 yf1a; icz icz];
    p1_vr = inv_r*p1_vects;
    plot3([p1_vr(1,1), p1_vr(1,2)],[p1_vr(2,1), p1_vr(2,2)],[p1_vr(3,1),
p1_vr(3,2)],'red','LineWidth',2);
    % plot3([xf1a, xf1],[yf1a, yf1],[icz, icz],'r');

    % Principal direction 2
    princ2 = e2;
    xf2 = icx+1/2*princ2*f*cos(atan2(a(2,2),a(1,2)));
    yf2 = icy+1/2*princ2*f*sin(atan2(a(2,2),a(1,2)));
    xf2a = icx-1/2*princ2*f*cos(atan2(a(2,2),a(1,2)));
    yf2a = icy-1/2*princ2*f*sin(atan2(a(2,2),a(1,2)));
    %plot3([xf2a, xf2],[yf2a, yf2],[icz, icz],'magenta');
    p2_vects = [xf2 xf2a; yf2 yf2a; icz icz];
    p2_vr = inv_r*p2_vects;
    plot3([p2_vr(1,1), p2_vr(1,2)],[p2_vr(2,1), p2_vr(2,2)],[p2_vr(3,1),
p2_vr(3,2)],'blue','LineWidth',2);

    max_principalstrain(i)=princ1;
    min_principalstrain(i)=princ2;
    axis equal;
end;
```

plot_nonextension.m

```
function plot_nonextension(dt_final,x1,y1,z1,x2,y2,z2)
% Plots minimum and maximum strain vectors on one triangular element
global max_principalstrain;
global min_principalstrain;
global median_length;

numtri = size(dt_final,1);

trisurf(dt_final,x2,y2,z2,'facecolor','white','edgecolor','black')
hold on;
title('Non-Extension Field');
for i = 1:numtri
    v = dt_final(i,:);
    v1 = v(1); v2 = v(2); v3 = v(3);

    p1 = [x1(v1),y1(v1),z1(v1)];
    p2 = [x1(v2),y1(v2),z1(v2)];
    p3 = [x1(v3),y1(v3),z1(v3)];
    p4 = [x2(v1),y2(v1),z2(v1)];
    p5 = [x2(v2),y2(v2),z2(v2)];
    p6 = [x2(v3),y2(v3),z2(v3)];

    normal1 = cross(p1'-p2', p1'-p3');
    normal2 = cross(p4'-p5', p4'-p6');

    rot1 = vrrotvec(normal1,[0 0 1]);
    rot2 = vrrotvec(normal2,[0 0 1]);

    tri1 = [p1' p2' p3'];
    tri2 = [p4' p5' p6'];

    if mod(rot1(4),pi) == 0;
        rot_matrix1 = eye(3);
    else rot_matrix1 = R3d(rot1(4),rot1(1:3));
    end;

    if mod(rot2(4),pi) == 0;
        rot_matrix2 = eye(3);
        inv_r = eye(3);
    else rot_matrix2 = R3d(rot2(4),rot2(1:3));
        inv_r = R3d(-rot2(4),rot2(1:3));
    end;

    tri1_t = rot_matrix1*tri1;
    tri2_t = rot_matrix2*tri2;

    tform = maketform('affine',tri2_t(1:2,:)',tri1_t(1:2,:)');
    [a,b,c] = svd(tform.tdata.Tinv(1:2,1:2)');
    e1 = b(1,1);
    e2 = b(2,2);
    strain_mises = sqrt(0.5*((e1-e2)^2+e1^2+e2^2));

    icn = center(tri2_t(:,1),tri2_t(:,2),tri2_t(:,3),'incenter');
    icx = icn(1); icy = icn(2); icz = icn(3);

    % Scale vectors by size of triangle.  Currently a constant scaling for
    % visibility
    f = median_length*.5;

    d = a*b*c;

    princ1 = e1;
    princ2 = e2;

    % Lines of non extension only exist for a certain criterion of the
    % principal stretches.  P1>1 and P2<1 is the only valid condition.
    if princ1>1 && princ2<1
        syms th;
```

```
        % Find angle of lines of non extension: radius of circle = 0.5
        theta = eval((solve(1/(princ1/2)^2*(0.5*cos(th))^2+1/(princ2/2)^2*(0.5*sin(th))^2-1)));
        % Non extension lines are located at +/- theta and their respective
        % supplements
        th_1 = atan2(a(2,1),a(1,1))+abs(theta(1));
        th_2 = atan2(a(2,1),a(1,1))-abs(theta(1))+pi;
        threshold = 0.05; % Ensure that non extension directions are significant
        if (princ1-princ2)>threshold
            % Plot segments representing the direction of the lines of non
            % extension
            % Non extension direction 1
            xf1 = icx+1/2*f*cos(th_1);
            yf1 = icy+1/2*f*sin(th_1);
            xf1a = icx-1/2*f*cos(th_1);
            yf1a = icy-1/2*f*sin(th_1);

            % Rotate back to original normal on triangle face
            p1_vects = [xf1 xf1a; yf1 yf1a; icz icz];
            p1_vr = inv_r*p1_vects;
            plot3([p1_vr(1,1), p1_vr(1,2)],[p1_vr(2,1), p1_vr(2,2)],[p1_vr(3,1),
p1_vr(3,2)],'blue','LineWidth',2);
            % plot3([xf1a, xf1],[yf1a, yf1],[icz, icz],'r');

            % Non extension direction 2
            xf2 = icx+1/2*f*cos(th_2);
            yf2 = icy+1/2*f*sin(th_2);
            xf2a = icx-1/2*f*cos(th_2);
            yf2a = icy-1/2*f*sin(th_2);
            %plot3([xf2a, xf2],[yf2a, yf2],[icz, icz],'magenta');
            p2_vects = [xf2 xf2a; yf2 yf2a; icz icz];
            p2_vr = inv_r*p2_vects;
            plot3([p2_vr(1,1), p2_vr(1,2)],[p2_vr(2,1), p2_vr(2,2)],[p2_vr(3,1),
p2_vr(3,2)],'blue','LineWidth',2);
        end;
    end;
    axis equal;
end;
```