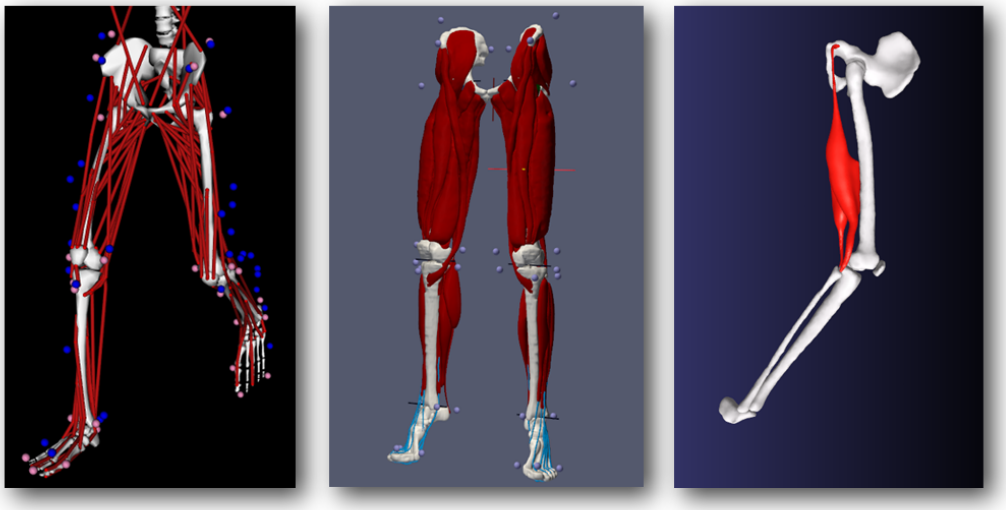
Unified platform for subject-specific neuromuscular and finite element simulations

P.W.A.M. Peeters - Utrecht University



Supervisor: Dr. Nicolas Pronost

Thesis number: ICA-0487953
January 2012

Abstract

Studying human motion using musculoskeletal models is a common practice in the field of biomechanics. By using such models, recorded subject's motions can be analyzed in successive steps from kinematics and dynamics to muscle control. However simulating muscle deformation and interaction is not possible, but other methods such as a *finite element* (FE) simulation are very well suited to simulate deformation and interaction of objects. We present a pipeline that can combine these two, by automatically generating a FE simulation based on subject-specific segmented MRI data, and a motion performed by the same subject. The pipeline resolves several types of data inconsistencies: noise in the dataset is removed by smoothing, objects that contain self-intersecting parts are corrected, missing tendon geometries are generated automatically and overlaps between objects are resolved. Much effort was made to resolve overlaps in a meaningful way of which several methods are discussed. This report shows the different steps of the pipeline, such as solving overlaps in the segmented surfaces, generating the volume mesh and the connection to a musculoskeletal simulation. The pipeline is validated by recreating an experiment done on live subjects where passive hamstring resistance was measured and by comparing experimental results.

Acknowledgments

This work was an effort of two years, beginning in February 2010 in Lausanne at the EPFL and ending here in Utrecht. I would like to thank those that helped me on this journey, first and foremost Dr. Nicolas Pronost, who was my supervisor during my stay in Lausanne and started working at Utrecht University at the same time I returned from Lausanne. I would like to thank Dr. Anders Sandholm, who worked with Dr. Pronost in Lausanne as a PhD and helped me during my stay in Lausanne. Dr. Arjan Egges, who helped me find the project and professors in Lausanne, and who supervised and helped me during the entire course of this project.

Additionally, I would like the attendees of the graduate meetings for their critical views and helpful ideas, especially Arno Kamphuis, Ben van Basten and Thomas Geijtenbeek.

Finally, I would like thank my friends and family for their incredible patience and support through these years.

Table of contents

CHAPTER 1

Introduction

The simulation of human motion using anatomically-based musculoskeletal models is of interest in many fields including computer graphics, biomechanics, motion analysis, medical research and virtual character animation. Many elements of the neuromusculoskeletal system interact to enable coordinated movement. The neuromusculoskeletal system consists of the nervous system, the muscles and the bones of the skeleton which all together enable the body to move. The bones constituting the skeleton are moved by the muscles, which in turn are activated by the brain through the nervous system. If a motion is performed repeatedly the muscle activation will show a pattern. This observed pattern is called an *excitation patterns*. Much research has been done to understand the neuromusculoskeletal system, and so there is a large amount of data describing the mechanics of muscle, the geometric relationships between muscles and bones, and the motions of joints [13, 47]. In the medical field, the neuromuscular system has been studied to get a better understanding of movement disorders in patients with cerebral palsy, stroke, osteoarthritis and Parkinson's disease. Thousands of patients have been studied, recording their neuromuscular excitation patterns both before and after treatment. However, the detailed understanding of the function of each of the elements of the neuromusculoskeletal system remains a major challenge.

Researching these diseases in real-life experiments has the following limitations. Firstly, many important variables are hard to measure in an experiment, such as the force generated by each muscle. Secondly, it is difficult to deduce cause-effect relationships in complex dynamic systems based on experimental data alone. For example, an excitation pattern can be studied, but cannot be changed in a real experiment, so cause-effect relations will always be made on a very high level, not on a individual muscle level. Therefore, understanding the functions of muscles from experiments is not straightforward. For example, electromyographic (EMG) recordings can give an indication of when a muscle is active, but from the EMG alone, one cannot determine the motion of the body. The difficulty arises because a muscle can apply force on joints over which it does not span and it can move body segments to which it does not attach [52].

These problems that arise when analyzing experimental data can be solved for a large part by combining the experimental data with a neuromuscular simulation framework. Neuromuscular simulation allows one to study the different facets of neuromuscular activity, specifically the cause-and-effect relationships between neuromuscular excitation patterns, muscle forces and resulting motion of the body. It can integrate theoretical models describing the anatomy and physiology of the neuromusculoskeletal system and the mechanics of multijoint movement. Simulations also enrich experimental data by providing estimates of important variables such as muscle and joint forces, which are difficult to measure experimentally.

A neuromusculoskeletal simulation is based on several sources of data. The most important one is *pose-based motion* data, which tracks the motion of limbs by means of reflective markers, placed on the body according to a specific protocol. Each marker has a specific place, based on an anatomical landmark. Since the musculoskeletal model defines the same marker positions, each recorded marker position has its corresponding marker within the musculoskeletal model.

Another approach to simulating motion is the use of *finite element analysis* (FEA). Finite element analysis is a method to simulate a complex environment by representing it as a set of finite elements which are interconnected by a means of (differential) equations, which define the properties of the environment. The method of FEA has been applied in many fields, originally in the fields of civil and aeronautic engineering, and over the years has proved itself also as a useful tool in biomechanics research [45]. It enables detailed simulations of complex objects interacting and allows visualization and extraction of important physical variables such as stress and strain on each element. Typical datasources for creating FEA setups are volume scans of subjects, using techniques such as *magnetic resonance imaging* (MRI) or *X-ray computed tomography* (CT). Using such a volumetric dataset of a subject, a detailed FE simulation setup consisting of a set of elements that describe bones, muscles and tendons, can be created. The main problem of these datasets are the inaccuracies which

arise from and are inherent to the data acquisition process. Even the best acquisition methods such as using the visible human dataset need some steps to refine the data [45]. The result of a finite element simulation also gives a motion that needs to be performed and therefore can be used to study the interactions between muscles and bones of the subject. Since FEA offers such an advantage in simulating interaction between muscles and bones over a more high-level type of simulation as neuromusculoskeletal simulation, the question arises if we cannot create a bridge between these types of simulation environments.

The aim of this study was to extend the capabilities of the musculoskeletal platform by using the subject-specific motion generated using the musculoskeletal simulation environment to drive a MRI-based finite element simulation. We designed a pipeline to connect the motion from a musculoskeletal simulation to a finite element simulation that is generated from a subject-specific MRI-dataset. This pipeline allows one to configure a finite element simulation by choosing specific muscles and bones to be in the output simulation, thereby enabling the creation of simulations of any combination of different muscles, bones and tendons within the dataset. In this study, we demonstrate the method on the lower limb with a particular attention to the knee area, where many interactions take place during daily activity. The method described in this report is a step toward the automatic simulation of muscle deformations in virtual humans in a predictive manner through the interaction of the muscle anatomy and function with applications in computer graphics. Similarly, the method can simulate a complex muscle structure so that muscle function can be investigated in bioengineering.

The MRI dataset we used for this study is segmented using existing methods. Because of the low-quality of some MRI volume scans, the segmented MRI data must be preprocessed in a series of sequential steps. This pipeline uses several techniques from the field of geometric algorithms such as smoothing, self-intersect removal and volume mesh generation.

The main contributions of this study to the field of musculoskeletal and FE simulations is the **automated pipeline**, that **resolves data inaccuracies, such as muscle overlaps and self-intersecting muscles**. This work also contributes by providing a **connection between the musculoskeletal motion and the FE simulation**, and thereby creating a unified platform for neuromuscular and finite element simulations.

This report is structured as follows. In the next Chapter, we explain the context in which this research is situated and how this project differs from the most recent developments. In Chapter 3 we present an overview of the main steps in the pipeline we developed, from the input datasources of MRI data and motion capture data to the final FEA simulation. In Chapter 4, we explain the most important part of the pipeline, where we start from the MRI dataset and end with the volume meshes that are used for the FEA simulation. Chapter 5 is dedicated to the problem of resolving of overlaps between two closed surfaces representing elements from the musculoskeletal system, such as bones or muscles. In Chapter 6 we describe how we connect the musculoskeletal motion data with the FEA simulation, and we show an experiment performed with the pipeline presented. Finally, we end with the conclusion in Chapter 7, where we summarize the findings of the study, and suggest further possible research directions.

CHAPTER 2

Related work

In this chapter we give an overview of the research related to the creation of a unified platform for neuromuscular and finite element simulation. First we explain the context of neuromuscular simulation and its limits. Then we describe the existing research done in the field of finite element simulation in biomechanics, specifically the simulation of muscle tissue and the use of versatile subject specific data in both approaches.

2.1 Neuromusculoskeletal simulation

The value of dynamic simulations of movement is broadly recognized. It has been used to study hamstring mechanics and rehabilitation of hamstring injury [46], to study the effects of a surgical change in the musculoskeletal system [13] such as joint replacements [37] and to study muscular coordination of walking [26, 34], jumping [51] and cycling [39].

The biomechanics community has ongoing efforts to create detailed human musculoskeletal models. Although recent models [14,31] provide accurate muscle parameters for the whole body, they by definition do not provide subject-specific geometrical data needed to simulate very detailed muscular models. Moreover, these models use lumped-parameter 1D muscle models that do not account for the various muscle geometries. However, this model lacks information regarding to the force generating properties of the muscles, as we used generic scaled values from [6]. In musculoskeletal representation physiological parameters such as muscle lengths and muscle forces have been of primary interest, and the realistic visualization has played a secondary role. The muscle paths have been represented using a series of points connected by line segments validated against image data or cadaveric experiments [6]. The insights into muscle functioning gained from these models have helped to improve diagnosis and treatment of people with movement disorders. The biomechanical models that estimate the distribution of stresses, kinematics of joint elements during various postures, postural transitions and physical activities can provide significant insight into the underlying mechanisms of a joint pathology and give an objective evaluation of its function [49].

Over the years, neuromuscular simulation has evolved from a fragmented community where each research group developed their own simulation software into a more collaborative environment. This has mostly been the result of the efforts by the creators of the OpenSim platform [14], by providing an open-source platform that lets the user develop a wide variety of musculoskeletal models that can be shared due to the open nature of the software. The simplest model of the knee joint is a single rotation about a stationary axis in the sagittal plane. But more complex models exist, such as the model of Walker et al. [47], and Yamaguchi et al. [48]. These models have already been implemented in OpenSim musculoskeletal models respectively by Arnold et al. and Delp et al. [6, 13]. For the experiment presented in this work we used the model from Walker et al. In this knee-joint model, the configuration is parametrized by the flexion-extension angle. Dependent on this angle are two translational degrees of freedom, anterior-posterior and inferior-superior, specified by two *natural cubic splines*. The knee-model also specifies a slight rotation around the other two axis, each defined by their own natural cubic spline.

OpenSim can process a marker-based motion file and apply inverse kinematics to fit the marker-based motion to the desired Opensim model poses, where the joints are represented as rotations and local translations. The orientation of the femur and the tibia in the FES is defined by the pose based motion from OpenSim.

2.2 Finite element simulation in biomechanics

Many attempts have been made at simulating muscles in high detail using the method of finite element simulation [17, 23, 25], for example, to investigate intramuscular pressures [22]. It has also been used to study the significance of myofascial force transmission which is relevant for the study of muscular dystrophies [50].

Teran et al. have designed a framework for extracting and simulating musculoskeletal geometry from the visible human dataset [5, 45]. The visible human dataset consists of high resolution images of millimeter-spaced cross sections of an adult human male. The visual human dataset was obtained by making cryosections at 0.174mm intervals and photographed at a resolution of 1056 x 1528 pixels. The study used a motion from a different subject, since the visual human dataset is obtained from a deceased subject and no motion capture has been previously performed. The same dataset is used in [16], where the dataset is used to create a 3D model of the human leg, specifically for visualization of deformations and incorporates also the rendering of muscle fibers using textures. In [45] the segmentation of this data was performed by creating a level set representation of each tissue relevant for the simulation. The signed distance function required for the level set procedures and to generate the triangulated surface that was used is the fast marching method [42]. They use slice-by-slice contour sculpting to repair problem regions. First they manually examine each slice visually to check for and eliminate errors. Level-set smoothing techniques are applied afterwards, such as motion by mean curvature [35] to eliminate any further noise. In this project, we used MRI data of a much lower resolution than the visual human dataset (See Section 4.1). Also, our segmentation process is automatic and therefore needs a more rigorous repair process.

Blemker and Delp used an MRI dataset to create simulations of several sets of muscles to study the variation in moment arms across fibers within a muscle [9] and to predict rectus femoris and vastus intermedius fiber excursions [10]. They used MRI of live and cadaver specimen, and manually segmented the areas of interest. They also created a fiber map for each muscle of interest, based on template fiber geometries morphed to each muscles target fiber geometry. They use a manual segmentation process instead of an automatic segmentation as is used in this work. While this method of segmentation provides a surface mesh of higher quality, the method is not automatic.

The segmentation method used in this project comes from Schmid et al. [41]. This method is based on an earlier work of Gilles et al. [18], who present a registration and segmentation method for clinical MRI datasets based on discrete deformable models. It uses a force-based optimization technique where each goal is defined as a force. Gilles uses forces on the medial axis (MA), where the forces consist of shape and smoothing constraints, non-penetration constraints and external forces of intensity profiles. Schmid extends this method with shape priors in the form of a principal component analysis (PCA) of global shape variations and a Markov random field (MRF) of local deformations that impose additional spatial restrictions in shape evolution. Unfortunately, since the method is dependent on forces, balancing the weights of the non-penetration constraints and the other constraint can be a difficult if not impossible task. Therefore, the resulting surfaces suffer from intersections between surfaces and also self-intersections, which we resolve in this study. Since the density of the vertices of the shape priors used by the method are uniformly spread, the resulting shapes have nearly the same property. The segmentation algorithm also includes tendon and attachment specifications. These are specified by vertex indices in the resulting muscle meshes.

We used the software package FEBio developed at the University of Utah [3, 27] for solving the FE simulation. FEBio is a nonlinear finite element solver that is specifically designed for biomechanical applications. Since it does not provide mesh generation facilities, our pipeline creates the mesh and a complete input file with which the FEBio solver can use without further necessary configuration. Besides the FE solver, the developers of the FEBio software also provide a viewer for FEBio input and output files. The latter can be used to make detailed analyses of results of the simulations, such as visualizations of pressure and stress.

2.3 Geometric algorithms

During this research we had to apply a number of methods from the field of surface mesh manipulation.

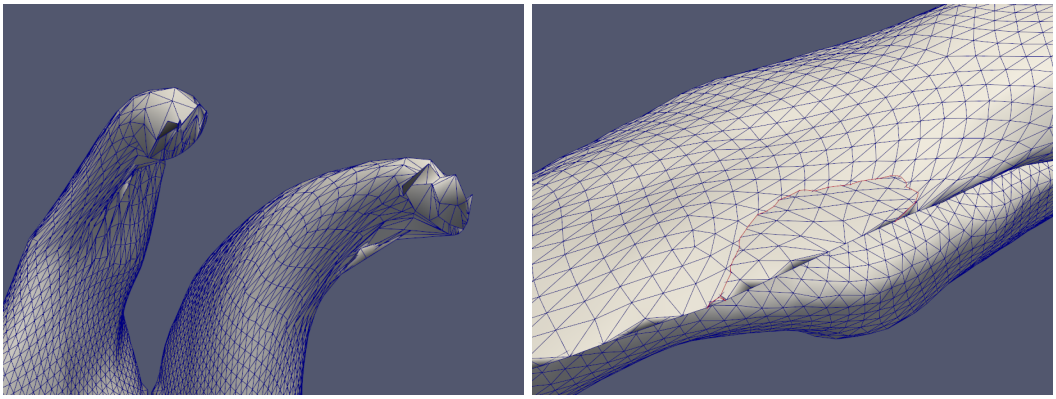


Figure 2.1: Left: Segmentation artifacts: the vertices of the gastrocnemius muscle are not ordered in a smoothly fashion. Right: Self intersection in the vastus intermedius muscle. The red line indicates the border of the intersection.

2.3.1 Surface smoothing

First, we had to apply smoothing methods to remove artifacts from the segmentation algorithm (see Figure 2.1). Taubin introduced a surface smoothing method that prevents shrinking of the objects [43], by iteratively applying a Gaussian filter alternating a reverse growing step that does not include the just erased low-frequencies. In the same year he proposed to view surface smoothing as a signal processing problem [44], so the smoothing becomes an application of Fourier analysis, since the classical Fourier transform of a signal can been seen as the linear combination of the eigenvectors of the Laplacian operator. By defining a new operator taking the place of the Laplacian the Fourier analysis can be extended to surfaces of arbitrary topology. Desburn et al. extended this idea by formulating the Laplacian smoothing algorithm as time integration of the heat equation [15], which leads to an implicit integration scheme. This approach resolves some problems with the uniform approximations of the Laplacian by Taubin when applied to irregular connectivity meshes, such as geometric distortion, numerical instability and slow convergence for large meshes. In this work we applied the method of Taubin, since it is fitted to our type of surfaces, which are closed surfaces with a very uniform vertex distribution. The method of Taubin [43] is the most suitable option with regards to implementation complexity.

Smoothing without shrinkage algorithm

For this work, we implemented the smoothing technique of Taubin [43]. The method consists of iteratively applying a Gaussian filter alternated with a reverse growing step that does not include the just erased low-frequencies. Below we will explain the algorithm in detail.

Surfaces are represented as a list of vertices $V = \{v_i : 1 \leq i \leq n_V\}$ and a list of faces $F = \{f_k : 1 \leq k \leq n_F\}$ each face $f_k = (i_1^k, \ldots, i_{n_{f_k}}^k)$ consisting of a sequence of indices in the vertex list. A surface $S = \{V, F\}$ is a pair of one vertex list combined with a face list. In this work, all surfaces are consisting of only triangles, so the faces always have three vertices $f_k = (i_1^k, i_2^k, i_3^k)$.

A neighborhood of a vertex v_i is a set v_i^{av} of indices of vertices. If the index j belongs to the neighborhood $i*$ we say that v_j is a neighbor of v_i. The neighborhood structure of a shape is defined as the set of all the neighborhoods $v_i^{av} : i = 1, 2, \ldots, n_V$. In our implementation of the smoothing algorithm we use the first order neighborhood, wherein two vertices are neighbors if they are both present in the same face $v_i^{av} = \{j : j \in f_k, i \in f_k\}$.

In the Gaussian smoothing algorithm the position of each vertex is replaced by a weighted combination of the positions of itself and its neighbors. Alternatively, Gaussian smoothing can also be reformulated as follows. First, for each vertex v_i, a vector average

$$\Delta v_i = \sum_{j \in v_i^{av}} w_{ij}(v_j - v_i)$$

is computed as a weighted average of the vectors $v_j - v_i$, that extends from the current vertex to a neighbor vertex v_j. For each vertex v_i the weights w_{ij} are positive and add up to one, but otherwise they can be chosen in many different ways. The most obvious choice that produces good results is

to set w_{ij} equal to the inverse of the number of neighbors $1/|v_i^{av}|$. Once all the vector averages are computed, the vertices are updated by adding to each vertex current position v_i its corresponding displacement vector

$$v_i' = v_i + \lambda \Delta v_i$$

computed as the product of the vector average Δv_i and the scale factor λ, obtaining the new position v_i'. The scale factor, which can be a common value for all the vertices is a positive number $0 < \lambda < 1$.

The advantage of the Gaussian smoothing method is that it produces geometric smoothing. The main disadvantage is that to produce significant smoothing, the Gaussian smoothing algorithm must be applied iteratively a large number of times using first order neighborhoods. However, by doing so a significant shrinkage effect is also introduced.

This can be overcome if we apply the extension on the Gaussian smoothing algorithm developed by Taubin [43]. After the first smoothing step with a positive scale factor λ, we apply a second smoothing step but with a negative scale factor μ greater in magnitude than the first scale factor ($0 < \lambda < -\mu$). To produce a significant smoothing effect, these two steps must be repeated, alternating the positive and negative scale factors a number of times. This method produces a low pass filter effect, where surface curvature takes the place of frequency.

2.3.2 Self intersections

The surface data provided by the segmentation algorithm also contained self-intersections, as can be seen in Figure 2.1. We removed these using the algorithm proposed by Jung, Shin and Choi [24], who designed it originally to remove self-intersections from a raw offset triangular mesh. The algorithm uses a region growing approach, keeping a list of valid triangles. Starting with an initial seed triangle, the algorithm grows the valid region to neighboring triangles until it reaches triangles with self-intersection. Then the region growing process crosses over the self-intersection and moves to the adjacent valid triangle. Therefore the region growing traverses valid triangles and intersecting triangles adjacent to valid triangles only. We chose to use this method, since it is the only work that is specifically solving self-intersection problems.

2.3.3 Self intersect removal algorithm

The method of Jung is a region growing algorithm. Starting with a seed-triangle that is known to be valid, the valid region is grown in all directions until an intersecting triangle is encountered. This triangle is split into several sub-triangles and the growing continues on this sub-triangle level, until a crossing is found. On the crossing, the corresponding sub-triangle of the intersecting triangle is marked as valid and the growing continues.

In Figure 2.2 an overview of the self-intersection removal algorithm is shown. The method requires the surface to be void of degenerate triangles, so the first step is to resolve those. Then, the intersecting triangles of the surface mesh are identified. A seed triangle has to be determined after which the region growing can start. As a final step, the excess triangles are discarded and the new surface datastructure is constructed.

During the process, triangles are classified into three groups: valid triangles, invalid triangles, and partially valid triangles. Valid triangles are those to be entirely contained in the valid region and remain in the mesh after the self-intersection removal. Invalid triangles are the ones that are to be deleted entirely. Partially valid triangles lie on the boundary between a valid region and an invalid region. A partially valid triangle has intersections with other triangles, and a portion of a partially valid triangle is to be included in the resulting mesh. A partially valid triangle needs to be split into sub-triangles and these sub-triangles should again be classified into valid and invalid sub-triangles.

Remove degenerate triangles

A degenerate triangle is a triangle with (nearly) zero area. Triangles with edges with a length $l < \varepsilon_e$ (zero length tolerance) and triangles with a minimum angle $\alpha < \varepsilon_\alpha$ (zero angle tolerance) are classified as degenerate triangles and are resolved by an edge collapse as in [21] and swapping diagonal edges, respectively, as shown in Figure 4.6 on page 27.

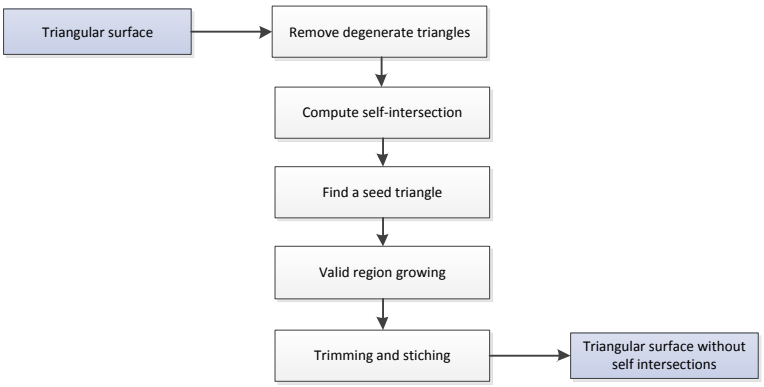


Figure 2.2: Overview of the self-intersection removal algorithm.

Computing self-intersections

To avoid computing intersections between all triangles, we use a bucket structure for reducing the number of triangle-triangle intersection (TTI) tests. The bucket structure partitions the input surface mesh into buckets, where each bucket contains less than a fixed number C of triangles.

Constructing the bucket structure starts with a single bucket containing all triangles. If the number of triangles in a bucket is larger than C, the bucket is subdivided into two by the plane splitting the longest side of its AABB (axis aligned bounding box). Triangles crossing the bucket plane are stored in both of the buckets. The bucket subdivision process is applied recursively until each bucket contains less than C triangles or no improvement can be made.

For each bucket, we simply compare all pairs of triangles within a bucket. For the fast TTI test we use the 'interval overlap method' suggested by Möller [32]. Each intersection segment stores pointers to both the participating triangles and each triangle maintains a list of intersection segments that belong to it.

Finding a seed triangle

A seed triangle is a valid triangle used to initiate the valid region growing. Let $VC \subset V$ be the set of vertices on the convex hull of V. If the input surface is a raw offset triangular mesh, VC belongs to the valid region. Any triangle $f \in F$ having at least one vertex in VC is valid or partially valid and can serve as the seed triangle.

Valid region growing

The algorithm for valid region growing is composed of the following steps:

1. Each triangle has one of three states: unvisited, valid or partially valid. Initially, all triangles are marked as unvisited.
2. The seed triangle is marked as valid and inserted into W, the set of wavefront triangles
3. If W is empty, go to 5, otherwise remove f_k from W
4. For each unvisited triangle f_l adjacent to f_k, if f_l has no intersections, it is marked as valid and inserted into S. Otherwise, f_l is marked as partially valid and inserted into P, the set of partially valid triangles encountered. For each f_l, the entrance edge e_p, which is the edge shared by f_k and f_l, is also saved.
5. If P is empty, go to step 7. Otherwise, remove a partially valid triangle f_p from P.
6. Sub-triangulate f_p, and grow the region over f_p and its counterpart triangles. If another seed triangle is found, go to 2, otherwise go to 4.
7. The valid region growing step is completed.

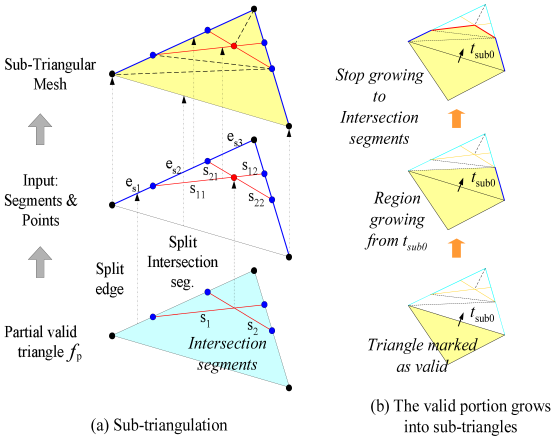


Figure 2.3: Sub-triangulation and valid region growing in sub-triangular mesh (Image from [24].)

Sub-triangulation

A partially valid triangle f_p has intersection segments in it and needs to be sub-triangulated. We now need to perform two tasks: (1) to split f_p into sub-triangles that contain the intersection segments as their edges, as seen in Figure 2.3(a), (2) to propagate the valid region within the sub-triangular mesh, as in Figure 2.3(b).

The steps in detail for the first task are as follows:

1. Split each edge e_u of f_p by all intersection segments s_i.

2. Split each s_i at the intersection points among them.

3. Sub-triangulate f_p by 2D constrained Delaunay triangulation together with the edges and intersection segments that were split in step 1 and 2

As shown in Figure 2.3(b), the valid region growing in the sub-triangular mesh starts from the entrance edge e_p. We will denote the sub-triangles as $t_{\#}$. The sub-triangle t_{sub0} which is adjacent to e_p is marked as valid and becomes the seed for the valid region growing in the sub-triangular mesh. Then, the valid part of f_p grows into neighboring sub-triangles until it reaches intersection segments, which play the role of the entrance edge for the counterpart (partially valid) triangle f_c in the next step.

Crossing the river

The region growing process crosses over the self-intersection and moves to the sub-triangles of the counterpart triangles. Figure 2.4 illustrates the detailed steps of propagating the valid region into the sub-triangles of the counterpart triangle across the intersection of a partially valid triangle. This process starts by sub-triangulating the counterpart triangle f_c as in Section 2.3.3. In Figure 2.4(b), there are two sub-triangles t_3 and t_4 of f_c adjacent to the previously found entrance edge. (Note that t_1 and t_2 are sub-triangles of f_p.) By considering the normal vector orientation compatibility with f_p, the sub-triangle t_4 is selected as valid one, which serves as the valid seed triangle in the region growing within the sub-triangular mesh of f_c. Eventually, as is shown in Figure 2.4(c), f_v is found as a valid triangle and is inserted into W.

Trimming and stitching

Since all partially valid triangles are replaced by sub-triangles and all valid triangles are marked, the trimming and stitching can be done very simply. The trimming step is to retain valid triangles only and to remove invalid ones. The next step is to stitch the self-intersection triangles together by assigning topological relation between adjacent valid triangles.

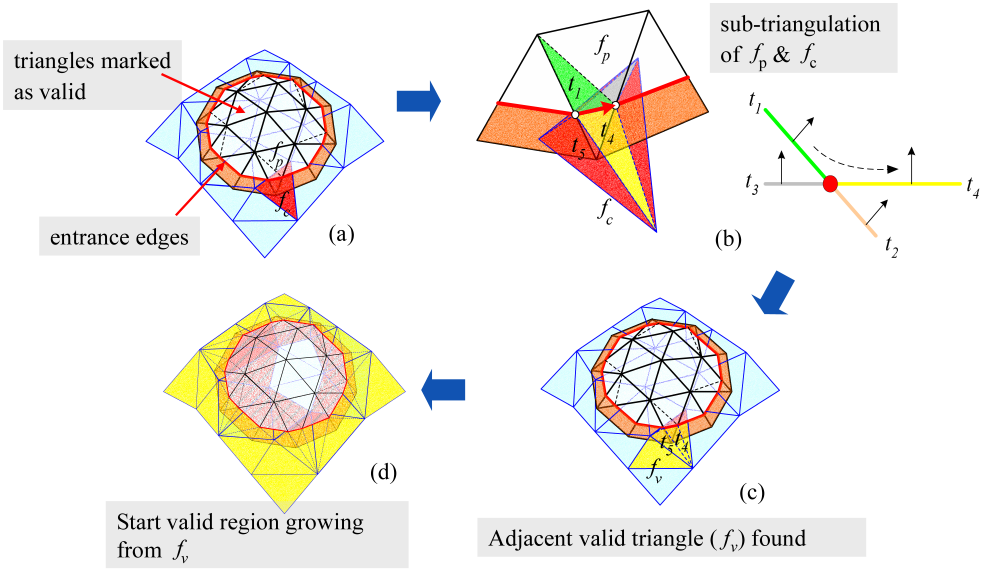


Figure 2.4: Detailed steps of crossing the river (Image from [24].)

2.3.4 Surface intersection removal

The surfaces resulting from the segmentation algorithm contain surfaces that intersect each other. In the final pipeline we used a boolean operator implemented in the Carve constructive solid geometry library (CSG) [1], since it is the main freely available CSG library. The boolean operators are implemented using the concept of Nef polyhedra [19, 33].

During this research, we tried several methods of approaching the overlap problem, which in the end were not sufficient. One of the promising ones was using a shrink-based method. The method needs an algorithm that uniformly shrinks an object, so unfortunately a simple Laplacian smoothing applied iteratively does not suffice, since it grows the surface in some areas [15]. A seemingly promising way of uniform shrinking was a skeleton based extraction method, based on the work of Au et al. [7]. The method contracts the mesh geometry into a zero-volume skeletal shape by applying implicit Laplacian smoothing with global positional constraints. The contracted mesh is then converted into a 1D curve-skeleton by removing all the collapsed faces while preserving the shape of the contracted mesh and the original topology. The application of the algorithm for this study is described in Section 5.2.2.

2.3.5 CGAL

For the FE simulation, we need to provide a 3D volume mesh to the FEBio software. To generate the volume meshes from the surface meshes, we use the 3D mesh generation algorithm from the CGAL library [2, 36, 40].

The mesh generation algorithm allow generating meshes of configurable volume density and surface density. Therefore, the vertex indices specified by the segmentation results need to be saved in a way invariant to vertex ordering. We do this by generating attachment and tendon area's by applying the convex hull algorithm from CGAL [20].

2.4 Research objectives

The main contributions of this study to the field of musculoskeletal and FE simulations is the **automated pipeline**, that resolves data inaccuracies, such as **muscle overlaps and self-intersecting muscles** from an **MRI dataset**. This is achieved by applying and adapting several techniques from the field of geometric algorithms, such as surface mesh smoothing and a Boolean difference method. This work also contributes by providing a **connection between the musculoskeletal**

motion and the FE simulation, and thereby creating a **unified platform for neuromuscular and finite element simulations**.

CHAPTER 3

Pipeline overview

In this chapter we will give an overview of the automated pipeline developed to generate the FE simulation. Figure 3.1 shows the overview of the pipeline. If we look at the schematic pipeline, it takes a human subject as 'input'. The subject is recorded in two ways: a scan is made in the MRI scanner and a motion is recorded using optical markers.

The marker motion recorded by the motion capture equipment is imported into a musculoskeletal simulation platform, in our case OpenSim. The platform has a musculoskeletal model that is scaled to the subject using positions of anatomical landmarks (markers). The motion of the markers is converted to an angle based scaled musculoskeletal model by applying inverse kinematics (See Section 6.1).

The MRI scan is a volume scan of the legs of the subject. The leg is segmented using the algorithm of Schmid and Magnenat-Thalmann [41]. The segmentation algorithm produces closed surfaces of muscles and bones and the attachment sites where the muscle connects to the bones.

The closed surfaces from the segmentation result cannot be converted directly to volume meshes. The segmented data contains many segmentation artifacts and noise (See Sections 4.3, 4.6 and 4.7). The data first has to be cleaned before it can be given to the mesh generator. The most important step here is the resolving of overlaps between meshes.

The next step is to generate the volume meshes from the cleaned-up surfaces. The volume meshes, together with the motion from the musculoskeletal model are combined into a finite element simulation. The motion from the musculoskeletal model is converted into the coordinate frame of the MRI dataset (See Section 6.1.2). Finally, all generated volume meshes, material specifications, motion data and attachment specifications are combined into one finite element setup file, that can be read by the finite element solver which produces the final output of the pipeline (See Section 6.2.2).

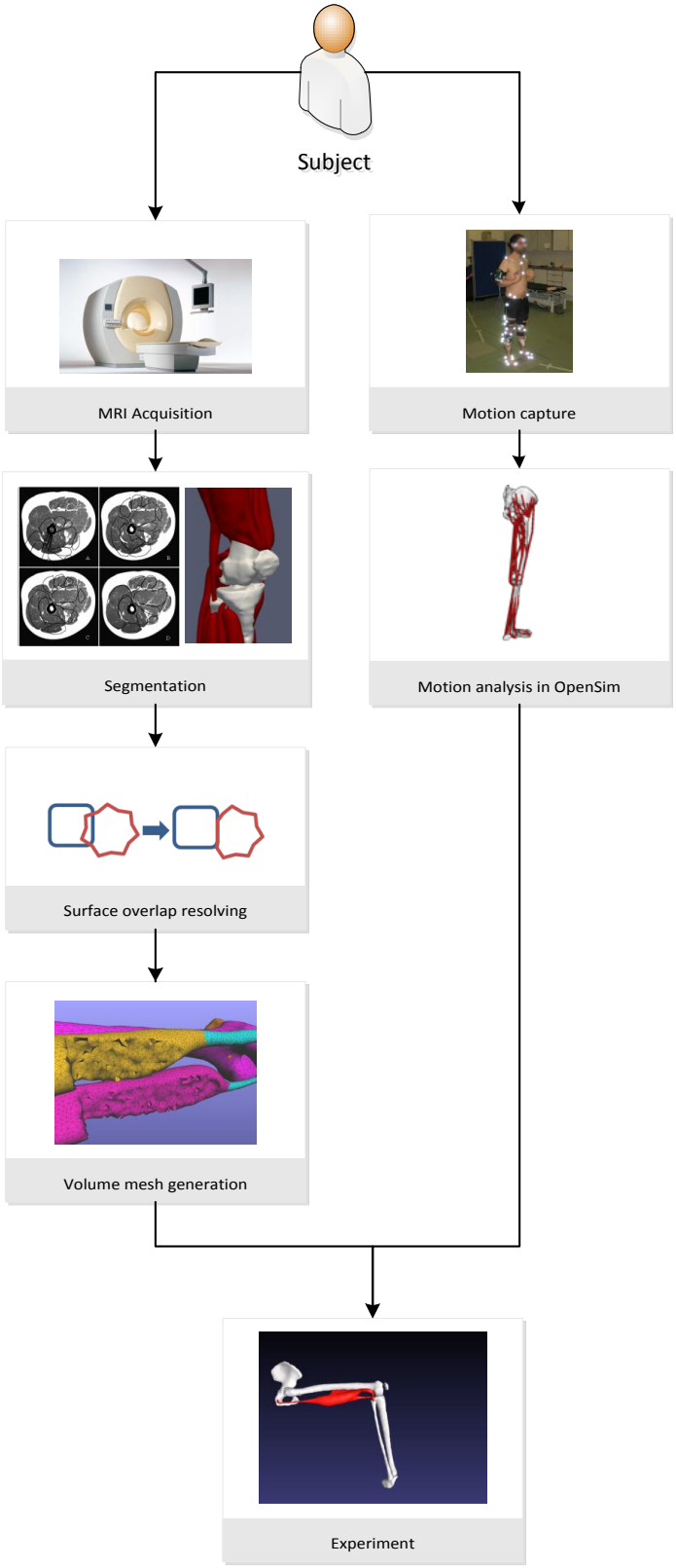


Figure 3.1: Overview of the pipeline.

CHAPTER 4

MRI to volume mesh

In this chapter we will describe the full process of converting an MRI scan to the volume mesh needed for the final finite element mesh setup. First we will describe how the segmentation algorithm produces the surface data we are using in Section 4.1. In Section 4.2 we give an overview of the different stages the segmented MRI data traverses in the rest of the pipeline to obtain the volume meshes.

4.1 MRI segmentation

A subject is scanned in a lying resting pose with an MRI scanner (1.5T Philips Medical Systems). In close collaboration with radiologists, a protocol for the imaging of soft and bony tissues was defined: Axial 2D T1 Turbo Spin Echo, TR/TE = 578/18ms, FOV/FA = 40cm/90°, matrix/resolution = 512x512/0.78x0.78mm, thickness: 2mm(near joints) to 10mm (long bones).

The resulting dataset is segmented into surfaces meshes using a template method that uses a minimal energy optimization to fit a template muscle or bone to the acquired MRI data [41]. This dataset of surface meshes will be the input to our pipeline. The segmentation algorithm is based on discrete deformable models. It uses a force-based optimization technique where each goal is defined as a force. It uses forces on the medial axis (MA), where the forces consist of shape and smoothing constraints, non-penetration constraints and external forces of intensity profiles. Shape priors in the form of a principal component analysis (PCA) of global shape variations and a Markov random field (MRF) of local deformations impose additional spatial restrictions in shape evolution. Unfortunately, since the method is dependent on forces, balancing the weights of the non-penetration constraints and the other constraints can be a difficult if not impossible task. Therefore, the resulting surfaces suffer from intersections between surfaces and also self-intersections. Since the density of the vertices of the shape priors used by the method are uniformly spread, the resulting shapes have nearly the same property. The segmentation algorithm also includes tendon and attachment specification and are specified by vertex indices in the resulting muscle meshes.

4.1.1 Bones and muscles

The segmentation algorithm produces surfaces of bones and muscles. Surfaces are represented as a list of vertices $V = \{v_i : 1 \leq i \leq n_V\}$ and a list of faces $F = \{f_k : 1 \leq k \leq n_F\}$, each face $f_k = (i_1^k, \ldots, i_{n_{f_k}}^k)$ consisting of a sequence of indices in the vertex list. A surface $S = \{V, F\}$ is a pair of a vertex list combined with a face list. All surfaces are consisting of only triangles, so each face always has three vertices $f_k = (i_1^k, i_2^k, i_3^k)$.

The segmentation result generally puts each muscle and bone in a separate file. The tibia and fibula bones are put together into one file. The gastrocnemius muscle has two heads and is divided into two surfaces in the same file.

4.1.2 Attachments and tendons

An attachment site A is defined in the segmentation result as indices in a corresponding vertex list V in the order they are defined in the muscle files.

$$A = \{i_1, \ldots, i_{n_A}\} \tag{4.1}$$

Because the stages in the pipeline change the amount and ordering of the vertices in the datafiles, we process the attachment sites to be index invariant by defining the attachment areas by geometry. This is explained in more detail in Section 4.9. The tendons are defined in the same way as the attachments, and for those we also need to convert the indices to a geometric representation.

4.2 Stage overview

The process of transforming the segmented MRI data to the volume mesh needed for the FE simulation is divided into several stages. Figure 4.1 on page 23 shows an overview of all the stages. The following sections explain each stage in detail. We first start by smoothing the raw surface data to remove segmentation artifacts (Section 4.3). This does not change the mesh topology, so we can use the data to feed the convex hull generation step. The next step is to remove the unwanted components from each surface (Section 4.4), which will remove objects from a surface file, so the vertex order in the files can be changed. Next, new tendons are generated for muscles that are missing tendons (Section 4.5), which adds new geometry to the surface meshes. Then, the self-intersecting parts from the surfaces are removed (Section 4.6). Surfaces that overlap will be separated next (Section 4.7). Then the volume meshes are generated and a small cleanup step is executed (Section 4.8). Finally, the attachments and tendon specifications are transformed into convex hulls and finally to indices pointing to the volume mesh data (Section 4.9).

4.3 Smoothing

The first stage of the algorithm is a smoothing procedure to remove segmentation artifacts. Figure 4.2, Left shows the noise that can result from the segmentation step, as explained in Section 4.1. We implemented a common smoothing technique that does not introduce shrinking [43]. Section 2.3.1 describes the algorithm in detail.

The algorithm is a low pass filter, with three parameters λ, μ and the iteration count N. The low pass filter properties are the pass-band frequency k_{PB}, the pass-band ripple δ_{PB}, the stop-band frequency k_{SB}, and the stop band ripple δ_{SB}. The parameters are related to the low pass filter properties as follows:

$$\lambda < \frac{1}{k_{SB}} \tag{4.2}$$

$$\frac{1}{\lambda} + \frac{1}{\mu} = k_{PB} \tag{4.3}$$

$$\left[(\lambda - \mu)^2/(-4\lambda\mu)\right]^N < 1 + \delta_{PB} \tag{4.4}$$

$$\left[(1 - \lambda k_{SB})(1 - \mu k_{SB})\right]^N < \delta_{SB} \tag{4.5}$$

The surfaces in our dataset are smoothed with these parameters: a $\lambda = 0.33$, $\mu = -0.331$ and $N = 1000$. Figure 4.2 shows the result of the operation on the gastrocnemius muscle using these parameters.

4.4 Removing unwanted components

Some muscles in the segmentation result dataset contain tendon geometries for tendons. In general they are part of the muscle surface itself, but in some cases, they are defined as a separate surface. In Figure 4.3, we can see the different ways of representing tendons. The tendons specified as separate objects unfortunately do not have a corresponding attachment description. Therefore, we remove the tendon from the muscle in these cases. Because these tendons are important for the workings of the muscle, an alternative tendon structure can be generated, as is explained in Section 4.5.

If a tendon is separately specified, it is still defined in the same datafile, sharing the same vertex and face lists $S = \{V, F\}$ (as explained in Section 4.3). The connectivity of the geometry is defined by the faces. From the faces, we can derive a set of edges E, which consist of all the pairs of vertices that occur in the same face.

$$E = \{(i_1, i_2) : f_k \in F, i_1 \in f_k, i_2 \in f_k\} \tag{4.6}$$

We separate the different objects in a datafile by applying a union-find data structure. For each vertex v_i a we *create* a set in the union-find data structure using its index as the set-id. For each edge, we *find* the two sets they belong to, and perform a *union* operation on the two sets. After we add all edges to the union-find datastructure, we can use a *find* operation to get the set-id that each vertex belongs to.

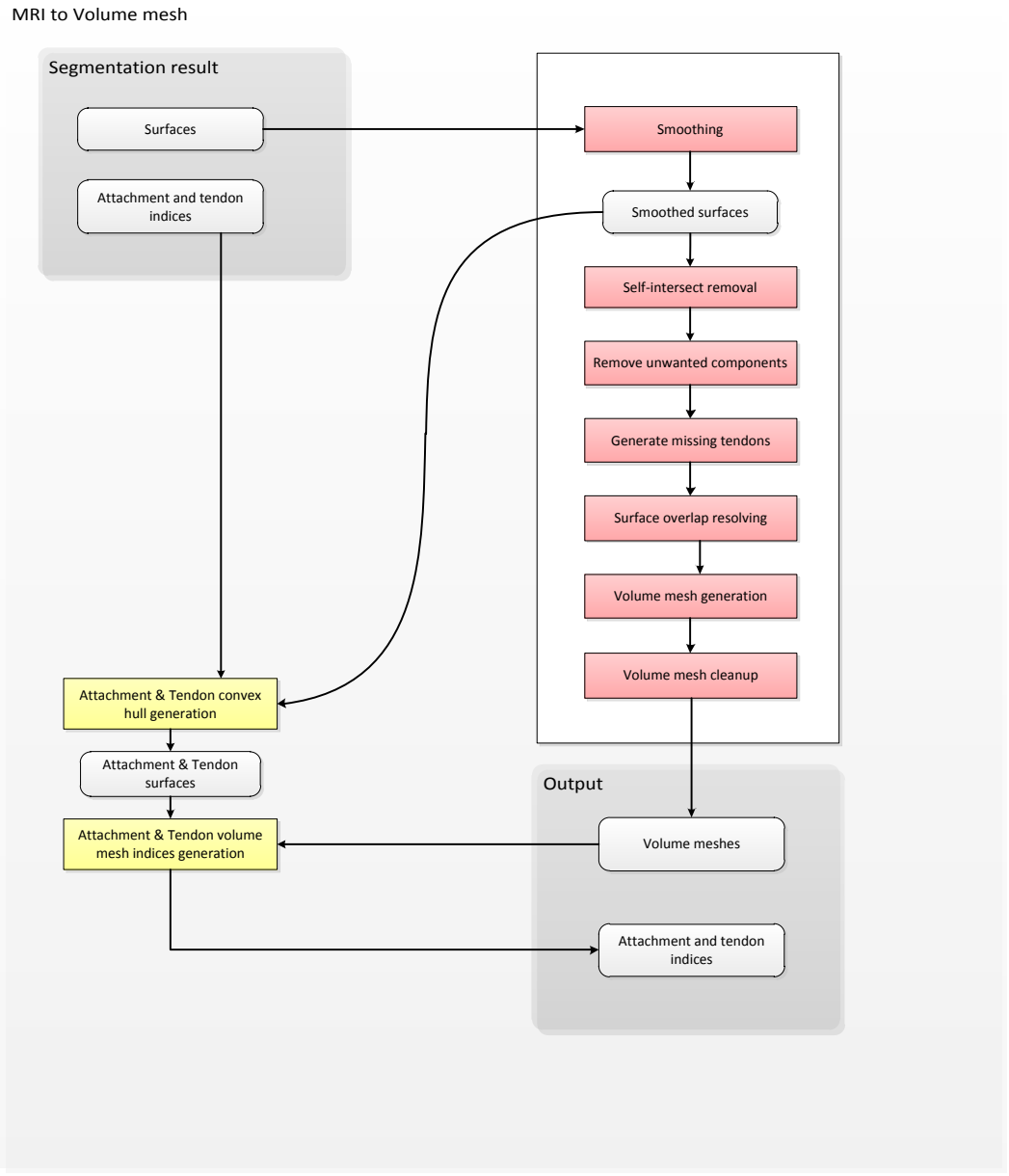


Figure 4.1: Overview of the stages within from MRI segmentation to volume mesh output.

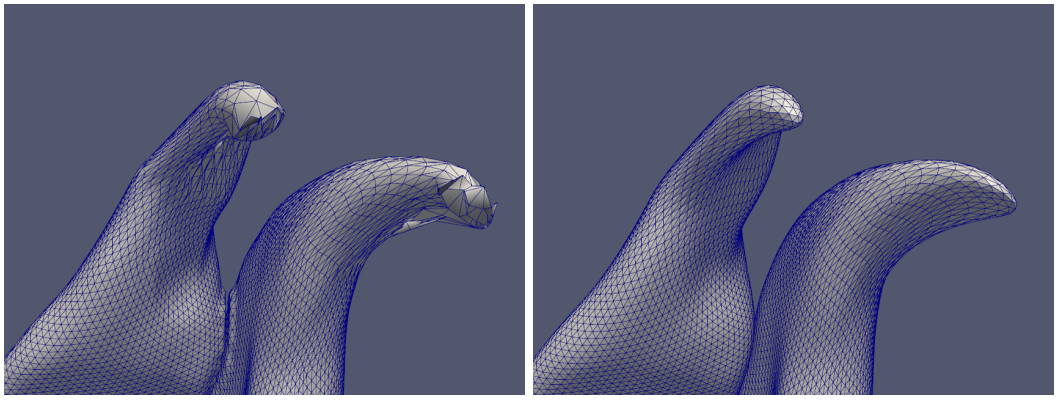


Figure 4.2: Smoothing results. Left is before and right is after the smoothing operation.

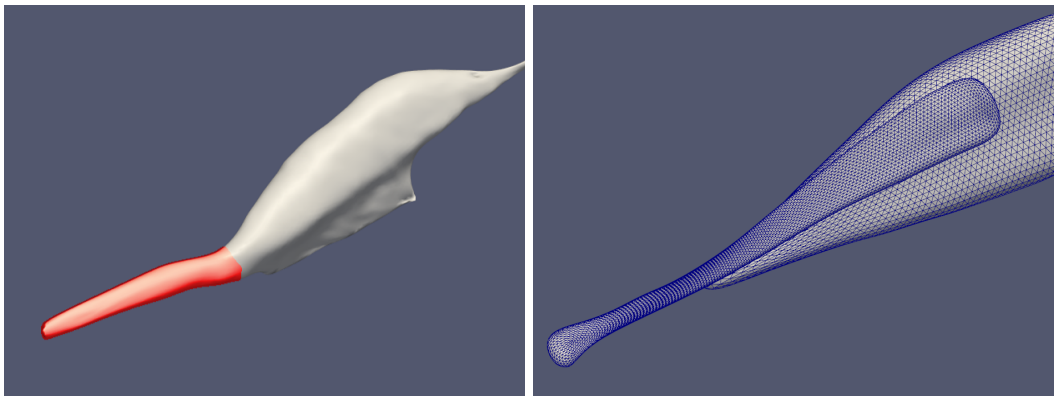


Figure 4.3: Left: The biceps femoris has its tendon included in the closed surface. Right: The soleus and achilles tendon are two separate closed surfaces

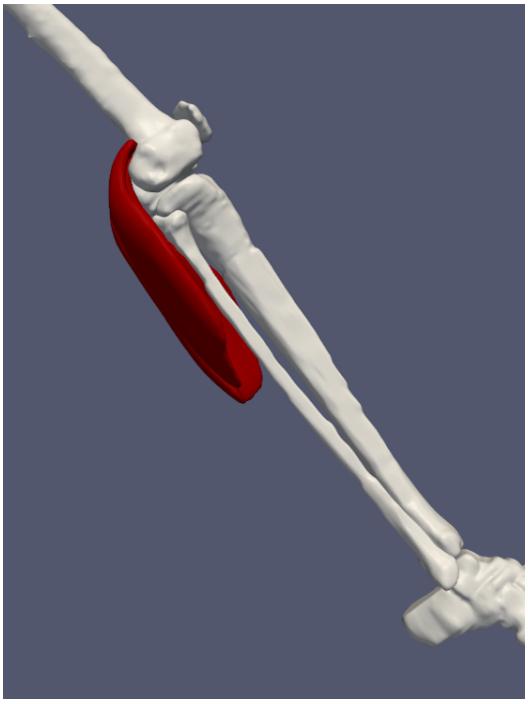


Figure 4.4: The gastrocnemius muscle is missing its tendons that would connect it to the calcaneus.

The remaining sets are each given a sequence number ordered by their set-id, for example: if the two sets remaining are 489 and 132, they receive sequence numbers 1 and 0 respectively. In the configuration of the pipeline, the muscles can be given a sequence number indicating which object to retain. We delete any vertices and reliant faces from these muscles that do not correspond to this set.

4.5 Generating missing tendons

For some muscles, the tendons are not defined and for other muscles the tendons are defined, but their attachment to the muscle is not (see also Section 4.4). Typical muscles missing tendons are the gastrocnemius muscle and the soleus muscle, which are attached to the femur, tibia or fibula at the top of the muscle, and are missing their connection to the foot (see Figure 4.4). Since these muscles are valuable for a simulation of the knee, we developed an algorithm that can automatically edit the surface mesh to incorporate a tendon structure, including an attachment and tendon specification expected further along in the pipeline.

The tendon generator gets a list of muscles that do not have tendons. For these muscles, the 'bottom' part of the muscle is detected, and is extruded toward the foot. In this case the 'bottom' direction is the negative direction along the superior-inferior axis, which in our dataset is the in $-z$ direction. This direction is based on the lying position of the scanned subject, where the subject superior-inferior axis is aligned with the $-z$ axis. It should be adapted in other scanning configurations, for example by detecting the principal axis of the tibia bone.

As a first step of the algorithm, the z-value $v^z_{foot_{max}}$ of the top vertex of the foot S_{foot} is determined.

$$v^z_{max} > v^z_i, v_i \in V$$

For each muscle S_{muscle} the faces on the bottom part are recursively traversed, starting with the faces connected to the vertex $v^z_{muscle_{min}}$ with the lowest z-value, creating a set F_{bottom} of faces belonging to the bottom part of the muscle. The set F_{bottom} is defined as follows:

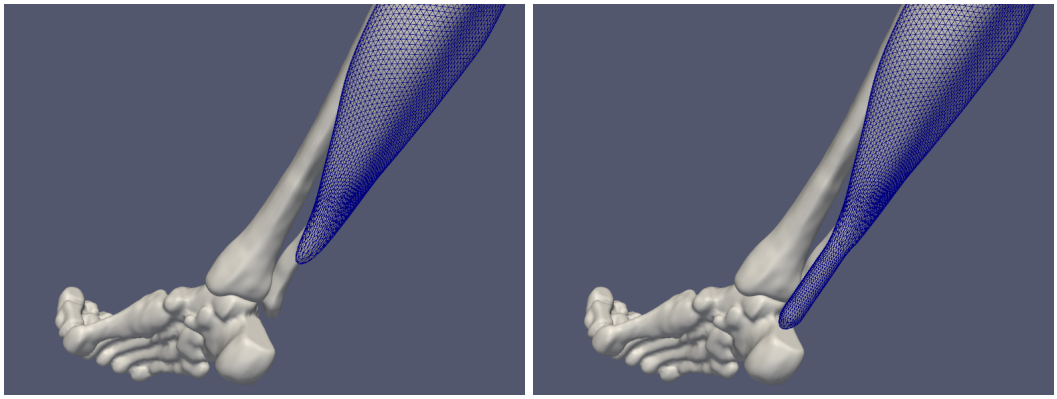


Figure 4.5: Left: The soleus muscle without tendons connecting to the foot. Right: The soleus muscle with an attachment to the foot generated automatically.

$$F_{bottom_0} = \{f_k : v^z_{min} \in f_k\} \tag{4.7}$$

$$F_{bottom_{t+1}} = F_{bottom_t} \cup \left\{f_k : f^{af}_k \cap F_{bottom_t} \neq \emptyset, angle(f_k) < 60\right\} \tag{4.8}$$

$$angle(f_k) = \frac{\cos^{-1}(fn_k \cdot -\vec{z})}{180\pi} \tag{4.9}$$

$$\tag{4.10}$$

Where f^{af}_k is the set of faces adjacent to f_k, $angle(f_k)$ the function providing the angle between a face and the xy-plane, fn_k is the face-normal of face f_k and iterations of F_{bottom} are denoted as $F_{bottom_\#}$. Summarized, we collect the set F_{bottom} of adjacent faces that have an angle with the xy-plane lower than 60 degrees.

After determining the faces that comprise the bottom of the muscle, we start the extrusion. For each face $f_k \in F_{bottom}$, we set the z-value of each of its vertices to $v'^z_i = v^z_i + (v^z_{foot_{max}} - v^z_{muscle_{min}})$. Then we determine the edges E_{border} on the border of F_{bottom}.

The vertices in the border edge set E_{border} are duplicated and are set to their original position. The gap between the original border and the duplicated border is filled with regular spaced vertices and corresponding faces F_{gap}. All the faces from F_{bottom} and the newly added faces F_{gap} are combined into a new set $F_{tendon} = F_{bottom} \cup F_{gap}$, and are together locally smoothed according to the method described in Section 4.3. In Figure 4.5 it is shown how the newly generated 'tendon' for the soleus muscle compares to the version without tendon tissue.

The newly generated piece of muscle is also saved as being tendon material by immediately generating a tendon convex hull for the new vertices, as described in Section 4.9.

4.6 Resolve self-intersections

The segmented MRI dataset contains some artifacts from the segmentation process, among which self-intersecting objects (see Figure 2.1, Right). We chose an algorithm from Jung, Shin and Choi [24], who developed an algorithm to clean up raw mesh-offsets and we use it here to remove general self-intersections. In Section 4.7 we apply the same method to remove overlaps between two surfaces. Section 2.3.3 explains the algorithm of Jung in detail, and in the following subsection we will describe the adaptations we made to the algorithm of Jung to fit it to our own purposes.

4.6.1 Remove degenerate triangles

As in the algorithm description in Section 2.3.3, we remove faces that have edges smaller that a predefined ε_e, or reconfigure faces that have angles smaller than ε_α, as can be seen in Figure 4.6. We check all faces in each surface for these two properties and solve them accordingly. For faces where an edge $e_p = (v_i, v_j)$ is smaller than ε_e, we collapse that edge by setting $v_i = (v_i + v_j)/2$ and removing v_j. Then we update the faces containing v_j to refer to v_i. After applying this operation a

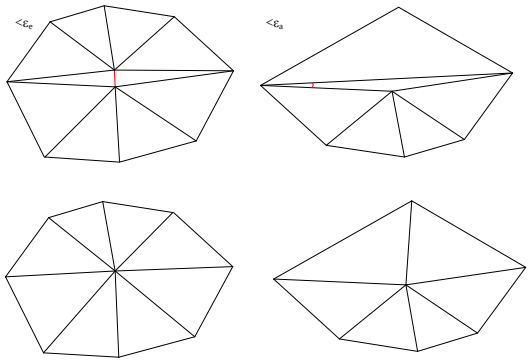


Figure 4.6: Left: Degenerate triangle having an edge smaller than ε_e. Right: Degenerate triangle having an angle smaller than ε_α.

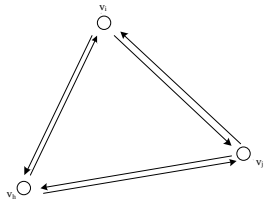


Figure 4.7: Two triangles sharing the same vertices in opposite order. One triangle defined as $\{v_i, v_j, v_h\}$ and the other $\{v_h, v_j, v_i\}$.

number of times, the geometry might contain some new topologic problems. The surface can contain 2 triangles that are each others opposites as can be seen in Figure 4.7. We detect these faces and remove them, which is a valid operation, since it is a part of the surface that defines no volume.

For a triangle f_k that has an angle smaller than ε_α, we do not remove any edges, but we change its configuration. First we determine the longest edge $e_u = (v_i, v_j)$, and find the triangle f_m sharing that edge. We then have $f_k = \{v_h, v_i, v_j\}$ and $f_m = \{v_g, v_j, v_i\}$. We now swap the edge e_u, such that $f_k = \{v_h, v_i, v_g\}$ and $f_m = \{v_g, v_j, v_h\}$, as can be seen in Figure 4.6, Right.

4.6.2 Finding a seed triangle

Because the method originally was developed for raw offset triangular meshes, selecting a seed triangle was easily determined by calculating the convex hull $VC \subset V$, and picking any triangle $f \in F$ which has at least of vertex $v_i \in VC$. In our case however, triangles taking part in the convex hull can also be invalid, as is the case in the vastus intermedius muscle seen in Figure 4.7 on page 27. For this reason, we add an additional constraint to the seed triangle selection.

First, we calculate the center of mass of the vertices of the surface, by a simple averaging of the vertex positions.

$$p_{com} = \frac{\sum_{v_i \in V} v_i}{|V|} \tag{4.11}$$

Since the surface meshes in this work are of approximately uniform density (see Section 4.1), we have a rough estimate of the center p_{com} of the object. We use this value to determine if a vertex-normal vn_i of a vertex on the convex hull is face outward of the object or inward. The vertex normal is based on the average face-normals of the faces surrounding v_i:

$$\begin{aligned} v_i^{af} &= \{k : i \in f_k\} \\ vn_i &= \frac{\sum_{f_k \in v_i^{af}} fn_k}{|v_i^{af}|} \end{aligned} \tag{4.12}$$

Here v_i^{af} are the faces that contain v_i. We then compose the following equation that gives a measure of how much a vertex-normal is facing outward:

$$\begin{aligned} \vec{u} &= (v_i - p_{com}) \\ d_i &= vn_i \cdot \vec{u} \end{aligned} \tag{4.13}$$

We calculate the dot product of the vector from the center of the object to the vertex v_i with the vertex normal vn_i. The value d_i scales from 1 to -1, where 1 means vn_i is pointing straightly outward and -1 means vn_i is pointing exactly to the center p_{com}. We calculate this value for all $v_i \in VC$, and choose the vertex v_{opt} with the highest d-value. From this vertex v_{opt}, we pick a random face $f_k \in v_{opt}^{af}$ to be the seed triangle.

4.7 Resolving overlaps

Another artifact resulting from the segmentation process is surfaces that intersect each other. We have tried several methods to resolve this problem, which is explained in detail in Chapter 5. In Section 5.3 we describe the method used in the final pipeline. The developed methods described in Chapter 5 all have common components that we explain below.

Before substracting one surface from the other, we grow the surface to be substracted, S_y by calculating a raw mesh offset. We do this by moving each vertex $v_i \in V_y$ in the direction of vn_i. The reason for doing this is to prepare for rounding errors that might occur in the next stages of the pipeline, such as the calculation of the volume mesh. These rounding errors might cause the final simulation setup file to have slight intersections between objects, and so the FE solver will not be able to find a solution for the first simulation frame. The offset S'_y is calculated as follows:

$$v'_i = v_i + \lambda vn_i \tag{4.14}$$

Where v'_i are the vertices in the new V'_y, and λ is a growth factor.

4.7.1 Order of overlap removal

The order in which the substractions are executed determines the outcome of this stage. The object that is processed first will most likely lose the most of its geometry, and will have no influence on the other objects. The object that is processed last will not be influenced at all by other objects.

We chose the bones to always be processed last, since their segmentation quality is the highest. The bones are relatively easy to distinguish on an MRI scan, therefore the segmentation result is the most reliable. Since the bones are processed lastly, they will never be influenced by muscle objects. The muscles and bones are among themselves are ordered in descending order of volume. This way the surfaces that are smaller will be processed later, meaning they will be the last to be substracted from. This is appropriate, since the impact of changes on small objects is usually higher. We use a simple bounding box method to predetermine if a substraction should be executed to reduce the number of substractions.

4.8 Generating volume meshes

Since the FE simulation is based on a volumetric representation of elements, we need to convert the triangular surface representation of our objects to a tethrahedral volume representation.

We use the algorithm provided by the CGAL library for this step [2, 36, 40]. This library allows us to parametrize the output of our mesh, such as the density of nodes inside the mesh, and the density of surface nodes. These are important, as the performance of the final simulation is largely dependent on the number of elements the model consists of. The parameters are the following:

- Facet angular bound: Controls the shape of surface facets. It is a lower bound for the angle (in degree) of surface mesh facets. The termination of the meshing process is guaranteed if the angular bound is at most 30 degrees.
- Facet radius bound: Controls the size (edge length) of surface facets. Each surface facet has a surface Delaunay ball which is a ball circumscribing the surface facet and centered on the surface patch. The radius bound is an upper bound on the radii of surface Delaunay balls.

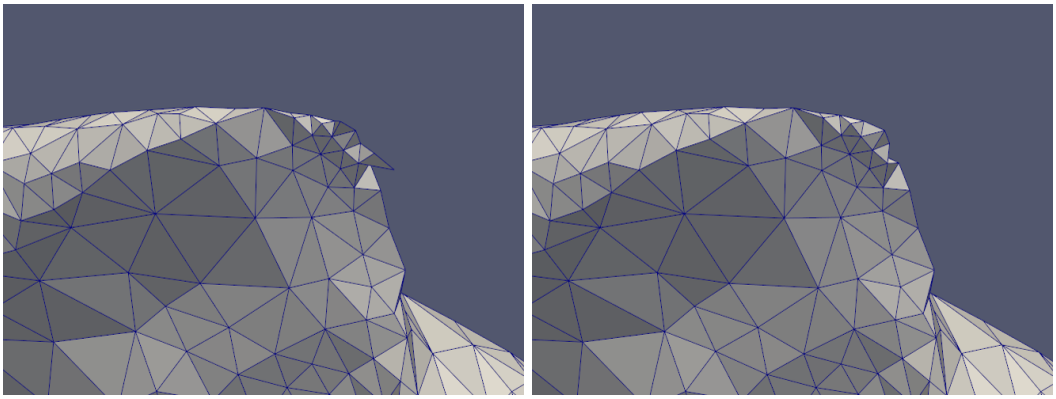


Figure 4.8: Volume mesh of the rectus femoris muscle, before (left) and after (right) the mesh cleanup step.

- Facet distance bound: Controls the approximation error of the surface. It is an upper bound for the distance between the circumcenter of a surface facet and the center of a surface Delaunay ball of this facet.
- Cell radius-edge bound: This parameter controls the shape of mesh cells. It is an upper bound for the ratio between the circumradius of a mesh tetrahedron and its shortest edge.
- Cell radius bound: This parameter controls the size (edge length) of mesh cells. It is an upper bound on the circumradii of the mesh tetrahedra.

The volume mesh produced by the algorithm is a set of vertices $V = \{v_i : 1 \leq i \leq n_V\}$, and a set of tetrahedral cells $C = \{c_k : 1 \leq k \leq n_C\}$. Each cell $c_k = (i_1^k, i_2^k, i_3^k, i_4^k)$ consists of a sequence of exactly 4 indices in the vertex list, since the algorithm only produces tetrahedral cells. A volume mesh $M = \{V, C\}$ is a vertex list combined with a cell list.

We apply some small cleanup steps after the mesh generation to remove "loose" tetrahedra, tetrahedra that have only one connection to the rest of the mesh, because these tetrahedra can lag behind because of inertia and can produce oscillations because they do not move in sync with the rest of the tissue they belong to. These oscillations can make the simulation unstable and can prevent the solver of finding solutions. If c_i^{ac} represents the cell indices of cells surrounding cell c_i, we can define the loose cells as:

$$CL = \{c_i : |c_i^{ac}| \leq 1\} \tag{4.15}$$

After selecting these elements, we remove them from the volume mesh. Figure 4.8 show the result of the mesh cleanup process, before and after.

4.9 Creating attachment and tendon convex hulls

As explained in Section 4.1, the attachments and tendons are specified as indices in vertex lists that are saved in surface mesh file. In the previous stages in the algorithm, these files are changed topologically, and finally converted to volume meshes. To preserve the attachment data we convert the index based representation to a geometric representation.

The first step is to convert the index based representation $A = \{i_1, \ldots, i_{n_A}\}$ to a position based representation $AP = \{p_1, \ldots, p_{n_A}\}$. This is done simply by looking up the vertices in a compatible vertex list V. We can use any vertex list, as long as it has the same vertex ordering as the original vertex list from the segmentation algorithm. Therefore, we use the vertex list that results from the smoothing stage as described in Section 4.3. From AP, we can calculate a convex hull, to obtain the area where the attachment is active. The convex hull is a surface $AS = \{V, F\}$ on which we can apply standard surface operations. We use the convex hull algorithm from CGAL to implement this conversion [20]. We grow the surface slightly using the raw offset operation (see Equation 4.14), to account for possible rounding errors and save this as the attachment convex hull.

Figure 4.9 shows the resulting attachment and tendon regions. To calculate the tendon regions from the specification we use the exact same process as for the attachments.

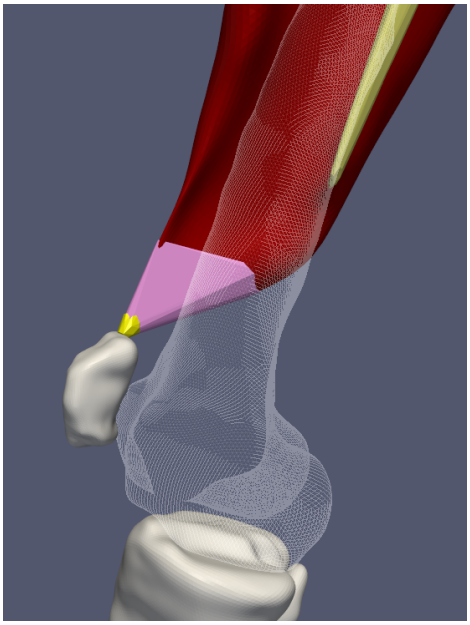


Figure 4.9: Vastus lateralis muscle in red, the tendon convex hulls are purple, the attachment convex hulls are yellow and the femur bone is transparent.

4.9.1 Converting hulls into volume mesh indices

As a final minor stage we convert the attachment and tendon regions into indices for in the volume meshes $M = \{V, C\}$. We check for each vertex $v_i \in V$ if it lies inside the attachment surface AS. We do this check using the Carve constructive solid geometry (CSG) library [1]. The new attachment specification can be described as follows:

$$AM = \{i : v_i \in V_M, inside(v_i, AS)\} \tag{4.16}$$

Here, AM is the set of indices that point to vertices in the volume mesh M. This new representation can be used directly in the final stages of the pipeline that generate the FEBio input file.

CHAPTER 5

Resolving overlaps

During the project the most complex problem was the cleaning up of surface meshes, specifically, the removal of overlaps between them. This is due to the nature of the segmentation algorithm as explained in Section 4.1. As illustrated in Figure 5.1 these intersections can be quite severe. In this chapter we explain the different methods we explored to solve the problem of resolving surface overlaps.

5.1 Push-based method

Firstly, we developed a method where two objects would push each other away. The reason for using a push-based method is that the topology of the object remains intact, and the attachment and tendon information are transformed together with the surface (see also Section 4.9). For each surface S_A and S_B we let surface S_A push the surface S_B away. The surfaces that have smaller volumes have a higher priority, meaning they will be the last to be deformed, because their changes will most likely have the most impact on the shape of the object. The bones will have the highest priority since they have the best segmentation quality, because they are the easiest to distinguish on the MRI volume (See also Section 4.7.1). Therefore, bones will never be pushed away by muscles.

First we create a copy of S_B, S'_B, and apply an iterative smoothing algorithm on it. The smoothing algorithm works as follows: for each vertex $v_i \in S'_B$, we take the set of vertices v_i^{av} that share an edge with v_i, and set v'_i as the average of $v_i^{av} \cup \{v_i\}$. We found that applying this smoothing step 50 times gives the desired result. Figure 5.2 shows the smoothed version S'_B which illustrates that the smoothed surface represents the general trend that the surface has.

Because S'_B follows the general trend of the surface, we can use this information to determine the direction u_i^B that a vertex v_i should move if it is pushed away by S_A.

This direction u_i^B is determined as follows:

$$u_i^B = (-vn'_i/2) + \begin{cases} -(v_i - v'_i)/2 & \text{if } (v_i - v'_i) \cdot n \geq 0 \\ (v_i - v'_i)/2 & \text{otherwise} \end{cases} \tag{5.1}$$

Here u_i^B is the direction we are going to move vertex v_i if it lies inside S_A, and vn_i is the vertex normal at v_i. As can be seen, u_i^B always points inwards into the surface, even if the smoothed v'_i lies

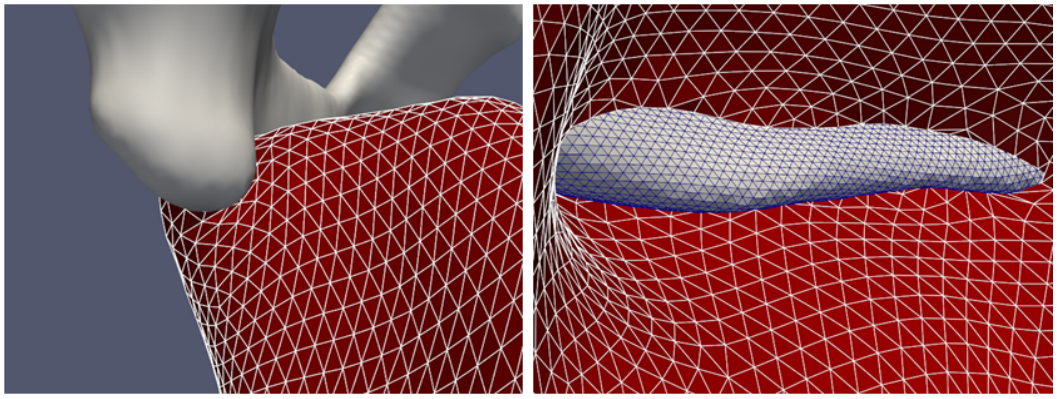


Figure 5.1: Left: Close-up of the adductor magnus where it intersects with the pelvis bone surface. Right: Same intersection as seen from the 'inside' of the adductor magnus muscle geometry.

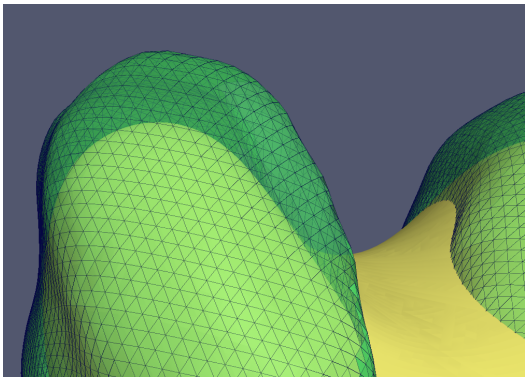


Figure 5.2: Detail of the femur, yellow being the smoothed variant of the green original.

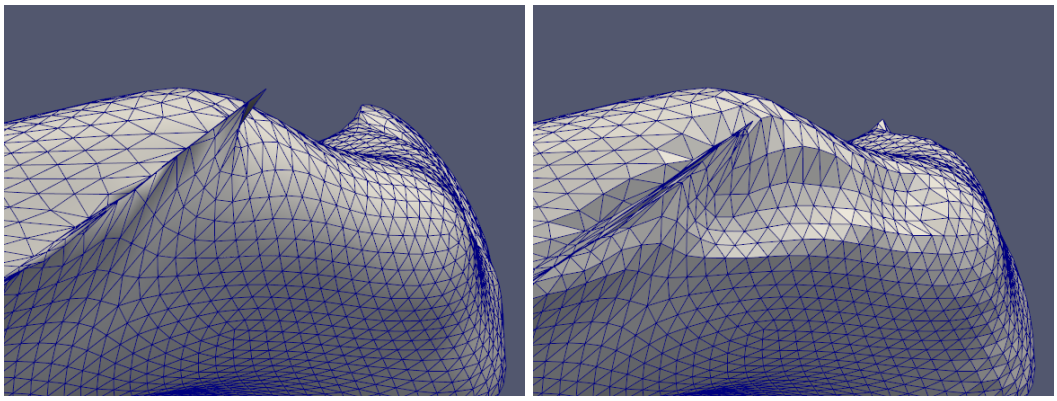


Figure 5.3: *Left:* Adductor magnus muscle before overlap has been resolved. *Right:* Adductor magnus after being pushed away by the pelvis bone.

outside. This is enforced by checking if the vertex normal vn_i is pointing in the opposite direction of $v_i - v_i'$ by using the dot product. We combine this vector with the inward normal vn_i' at v_i'.

For each vertex $v_i \in S_B$ we test if it lies inside the volume S_A. We do this check using the Carve *constructive solid geometry* (CSG) library [1]. If it lies inside we move the vertex in the direction of u_i^B, combined with the average vertex normal of the vertices of S_A close to v_i. This lookup is done using a kd-tree, for which the implementation from the CGAL library is used [2, 8]. We take the 10 closest vertices of S_A, take the vertex normals of those and average them. This gives us a second pushing direction vector for v_i which we will refer to as u_i^A, since it is based on the geometry of S_A. We take a weighted average of these two pushing direction vectors to obtain the final direction: $u_i = \lambda_A u_i^A + \lambda_B u_i^B$. This vector we feed to the pushing algorithm. We obtained the best results using values of $\lambda_A = 0.5$ and $\lambda_B = 0.5$.

The pushing algorithm works as follows. The vertex is iteratively moved in the direction of u_i. Before each step, we save the position of v_i to a variable v_i^{inside}. When v_i is moved outside of S_A, this position is saved to the variable $v_i^{outside}$. Then, using a binary search strategy, the best position for v_i is searched for between v_i^{inside} and $v_i^{outside}$ by sampling the position at $v_i \leftarrow (v_i^{inside} + v_i^{outside})/2$. If this new v_i is inside, v_i^{inside} is updated to the value of v_i and otherwise $v_i^{outside}$ is updated. The search is done when the difference between two iterations is smaller than 0.01 cm (the segmented MRI dataset coordinates are declared in cm). The result is shown in Figure 5.3, where the adductor magnus can be seen before and after it has been pushed away.

Unfortunately, this method fails if a surface S_A lies too far inside surface S_B, because pushing directions cannot be accurately determined anymore. Figure 5.4 shows an example of the resulting artifacts after applying this algorithm on a muscle that overlaps too far inside a bone.

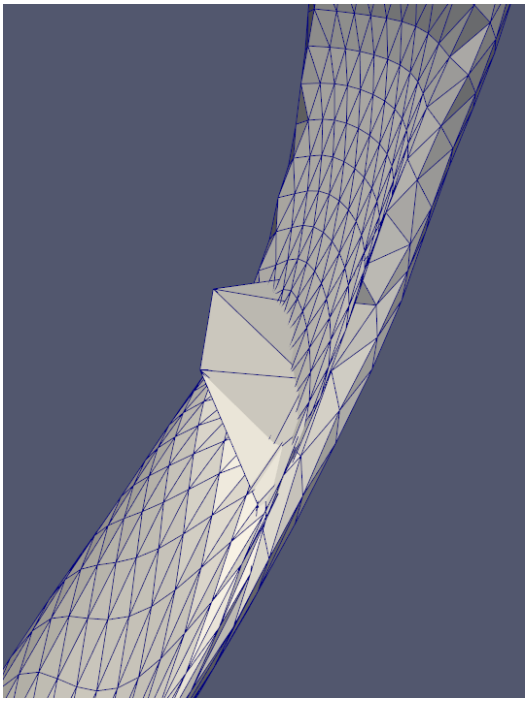


Figure 5.4: Detail of the sartorius muscle, that has been pushed away by the femur bone. At certain vertices the push direction is not adequate and the vertex is pushed wrongly.

5.2 Shrink-based method

We also developed a method that can make two surfaces push each other away by both moving vertices to a shrunk copy of the surface. For this method we need an algorithm that can give for a surface S a shrunk version S'. Shrinking an object iteratively gives a sequence of shrunk surfaces $S^1 \dots S^n$. Similarly, each vertex $v_i \in V$, has a sequence of shrunk positions $v_i^1 \dots v_i^n$.

To solve a surface overlap between surface S_A and surface S_B, we first determine the set VI_A of vertices in V_A that lie inside S_B. For each vertex $v_i \in VI_A$ we update the position to v_i^1. Then, we iteratively execute the following operations:

1. Update VI_A: remove the vertices that lie outside of S_B.
2. For each vertex $v_i \in VI_A$, we set its position to v_i^t, where t is the iteration number.

We can execute the same algorithm for surface S_B simultaneously with S_A. This way the surfaces will push each other away, and one surface will not have priority over the other.

For this algorithm to work, the shrinking algorithm should produce a sequence of surfaces that:

1. Do not change the topology of the mesh.
2. Shrink smoothly, and in small steps.
3. Only shrink, and do not grow.

We examined a couple of possible shrinking algorithms, which we explain in the following sections.

5.2.1 Shrink by smoothing

Firstly, we used a shrinking algorithm based on standard Laplacian smoothing [43]. Each vertex is moved in the average direction of its neighbors:

$$v_i^{t+1} = v_i^t + \lambda \frac{\sum_{j \in v_i^{av}} v_j^t - v_i^t}{|v_i^{av}|} \tag{5.2}$$

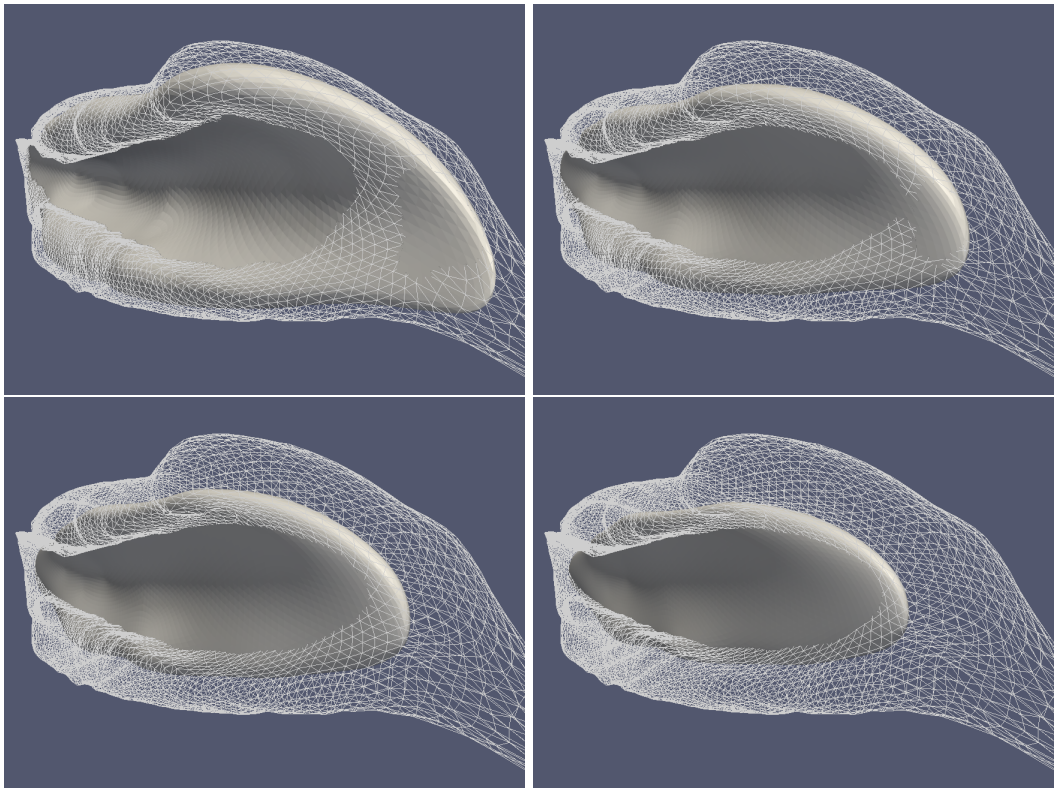


Figure 5.5: Iterations of the shrink by smoothing method of the vastus intermedius muscle. The wireframe represents the original surface. Top-left: after 25 iterations. Top-right: after 75 iterations. Bottom-left: after 125 iterations. Bottom-right: after 200 iterations.

Here λ is the scaling factor of the displacement done in each shrinking step. This method produces smooth shrinking, and can be done in arbitrarily small steps. The only problem of this method is that globally, the surface will shrink, but locally, the surface can expand. Specifically, concave parts of the surface will expand, and convex parts will shrink. Figure 5.5 illustrates this in the vastus intermedius muscle. The muscle expands in the concave part and if applied in the pushing method, the resulting surface might become larger while having been pushed away, therefore this shrinking algorithm is not sufficient. Figure 5.6 shows the results of the shrinking method applied in the shrink-based pushing algorithm.

5.2.2 Shrink to skeleton

The second shrinking algorithm we developed moves the vertices of the surface to a pre-calculated 'skeleton', which represents the global minimal structure of an object. We determine this skeleton using the method of Au et al. [7].

The algorithm takes a surface $S = V, F$ and produces a curve-skeleton $K = (U, E)$ with skeleton nodes U and edges E, where $U = \{u_1, u_2, \ldots, u_{n_U}\}$ are the node positions. The algorithm also produces a mapping $coll(v_i) = u_k$ from vertices to skeletal nodes, that gives for each vertex v_i the node u_k to which it was collapsed in the skeleton. For each node we calculate the average collapse distance $colld(u_k)$ for all vertices of that node, and we set as root node u_{root} the node with the highest $colld(u_k)$. When we have a root node determined, we can also set parent/children relations for each node. Figure 5.7 shows the skeleton of the biceps femoris and illustrates which vertices belong to the same skeletal node.

The shrinking algorithm we developed moves the vertices toward an attraction point on the skeleton. Each vertex v_i has a 'current' node $curr(v_i) = u_k$, that is initially set to $coll(v_i)$. Then the attraction point is determined as follows:

1. If the $parent(curr(v_i)) = \emptyset$, meaning the current node is the root, then the attraction point a_i is set to $curr(v_i)$.

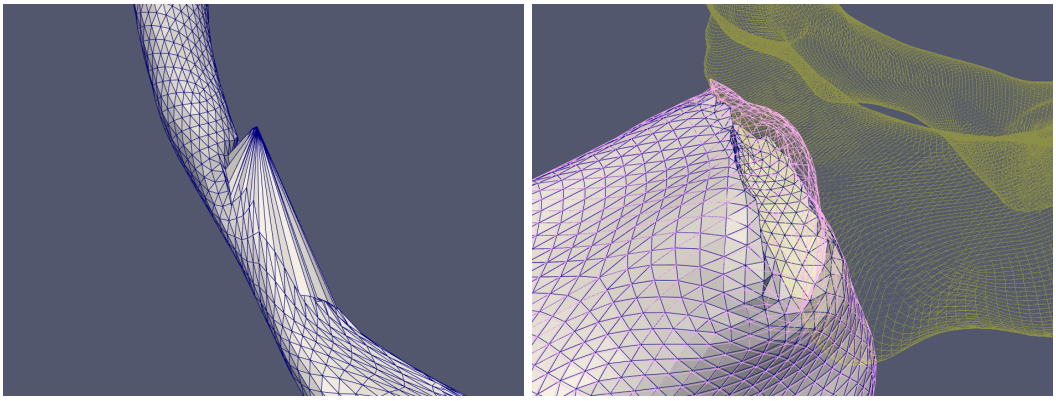


Figure 5.6: Left: Satorius muscle after being pushed away by the femur bone, using the shrink based algorithm using the smooth-shrink method. Right: Adductor magnus muscle pushed away by the pelvis bone. Yellow mesh represents the pelvis bone, pink mesh the original muscle mesh.

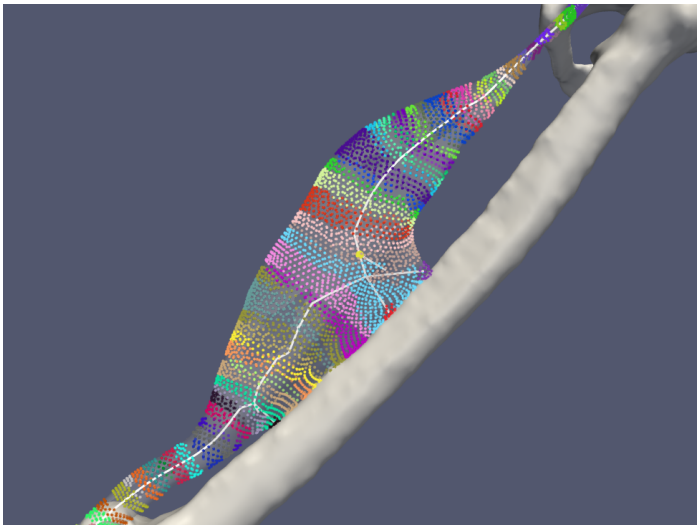


Figure 5.7: Biceps femoris muscle with the 'skeleton' in white, the root of the skeleton is a larger yellow dot. The vertices are classified based on the skeletal node they belong to, all vertices of the same color belong to the same node.

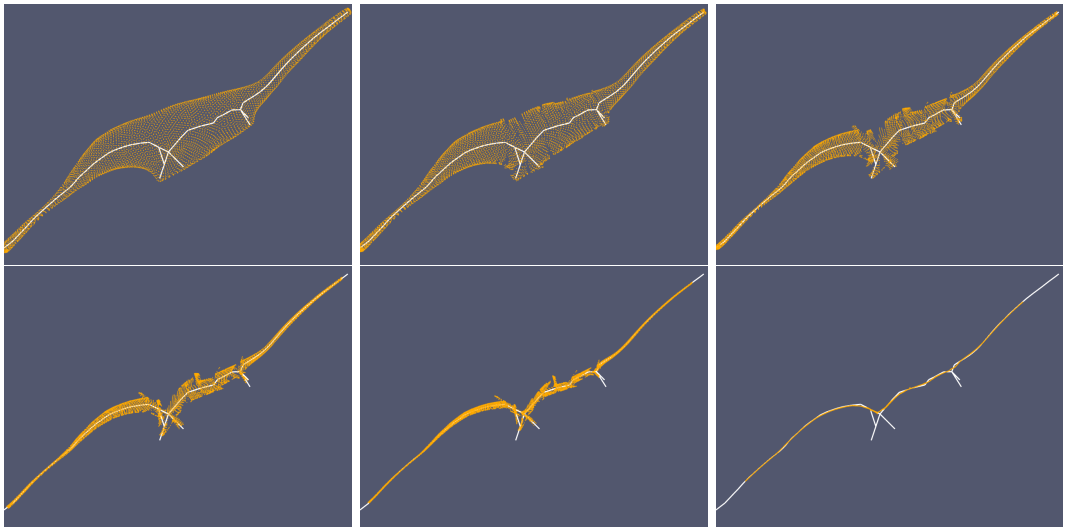


Figure 5.8: Iterations of the shrinking to skeleton method of the biceps femoris.

2. Otherwise, determine the projection parameter t of the projection of v_i on the line segment from $curr(v_i)$ to $parent(curr(v_i))$. Equation 5.3 below gives the projection parameter.
3. If $t \geq 0$, we set the current node $curr(v_i)$ of the vertex to its parent $curr(v_i) \leftarrow parent(curr(v_i))$, and go back to step 1.
4. The attraction point a_i has two components. The first component a_i^1 is set to the projection v_i' of v_i on the line segment $curr(v_i) \rightarrow parent(curr(v_i))$, and the purpose of this component is to make the vertex move toward the skeleton.
5. The second component moves the vertex toward the root: $a_i^2 = parent(curr(v_i)) - curr(v_i)$.
6. The final attraction point is then $a_i = a_i^1 + \frac{1}{|a_i^1 - v_i|} a_i^2$, where the influence of the parent direction increases as the vertex moves closer to the skeleton.

$$proj_param(l_b, l_e, p) = \left\| \frac{(l_e - l_b) \cdot (p - l_b)}{l_e - l_b} \right\|^2 \tag{5.3}$$

Figure 5.8 shows the results of the shrinking method described above. The smoothness of the shrinking could be improved by a different strategy of moving toward the skeleton. Unfortunately, the algorithm has a bigger defect: the algorithm produces for some muscles a skeleton that lies outside of the surface shape, as seen in Figure 5.9, which is not tolerable for our application, since the vertices can never go outside of the original shape.

5.3 Overlap resolving using boolean operators

Exact substractions of surfaces can be made using boolean operators. We used an existing implementation of boolean operators (Section 5.3.1) and also adapted the self-intersection removal method (Section 5.3.2) to implement substraction. Figure 5.10 shows an example of a substraction of a bone (femur) from a muscle (vastus medialis).

5.3.1 Carve

The final method we applied to resolve the overlaps was using boolean operators. We used a boolean operator implemented in the Carve constructive solid geometry library (CSG) [1], since it is the main freely available CSG library. It is being used in the popular 3D modeling software Blender [4]. It is able to perform boolean operations such as union, difference and substraction. We use the substraction operator to resolve the surface overlaps. The library can take two polygon meshes S_x and S_y and produces a mesh S_x' of which the part intersecting with S_y is removed. The boolean operators

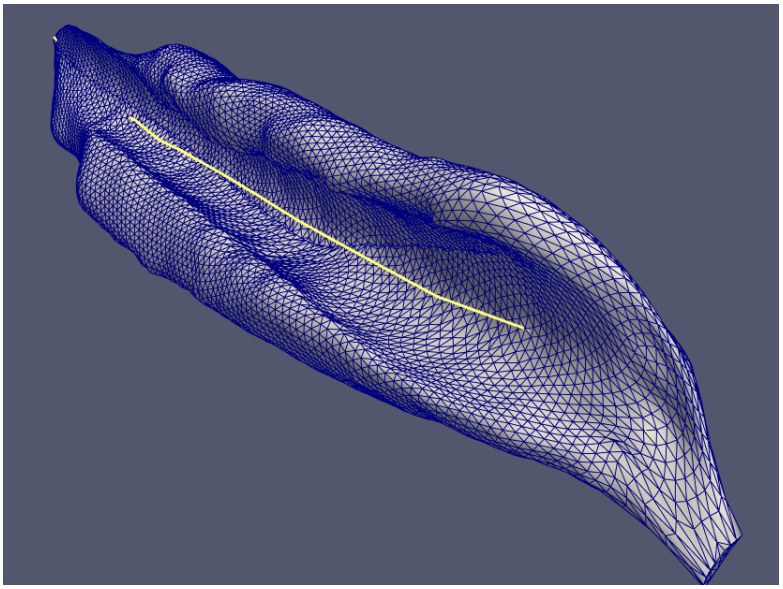


Figure 5.9: Vastus intermedius skeleton: the skeleton lies for the large part outside of the object.

are implemented using the concept of Nef polyhedra [19, 33]. Nef polyhedra in d-dimensional space are the closure of half-spaces under boolean set operation. In consequence, they can represent non-manifold situations, open and closed sets, mixed-dimensional complexes and they are closed under all boolean and topological operations, such as complement and boundary.

5.3.2 Substraction by self-intersection removal

The self-intersection removal algorithm (see Section 4.6) can also be employed as a substraction algorithm. The method obviously provides an intersection detection mechanism, but we can reuse nearly all components.

The method is globally described as follows. If we have surface S_x and surface S_y, and we want to calculate $S_x - S_y$. We merge the two surfaces into one datastructure S_z but we invert the face normal directions of S_y. We then have one surface S_z, on which we can apply the self-intersect removal algorithm with a seed triangle from S_x. The algorithm will detect the intersections between the two surfaces, and will regard the part of S_y as a self intersecting part of S_x.

The method gives the same quality results as the Carve library, but is in some cases less stable (when dealing with degenerate cases) and our implementation is less optimized than the Carve implementation. Therefore we use the Carve library in the final pipeline. In rare when cases Carve cannot find a correct solution we automatically fall back to the substraction by self-intersection removal method. The Carve library fails to create a substraction result when one of the input surfaces has one or more holes.

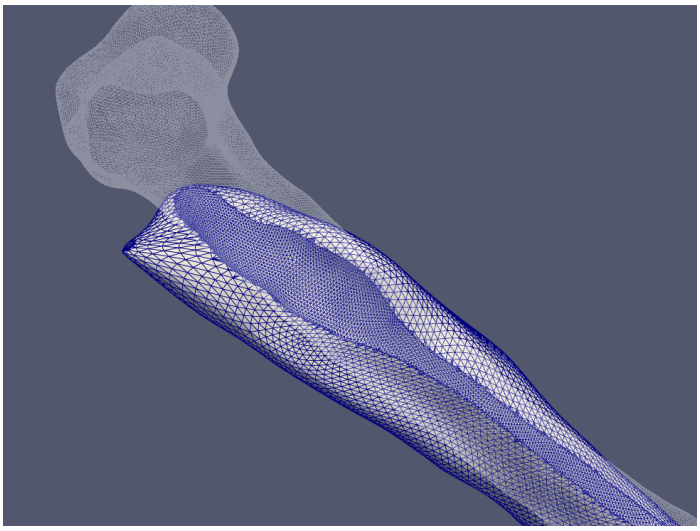


Figure 5.10: The vastus medialis after the femur has been substracted. The femur is represented as the transparent wireframe.

CHAPTER 6

Experiment

To test our method, we devised an experiment to evaluate the applicability and results of our method. In this chapter we describe the data we use from the musculoskeletal simulation and how to convert to be usable in the FE simulation (Section 6.1). Then we describe the implementation details of building the FE setups, the software used in the process, and how we can inspect and evaluate the experiment (Section 6.2). Finally, we show the individual components of the experiment we created and its results (Section 6.3).

6.1 Musculoskeletal input

We use the motion from a musculoskeletal simulation to drive our final simulation. This way we extend the musculoskeletal simulation with a FE simulation with which we can study muscle deformations in detail, which is not possible in the musculoskeletal simulation. We use the motion and the joint models from the musculoskeletal simulation in the FE simulation.

6.1.1 Musculoskeletal models

OpenSim is a freely available, open-source software system that lets users develop models of musculoskeletal structures and create dynamic simulations of a wide variety of movements [14]. The software can use inverse kinematics to generate a sequence of poses for a skeleton model from motion capture data.

The movement of the joint can be defined in a large number of ways. The simplest model of the knee joint is a single rotation about a stationary axis in the sagittal plane. But more complex models exist, such as the model of Walker et al. [47], and Yamaguchi et al. [48]. These models have already been implemented in OpenSim musculoskeletal models by Arnold et al. and Delp et al. [6,13] respectively. For this experiment we used the model from Walker et al. In this knee-joint model, the configuration is parametrized by the flexion-extension angle. Dependent on this angle are two translational degrees of freedom, anterior-posterior and inferior-superior, specified by two *natural cubic splines*. The knee-model also specifies a slight rotation around the other two axis, each defined by their own natural cubic spline.

6.1.2 Coordinate system conversion

At the first stage of the pipeline, we record a motion of the same subject of which we took the MRI scan and scale the OpenSim model accordingly. If we want to use the motion from the OpenSim model, we need to convert the joint motion from the OpenSim definition to a transformation (rotation and translation) of the bones in the finite element simulation.

The OpenSim model and the MRI data both contain a set of markers. They are located on specific anatomical landmarks, as defined by Cappozzo et al. [12]. The OpenSim model uses markers on well defined locations on the skin to track the subject, so these anatomical landmarks can be used to orient the model. To pinpoint the exact same landmarks on the MRI dataset, we can either directly scan the subject with markers filled with contrast agent or we can virtually locate the anatomical features in the 3D segmented images. The markers in the dataset used in this project consist of markers obtained using a mix of both methods.

Since the experiment we did for this research is constrained to the knee-joint, we will describe the method for the knee joints here. As we scale the generic musculoskeletal model to our subject, the joint description is also scaled. The IK algorithm of OpenSim will match the joint configurations to our motion capture data. From this scaled model we calculate the joint configuration for each time-frame of the motion, and transform this configuration to the coordinate system of the MRI

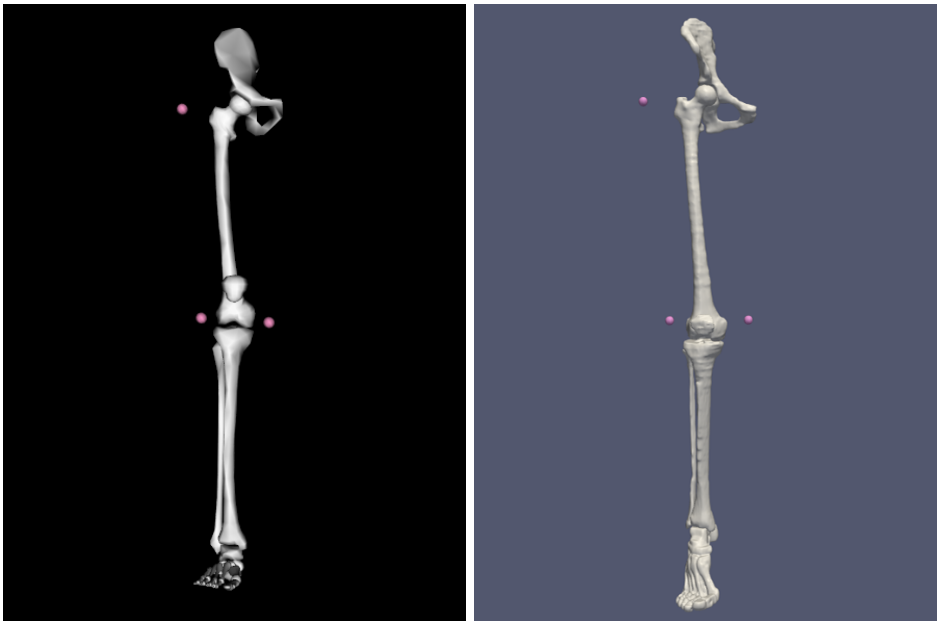


Figure 6.1: Markers used for the coordinate system conversion. Left in the OpenSim model, right in our MRI dataset.

data. This is done by taking three common marker positions from the MRI dataset and from the motion capture dataset and define coordinate systems between these markers. Between these two coordinate systems we can easily specify a transformation. For the knee-joint conversion, the markers we used are the GT (great trochanter), LFE (lateral femur epicondyle) and the MFE (medial femur epicondyle), shown in Figure 6.1.

Since OpenSim represents its motion locally, we converted the motion to world space, including the lower leg offset.

6.2 Implementation

This section describes the implementation of the pipeline used for the experiment that is described in Section 6.3. We discuess the knee model used in OpenSim (Section 6.2.1), the specifics of using the FEBio finite element solver (Section 6.2.2) and finally the parameters used for each algorithm in the pipeline (Section 6.2.3).

6.2.1 OpenSim

As we said earlier, the OpenSim model used in this experiment is developed by Arnold et al. [6]. To obtain the motion from this model, we read the .osim file scaled to our subject, and we need a corresponding .mot motion file. The .osim file describes how the joints behaviour is related to the joint-coordinates such as knee or hip angles. Since the OpenSim API does not provide functionality to extract joint configurations, we implemented the skeletal model of OpenSim that supports all the functions that relate coordinates to joint rotations and translations. We implemented this part of the pipeline in Java because of the native XML support, and the reflection capabilities that allowed us to easily describe a XML-object mapping.

6.2.2 FEBio

The pipeline described in Chapter 4 produces a number of data units that have to be combined to be ready to be used with a finite element solver. For this pipeline, we have used the FEBio software, developed at the University of Utah [3]. As can be seen in Figure 6.2, the FEBio file consists of several components. This part of the pipeline, from the segmented MRI data to the FEBio file was written in C++, since many geometry processing libraries such as CGAL and Carve were designed to be used with C++.

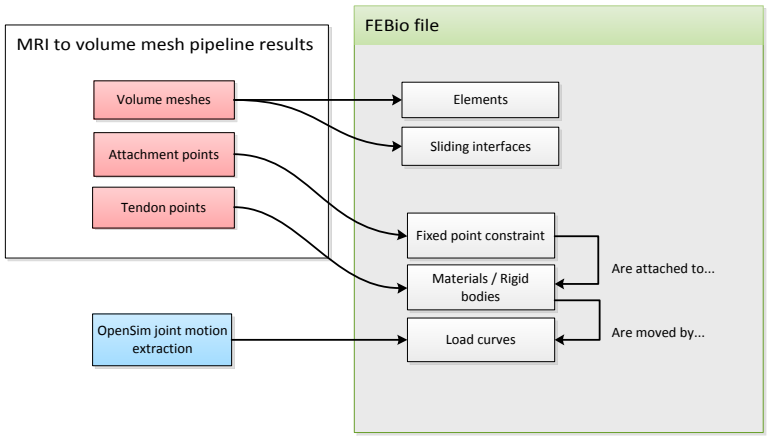


Figure 6.2: Overview of the compilation of a FEBio setup file.

The FEBio file consists of an element list, material definitions, and set of constrains. The elements in the FEBio file are the elements from the volume meshes from the MRI-to-volume mesh pipeline. The bones in our simulation are represented by rigid bodies. This can be done because bone deformation is only feasible with forces much greater than we are studying in our daily activity trials. Because the bones are rigid bodies, FEBio allows us to change their position and orientation during the simulation. The changes are defined using load-curves, one load-curve for each degree of freedom, in total six degrees of freedom (3 rotations, 3 translations). These six degrees of freedom are not all used in our simulation, but they can all be adapted if necessary. In the knee-model of Arnold et al. [6], 5 degrees of freedom are used, 3 rotations and 2 translations. We divide the bones into sets, one for each joint that is moved by a joint, and one for those that stay fixed. The bones that move together are given the same rigid body material, and will all move according to the appropriate load-curves.

Earlier in the pipeline (Section 4.9), the indices in the volume mesh of the attachments were calculated. We attach these vertices to the appropriate bone by creating fixed point constraints. These fixed points will move in the same way as the rigid body they are attached to. The attachment location on the bone is therefore not relevant, because all points on the bone move the same way.

All the muscles are given the same material specification (as described in Section 6.3.1). The tendon indices are used to identify elements that should get a tendon material assigned. If an element has all vertices inside the tendon-index set, it will get the tendon material assigned.

The creators of FEBio also created the evaluation tool PostView to view the results of the simulation. It can visualize a range of physical properties of the elements, such as displacement, pressure and strain. We use this tool in our analyses for our experiment, therefore all images of simulation results are made using this software.

6.2.3 Pipeline parameters

Not all algorithms in the pipeline can be used without configuration. In this section we describe all parameters used in the pipeline. The smoothing algorithm (Section 4.3) has three parameters λ, μ and N. The surfaces in our dataset are smoothed with these parameters: $\lambda = 0.33$, $\mu = -0.331$ and $N = 1000$ (iterations). The properties of the low-pass filter are the pass-band frequency k_{PB}, the pass-band ripple δ_{PB}, the stop-band frequency k_{SB}, and the stop band ripple δ_{SB}. The parameters

Parameter name	Muscle	Bone
facet angular bound	30.0	30.0
facet radius bound	5.0	15.0
facet distance bound	0.5	0.75
cell radius-edge bound	1.25	4.0
cell radius bound	3	100

Table 6.1: CGAL Mesh generation parameters

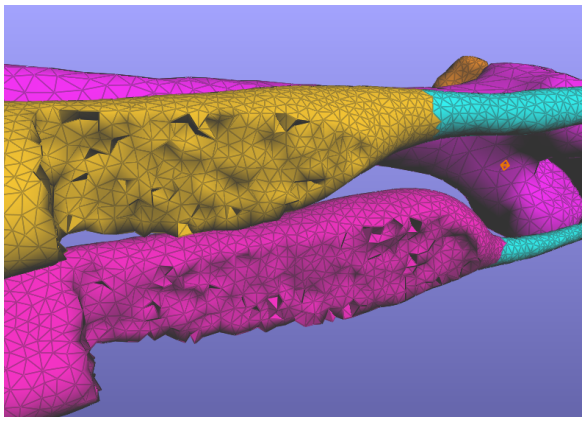


Figure 6.3: The biceps femoris and semimembranosus muscles after being cut in the middle. The elements on the inside of the muscles are visible.

are related to the low pass filter properties as follows:

$$\lambda < \frac{1}{k_{SB}} \tag{6.1}$$

$$\frac{1}{\lambda} + \frac{1}{\mu} = k_{PB} \tag{6.2}$$

$$\left[(\lambda - \mu)^2/(-4\lambda\mu)\right]^N < 1 + \delta_{PB} \tag{6.3}$$

$$\left[(1 - \lambda k_{SB})(1 - \mu k_{SB})\right]^N < \delta_{SB} \tag{6.4}$$

In the tendon generation step (Section 4.5), we add regularly spaced vertices to fill the gap between the end of the tendon and the start of the muscle. The space between these rows of vertices is set to 2. The value can be absolute, since the dimensions of the MRI data are always in millimeters (mm).

The degenerate triangle method (Section 4.6.1), needs two ϵ values, ϵ_e and ϵ_α, that indicate the minimal edge length and minimal angle, respectively. In the experiment, we use $\epsilon_e = 0.5$ cm and $\epsilon_\alpha = 0.01$ cm.

The overlap resolve algorithm (Section 4.7) creates a raw-offset of the surface that will be substracted. The parameter λ indicating the distance of the offset is set to $\lambda = 1$.

The volume mesh generation step has several parameters determining the final density and accuracy of the output mesh. Table 6.1 lists the parameters used on our dataset. For both muscle and bone objects, we choose the same values for angular bound and radius-edge bound. This is because these values do not influence the amount of elements created in the output mesh, but only influence the runtime of the mesh generation. Facet angular bound is set to 30 because the CGAL algorithm will guarantee a solution for this bound. The radius-edge bound is set to 1.25, because we want to avoid unbalanced tetrahedra. For both the radius bounds, we choose for the muscle a lower value since it will deform during the simulation and therefore needs a higher resolution. The bones are not going to deform so the size of the tetrahedra inside of the bones is not important, hence the high value. Figure 6.3 shows the result of these parameters of the generated volume meshes of bone and muscle.

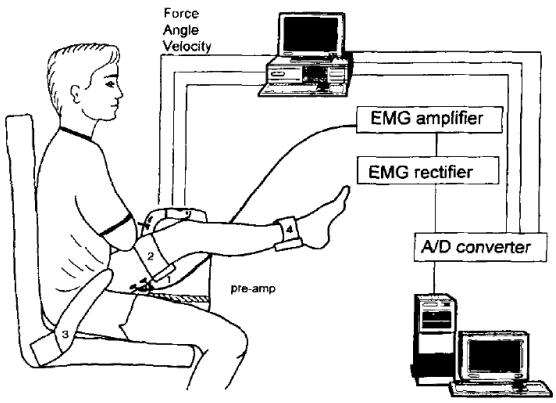


Figure 6.4: The experimental set-up. (Image and description from [28]) Resistance (Nm) to stretch was measured with the use of an isokinetic dynamometer during passive knee extension. Subjects were seated with the trunk perpendicular to the seat and the thigh rested on a specially constructed thigh pad (1) elevating it 0.3WI.45"(range) from horizontal, which disallowed complete knee extension and therefore placed tension primarily on the muscle-tendon unit rather than posterior capsular constraints. Passive force (N) was detected by the load cell (4). The dynamometer and knee joint axis were aligned and the torque about the knee joint was calculated by multiplying the measured force by the lever arm distance. The distal thigh and pelvis were firmly secured with straps (2, 3). Gross electrical activity of the human hamstring muscle group was measured with surface electrodes placed midway between the gluteal fold and the knee joint (1). Custom-made amplifiers with a frequency response of 20 Hz to 10 KHz and 1:1 pre-amplifiers were used for EMG signal sampling. The force from the load cell, the velocity and angle of the lever arm (via the KinCom PC) and the EMG signal were continuously and synchronously sampled via an A/D converter and stored on a PC for subsequent analysis.

6.3 Hamstring stretch experiment

The pipeline can generate simulations that are driven by the motion of the bones. This means that the muscles are not contracting actively. Therefore the muscles are moved only passively and we should compare the results of our experiment with research on passive muscle motion. We recreate the experiment described in [29] which is about the passive stretching of a knee joint. Figure 6.4 shows the setup of the experiment. Each subject was positioned on a testing apparatus lying on his left side with the pelvis and left thigh fixed at 90° against a padded anterior thigh block. The experiment measures the passive force of the entire knee joint while being stretched. Since in our finite element simulation result we cannot directly determine resistance we use the torque to stress conversion described in [28] with which we can compare the output of our simulation. In the experiment, the leg is stretched with constant angular velocity, as can be seen in Figure 6.5.

6.3.1 Materials

For all the bones in the experiment we use a rigid body material. Rigid bodies can be translated and rotated during the simulation, therefore they can be used to execute the motion.

The muscle and tendon material we used is Neo-Hookean material, which is a standard material in FEBio. We obtained the parameters for the Neo-Hookean materials from the work of Maurice et al. [30] (from [11]). For the tendon we use a Young's modulus of 1200 MPa and a Poisson's ratio of 0.4. The muscles have a Young's modulus of 120 MPa and a Poisson's ratio of 0.4.

6.3.2 Muscles

The muscles that we use in the simulation of the experiment are the biceps femoris and the semimembranosus which are two muscles that are part of the hamstring muscle group.

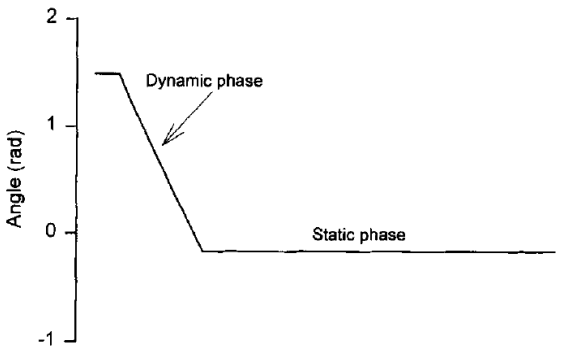


Figure 6.5: Graphic representation of a static stretch. In the dynamic phase the leg is passively extended (0.0875 radis) to a pre-determined final position. The static phase is when the leg remains stationary in the final position for 90 s. [28]

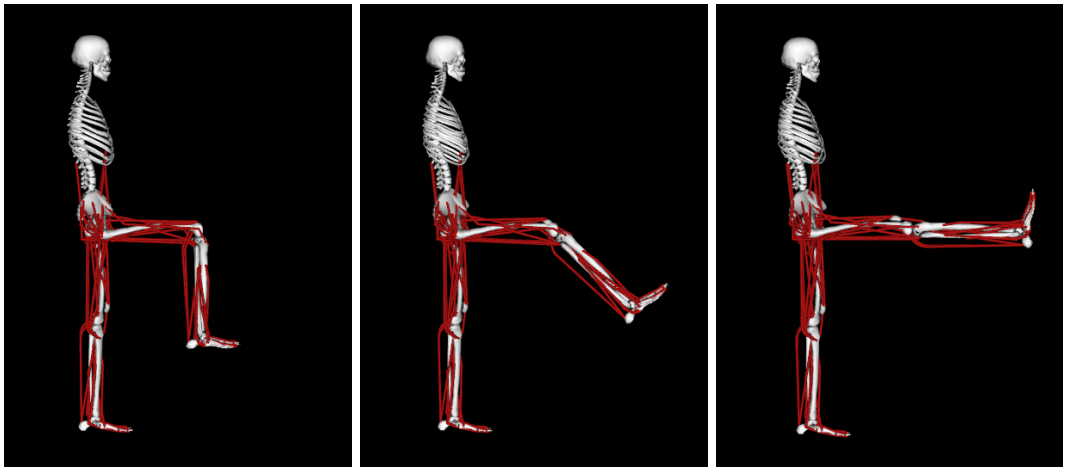


Figure 6.6: Motion of the experiment as visualized by OpenSim.

6.3.3 Motion

The motion in this experiment was not recorded since the exact motion of the leg is exactly defined already. We generated a .mot file to be entered into the pipeline. Since the leg in our MRI dataset is initially stretched, we first move the leg into the initial position. The hip is flexed 90° and at the same time the knee is bent 90°. Figure 6.6 shows three frames of the motion file loaded in OpenSim. In the conversion between the OpenSim motion and the simulation motion we generate a transition between the initial state of the bones and muscles in the segmented MRI data to the first frame of the OpenSim motion. This is done because we do not know how the muscles should be deformed in the initial state of the motion other than by simulating the deformation, therefore we first move to the initial state. Figure 6.7 shows the transition motion of the bones from the initial state in the segmented MRI dataset to the first frame of the OpenSim motion.

The final motion in the simulation starts after 6.1 seconds. The initial stage consists of 0.1 seconds motionlessness, then 5 seconds in which the initial state is reached and finally 1 second motionlessness again. After the initial stage the motion described by the hamstring stretch experiment is executed, which takes 20 seconds after which 3 seconds of motionlessness occurs. The times of motionlessness can be short since we are not using a dynamic solver, but a semi-static solver. Therefore dynamic effects such as inertia are not considered.

6.3.4 Results

The results of the experiment consist of the deformation of the muscles and the stress values reported by the finite element solver. The deformation can be seen in Figure 6.10, which starts at 5.2 seconds, after the initial stage is completed. Figure 6.8 shows the development of the stress values over

Figure 6.7: From the initial state of the bones in the segmented MRI data to the first frame of the OpenSim motion.

time for 6 elements from the simulation. We measured the stress values on three locations for each muscle: in the center and two on each superior-inferior side, just before the beginning of the tendon. Figure 6.9 shows the stress of the entire model.

The solving the simulation with FEBio took 1 hour and 8 minutes on a computer with an Intel Core2 Duo P7450 running at 2.13 GHz with 4 GB of RAM.

6.3.5 Analysis and comparison

In the visualization of the deformation of the muscles, we note that the inferior tendon of the biceps femoris muscle is bent quite sharply. We see also that the inferior tendon of the semimembranosus is at the end of the simulation pulled over the femur bone.

The stress development in the semimembranosus follows the expected pattern; the more the leg is stretched, the more stress the muscle undergoes (See Figure 6.8). For the biceps femoris, we see a completely different behavior of the stress variable. Each location on the muscle seems to show a different development. Elements from the area of E19105 do not seem to be influenced a lot by the stretching of the leg. They are however influenced by the flexion of the hip joint, since the initial stress value has grown from 0 to 3.3 in the initial stage. The stress of element E35631 starts relatively constant until the moment when the inferior tendon of the biceps femoris is no longer bent and starts stretching. This means that the area around element E35631 was pushed together by the bending of the tendon and is slowly de-pressurized by the unbending of the tendon. Element E42643 follows a similar development, but the moment the inferior tendon starts stretching it is itself pulled apart by the stretch and therefore the stress increases. We cannot confirm whether this behavior is realistic since we do not have data indicating otherwise.

In Figure 6.9 we see that the development of the average stress is comparable in shape to the results of the live subject experiment from Magnusson [29]. Our graph is more detailed than the one from Magnusson, which uses only 6 measurement points, explaining why the downward bump around second 19 is not found in their results.

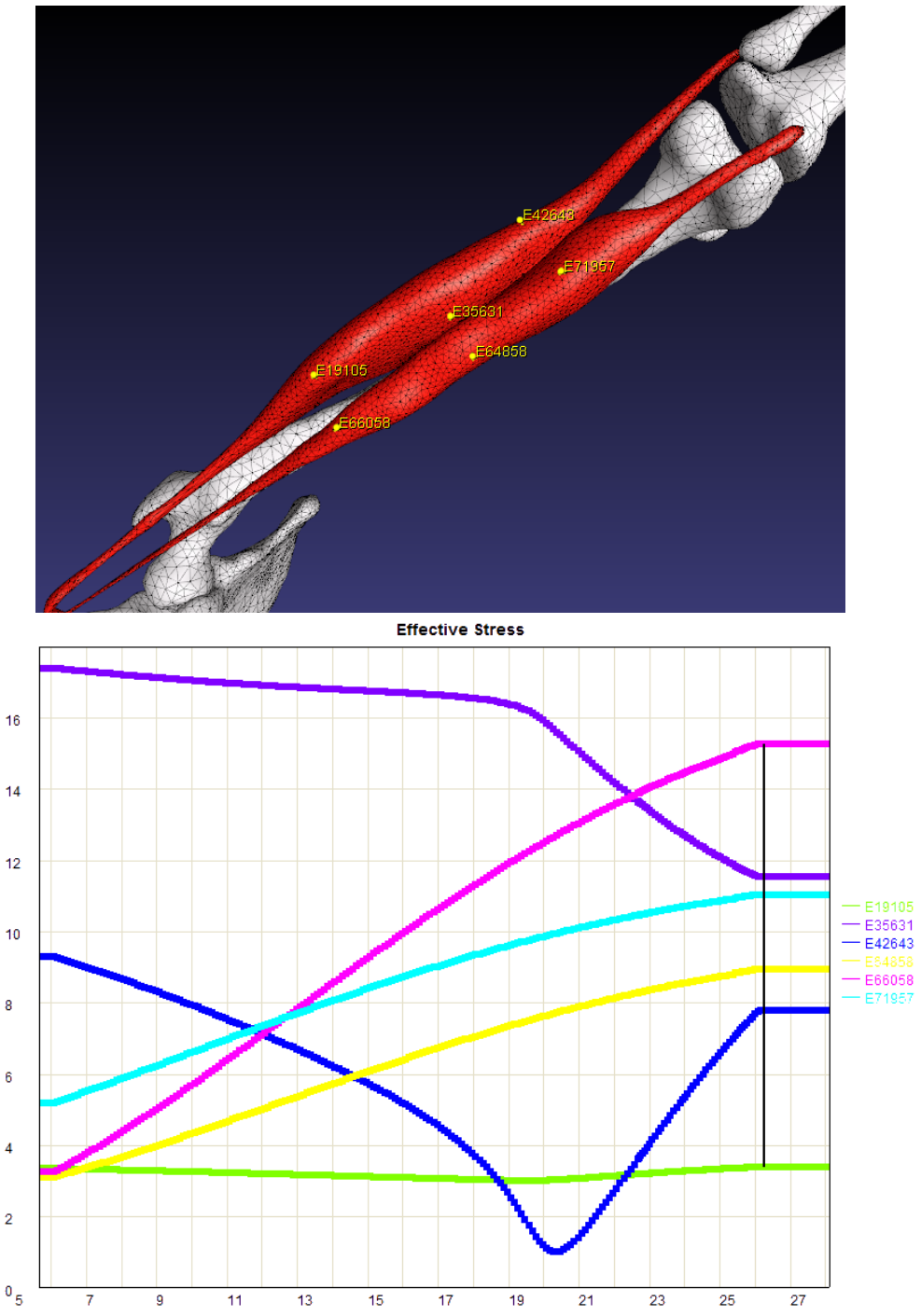


Figure 6.8: Bottom: Stress over time of elements in the simulation. Top: Location of the elements in the simulation. The elements marked E19105, E35631 and E42643 belong to the biceps femoris muscle and elements E66058, E64858 and E71957 belong to the semimembranosus muscle.

Figure 6.9: Average stress of the entire model.

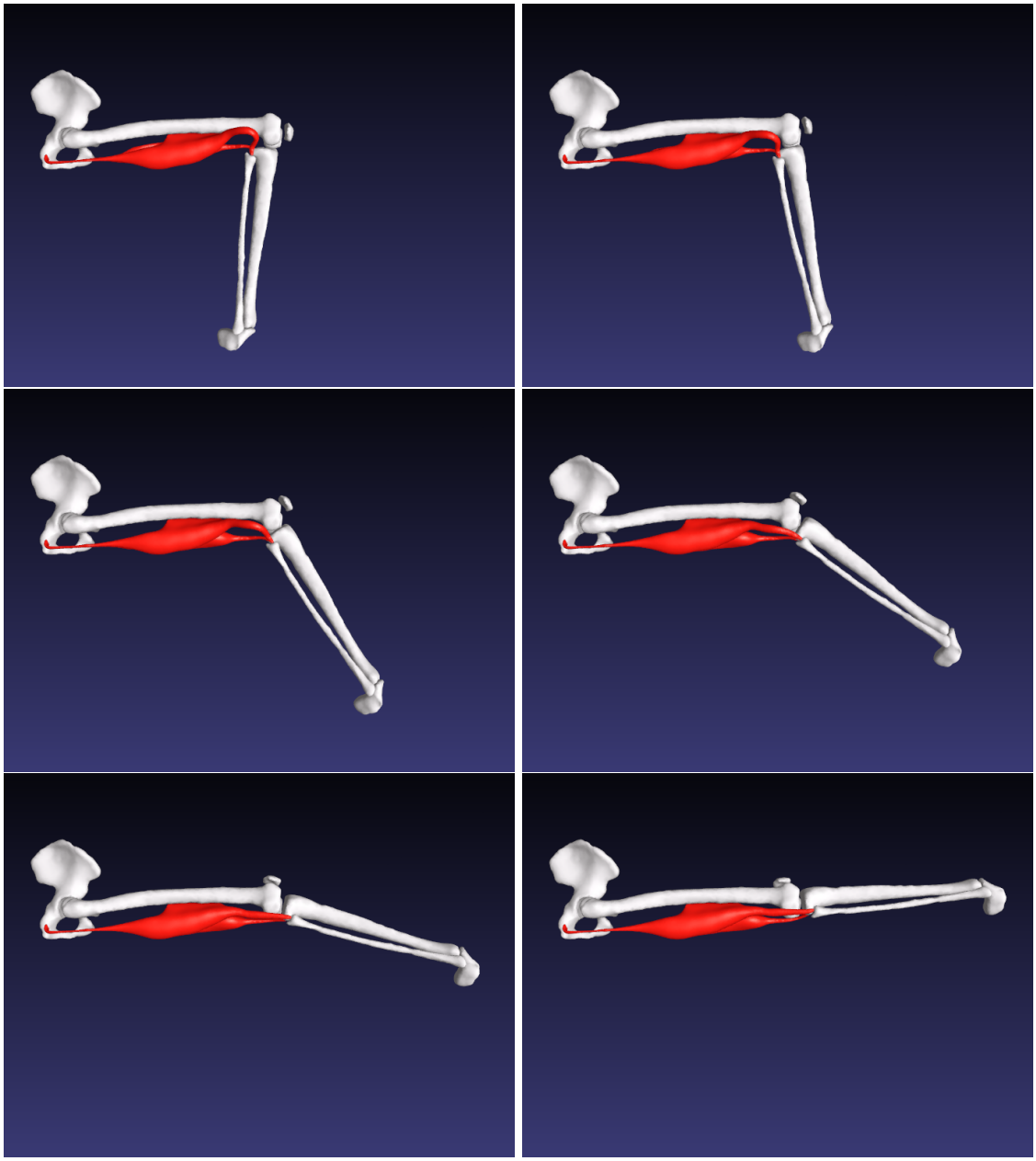


Figure 6.10: Visualization of the simulation results of the hamstring stretch experiment. From left top right, top to bottom, the timestamps of each frame are: 5.2, 9.4, 13.6, 17.8, 22, 26

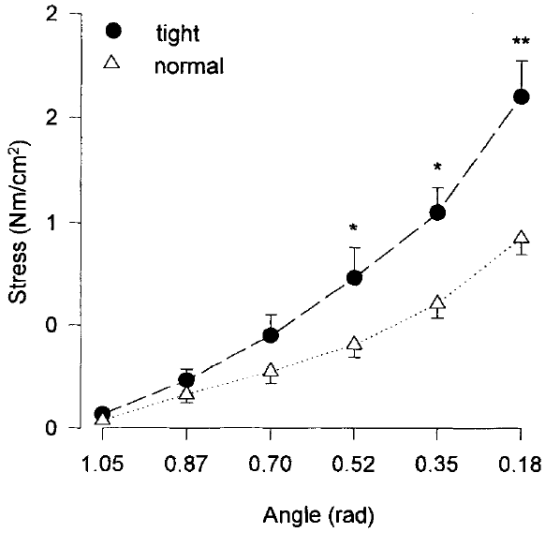


Figure 6.11: Stress development in the experiment of Magnusson [29]: Mean±SEM. Stress-strain (angle) curve in the common range for the normal (n=7) and tight (n=7) subjects. Significantly different, $*P < 0.05$, $**P < 0.01$. (Image from [28])

CHAPTER 7

Conclusion and future work

We have presented an automated pipeline that resolves data inaccuracies such as muscle overlaps and self-intersecting muscles. We also created an automated conversion between a musculoskeletal motion and a finite element simulation thereby creating a unified platform for neuromuscular and finite element simulation. The pipeline is able to extend the capabilities of the OpenSim musculoskeletal platform by taking the subject-specific motion generated using the musculoskeletal simulation environment to drive a MRI-based finite element simulation that is generated from a subject-specific segmented MRI-dataset. The pipeline allows the user not only to select the specific muscles to be used in the simulation, but also the density of elements of the simulation and the material properties of the muscles and tendons.

The pipeline's main features are the cleanup of input data, making the pipeline robust. Segmentation artifacts from low MRI image quality are smoothed out, self-intersections are removed and inter-object overlaps are resolved. The pipeline also has the ability to semantically clean up the data, by removing objects from specific datafiles and generating new tendon geometry in case of missing tendons. The volume mesh generation is also performed automatically but should be configured to the user's need, depending on the user's preferred detail and performance of the simulation. The attachment sites and tendon areas are automatically added to the output of the pipeline: a FEBio simulation file that has the volume data, material- and attachment specifications and the motion the bones should make over time. The motion is extracted from the musculoskeletal simulation platform OpenSim and is converted to the coordinate system of the finite element simulation.

Our experiment showed that the pipeline is applicable to real-world problems and gives results comparable to actual experiments. Unfortunately, the pipeline can only generate simulations that are passively driven, meaning that experiments where muscles are activated cannot be performed. Future research could include the activation of muscles during the simulation. The musculoskeletal simulation platform can produce muscle activations by analyzing the input motion, which could be used to determine the activation times and intensities for each muscle in the simulation. The bones can be either actively moved (as in the current pipeline), or passively moved by the muscles. In the latter case one of the goals of the simulation would be to for the resulting motion to match the original recorded motion as closely as possible.

In this work, we did not specifically focus on the material properties of the simulation. Our materials are isotropic, while in reality muscle and tendon materials do not deform uniformly. Materials which support fiber directions have been developed, but need a significantly more elaborate configuration. Together with muscle activation these materials would increase the accuracy of the simulation greatly. The disadvantage is of course the increase in parameters, and most likely the manual configuration of fiber directions for each muscle.

Future work should also focus on applying the pipeline on a dataset of higher quality. Higher resolution of MRI data with less noise can be obtained by using a scanner with a stronger magnetic field. The resulting segmentation would have less artifacts and the resulting simulation would be more accurate. Also, a future research could create dynamic MR images of a leg in motion, which could be compared to the results of the FE simulation performing the same motion. This way, the deformation of the muscles in the FE simulation can be compared to a real world example, which would validate the deformation of the muscles, thereby confirming or invalidating the parameters of the simulation. This would serve fine tuning process of the parameters of the materials.

Our pipeline provides a connection from musculoskeletal simulation to finite element simulation. Future research could also create a reverse connection, by visualizing the FE results in, for example, OpenSim such as done by Pronost et al. [38].

Bibliography

[1] Carve, constructive solid geometry library, July 2010. http://carve-csg.com.

[2] CGAL, Computational Geometry Algorithms Library, July 2010. http://www.cgal.org.

[3] FEBIO, Finite Elements for Biomechanics, July 2010. http://mrl.sci.utah.edu/software.php.

[4] Blender, The Blender Foundation, July 2011. http://www.blender.org.

[5] U.S. National Library of Medicine. The visible human project, June 2011. http://www.nlm.nih.gov/research/visible/.

[6] Edith M. Arnold, Samuel R. Ward, Richard L. Lieber, and Scott L. Delp. A model of the lower limb for analysis of human movement. *Ann Biomed Eng*, December 2009.

[7] O.K.C. Au, C.L. Tai, H.K. Chu, D. Cohen-Or, and T.Y. Lee. Skeleton extraction by mesh contraction. In *ACM SIGGRAPH 2008 papers*, pages 1–10. ACM, 2008.

[8] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, 1975.

[9] S.S. Blemker and S.L. Delp. Three-dimensional representation of complex muscle architectures and geometries. *Annals of Biomedical Engineering*, 33(5):661–673, 2005.

[10] S.S. Blemker and S.L. Delp. Rectus femoris and vastus intermedius fiber excursions predicted by three-dimensional muscle models. *Journal of biomechanics*, 39(8):1383–1391, 2006.

[11] J.A. Buckwalter, T.A. Einhorn, S.R. Simon, and American Academy of Orthopaedic Surgeons. *Orthopaedic basic science: biology and biomechanics of the musculoskeletal system*. American Academy of Orthopaedic Surgeons, 2000.

[12] A Cappozzo, F Catani, U Della Croce, and A Leardini. Position and orientation in space of bones during movement: anatomical frame definition and determination. *Clinical Biomechanics*, 10(4):171 – 178, 1995.

[13] S. L. Delp, J. P. Loan, M. G. Hoy, F. E. Zajac, E. L. Topp, and J. M. Rosen. An interactive graphics-based model of the lower extremity to study orthopaedic surgical procedures. *IEEE Trans Biomed Eng*, 37(8):757–767, August 1990.

[14] S.L. Delp, F.C. Anderson, A.S. Arnold, P. Loan, A. Habib, C.T. John, E. Guendelman, and D.G. Thelen. Opensim: Open-source software to create and analyze dynamic simulations of movement. *Biomedical Engineering, IEEE Transactions on*, 54(11):1940 –1950, November 2007.

[15] M. Desbrun, M. Meyer, P. Schröder, and A.H. Barr. Implicit fairing of irregular meshes using diffusion and curvature flow. In *Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, pages 317–324. ACM Press/Addison-Wesley Publishing Co., 1999.

[16] F. Dong, G.J. Clapworthy, M.A. Krokos, and J. Yao. An anatomy-based approach to human muscle modeling and deformation. *IEEE Trans. Vis. Comput. Graph.*, 8(2):154–17, 2002.

[17] AWJ Gielen, CWJ Oomens, PHM Bovendeerd, T. Arts, and JD Janssen. A finite element approach for skeletal muscle using a distributed moment model of contraction. *Computer Methods in Biomechanics and Biomedical Engineering*, 3(3):231–244, 2000.

[18] B. Gilles, L. Moccozet, and N. Magnenat-Thalmann. Anatomical modelling of the musculoskeletal system from mri. *Medical Image Computing and Computer-Assisted Intervention–MICCAI 2006*, pages 289–296, 2006.

[19] P. Hachenberger and L. Kettner. Boolean operations on 3d selective nef complexes: Optimized implementation and experiments. In *Proceedings of the 2005 ACM symposium on Solid and physical modeling*, pages 163–174. ACM, 2005.

[20] Susan Hert and Stefan Schirra. 3D convex hulls. In CGAL Editorial Board, editor, *CGAL User and Reference Manual*. 3.5 edition, 2009.

[21] H. Hoppe. Progressive meshes. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 99–108. ACM, 1996.

[22] T.R. Jenkyn, B. Koopman, P. Huijing, R.L. Lieber, and K.R. Kaufman. Finite element model of intramuscular pressure during isometric contraction of skeletal muscle. *Physics in medicine and biology*, 47:4043, 2002.

[23] T. Johansson, P. Meier, and R. Blickhan. A finite-element model for the mechanical analysis of skeletal muscles. *Journal of theoretical biology*, 206(1):131–149, 2000.

[24] W. Jung, H. Shin, and B.K. Choi. Self-intersection removal in triangular mesh offsetting. *Computer-Aided Design and Applications*, 1(1-4):477–484, 2004.

[25] M. Kojic, S. Mijailovic, and N. Zdravkovic. Modelling of muscle behaviour by the finite element method using hill's three-element model. *International journal for numerical methods in engineering*, 43(5):941–953, 1998.

[26] M. Q. Liu, F. C. Anderson, M. G. Pandy, and S. L. Delp. Muscles that support the body also modulate forward progression during walking. *J Biomech*, 39:2623–2630, 2006.

[27] S.A. Maas, B.J. Ellis, D.S. Rawlins, and J.A. Weiss. A comparison of febio, abaqus, and nike3d results for a suite of verification problems. 2009.

[28] S. P. Magnusson. Passive properties of human skeletal muscle during stretch maneuvers. A review. *Scand J Med Sci Sports*, 8:65–77, Apr 1998.

[29] SP Magnusson, E.B. Simonsen, P. Aagaard, J. Boesen, F. Johannsen, and M. Kjaer. Determinants of musculoskeletal flexibility: viscoelastic properties, cross-sectional area, emg and stretch tolerance. *Scandinavian journal of medicine & science in sports*, 7(4):195–202, 1997.

[30] Xavier Maurice, Anders Sandholm, Nicolas Pronost, Ronan Boulic, and Daniel Thalmann. A subject-specific software solution for the modeling and the visualization of muscles deformations. *Visual Computer*, 25:835–842, 2009.

[31] M.Damsgaard, J.Rasmussen, S.T.Christensen, E.Surma, and M. de Zee. Analysis of musculoskeletal systems in the anybody modeling system. *Simul. Model. Pract. Theory*, 14:1100–1111, 2006.

[32] T. Möller. A fast triangle-triangle intersection test. *Journal of graphics tools*, 2(2):25–30, 1997.

[33] W. Nef. *Beiträge zur Theorie der Polyeder: mit Anwendungen in der Computergraphik*, volume 1. Lang, 1978.

[34] R. R. Neptune, S. A. Kautz, and F. E. Zajac. Contributions of the individual ankle plantar flexors to support, forward progression and swing initiation during walking. *J Biomech*, 34:1387–1398, Nov 2001.

[35] S. Osher and R.P. Fedkiw. *Level set methods and dynamic implicit surfaces*, volume 153. Springer Verlag, 2003.

[36] S. Oudot, L. Rineau, and M. Yvinec. Meshing volumes bounded by smooth surfaces. In *Proceedings of the 14th International Meshing Roundtable*, pages 203–219. Springer, 2005.

[37] S. J. Piazza and S. L. Delp. Three-dimensional dynamic simulation of total knee replacement motion during a step-up task. *J Biomech Eng*, 123:599–606, Dec 2001.

[38] Nicolas Pronost, Anders Sandholm, and Daniel Thalmann. A visualization framework for the analysis of neuromuscular simulations. *Vis. Comput.*, 27:109–119, February 2011.

[39] C. C. Raasch, F. E. Zajac, B. Ma, and W. S. Levine. Muscle coordination of maximum-speed pedaling. *J Biomech*, 30:595–602, Jun 1997.

[40] Laurent Rineau, Stphane Tayeb, and Mariette Yvinec. 3D mesh generation. In CGAL Editorial Board, editor, *CGAL User and Reference Manual*. 3.5 edition, 2009.

[41] J. Schmid and N. Magnenat-Thalmann. MRI Bone Segmentation using Deformable Models and Shape Priors. In D. Metaxas, L. Axel, G. Szekely, and G. Fichtinger, editors, *MICCAI*, volume LNCS 5241, pages 119–126. Springer-Verlag Berlin Heidelberg, September 2008.

[42] J.A. Sethian. A fast marching level set method for monotonically advancing fronts. *Proceedings of the National Academy of Sciences of the United States of America*, 93(4):1591, 1996.

[43] G. Taubin. Curve and surface smoothing without shrinkage. In *Computer Vision, 1995. Proceedings., Fifth International Conference on*, pages 852–857. IEEE, 1995.

[44] G. Taubin. A signal processing approach to fair surface design. In *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, pages 351–358. ACM, 1995.

[45] J. Teran, E. Sifakis, S.S. Blemker, V. Ng-Thow-Hing, C. Lau, and R. Fedkiw. Creating and simulating skeletal muscle from the visible human data set. *IEEE Trans. Vis. Comput. Graph.*, 11(3):317–328, 2005.

[46] D. G. Thelen, E. S. Chumanov, M. A. Sherry, and B. C. Heiderscheit. Neuromusculoskeletal models provide insights into the mechanisms and rehabilitation of hamstring strains. *Exerc Sport Sci Rev*, 34:135–141, Jul 2006.

[47] P.S. Walker, J.S. Rovick, and D.D. Robertson. The effects of knee brace hinge design and placement on joint mechanics. *Journal of Biomechanics*, 21(11):965 – 967, 969–974, 1988.

[48] Gary T. Yamaguchi and Felix E. Zajac. A planar model of the knee joint to characterize the knee extensor mechanism. *Journal of Biomechanics*, 22(1):1 – 10, 1989.

[49] N.H. Yang, P.K. Canavan, H. Nayeb-Hashemi, B. Najafi, and A. Vaziri. Protocol for constructing subject-specific biomechanical models of knee joint. *Computer Methods in Biomechanics and Biomedical Engineering*, 2010.

[50] C.A. Yucesoy, B.H. Koopman, P.A. Huijing, and H.J. Grootenboer. Three-dimensional finite element modeling of skeletal muscle using a two-domain approach: linked fiber-matrix mesh model. *Journal of biomechanics*, 35(9):1253–1262, 2002.

[51] F. E. Zajac. Muscle and tendon: properties, models, scaling, and application to biomechanics and motor control. *Crit Rev Biomed Eng*, 17:359–411, 1989.

[52] F. E. Zajac and M. E. Gordon. Determining muscle's force and action in multi-articular movement. *Exerc Sport Sci Rev*, 17:187–230, 1989.

APPENDIX A

Software design

The pipeline has two data inputs from the same subject: the motion from OpenSim, and the segmented MRI data. These two data sources are processed in a different software module. The OpenSim musculoskeletal model is defined in a .osim file which is in XML format. This file is read by a Java application which interprets the model and converts the joint-based local transformations into world transformations in the coordinate system of the MRI data. It writes a file describing the motion of the bones from the MRI data that can easily be read by the other software module. The second and main software module consists of one application written in C++, which takes the raw segmented MRI data and the output of the Java application and produces an FEBio file after applying all the processing stages to the segmented MRI data.

A.1 Libraries

The software uses a number of libraries, of which the CGAL [2] and Carve [1] are the most important ones. The other libraries for the Java software module are:

- JUnit4: For creating unit tests.
- POI 3.6: For reading Excel files for configuration.

For C++, the libraries used in the final pipeline are:

- Carve 1.0
- CGAL 3.5.1
- Boost 1.42.0: For the filesystem module for operations on file paths.

During the development of the shrink to skeleton method (See 5.2.2) we also used the Taucs library for solving sparse linear systems of equations.

APPENDIX B

Configuration files

The main pipeline is configurable through two configuration files, 'setup.txt' and 'settings.txt'. The former specifies the experiment setup; which bones and muscles are to be used in the simulation. The latter specifies all other settings of the pipeline. Below we will list the configuration files of the experiment described in Section 6.3. The comments in the files are preceded by a hash '#' symbol.

B.1 File: setup.txt

```
# This file specifies which bones and muscles to use in the simulation
# syntax is: objectname [space] BONE/MUSCLE [space] RIGHT/LEFT

# *** BONES ***
pelvis BONE RIGHT
femur BONE RIGHT
tibiafibula BONE RIGHT
patella BONE RIGHT
footsimplified BONE RIGHT

# *** QUADRICEPS ***
#vastuslateralis MUSCLE RIGHT
#rectusfemoris MUSCLE RIGHT
#vastusmedialis MUSCLE RIGHT
#vastusintermedius MUSCLE RIGHT

# *** HAMSTRINGS ***
bicepsfemoris MUSCLE  RIGHT
semimembranosus MUSCLE RIGHT
#semitendinosus MUSCLE RIGHT
#gracilis MUSCLE RIGHT

# *** HIP MUSCLES ***
#ilio_psoas MUSCLE RIGHT
#gluteusminimus MUSCLE RIGHT
#gluteusmedius MUSCLE RIGHT

# *** CALF MUSCLES ***
#gastrocnemius  MUSCLE  RIGHT
#soleusachillestendon+0 MUSCLE  RIGHT
```

B.2 File: settings.txt

```
# WARNING: Only the objects specified in setup.txt are used in ANY of the operations.
# WARNING: If run.tests is turned on, ONLY tests will be run, and no other programs.

# NOTE: 1=true 0=false

paths.data = C:/dev/modgen10/Data

# run settings, these programs are executed sequentially
run.clearcache.surface = 0          # clear cache of surfaces
run.clearcache.mesh = 0             # clear cache of meshes
run.generatemusclesetupfiles = 0    # generates a file femur_tibiafibula_muscles.txt, in
    which all muscles are listed that are connected to both femur and tibia or fibula
run.createAttachmentIndex = 1
run.generateFootTendons = 0         # generates the tendons (in memory, not on disk), this
     is used in combination with run.secondsetup. The muscle for which to generate are
    specified in missingtendons.txt
```

```
# stages
run.preprocess.smoothThemUp = 1
run.preprocess.createAttachmentConvexHulls = 1
run.preprocess.createTendonConvexHulls = 1
run.preprocess.removeUnwantedComponents = 1
run.preprocess.generateFootTendons = 1
run.preprocess.resolveSelfIntersections = 1
run.preprocess.substractAllFromAllSIR = 1
run.preprocess.generateMeshes = 1
run.preprocess.volumeMeshCleanup1 = 1
run.preprocess.createMeshAttachmentIndicesSets = 1
run.preprocess.createMeshTendonIndicesSets = 1

run.preprocess.calcvolumes = 0      # recalculates the file volumes.txt, which is a cache
    of the volumes used to compare the sizes of objects
run.preprocess.meshsurfaces = 0     # regenerates the surfaces by first applying the CGAL
    mesh algorithm and then taking the boundary. The new surfaces overwrite the surface
    cache. Is ALWAYS turned off in the final version.
run.preprocess.resolveOverlaps = 0  # resolves the surface overlaps by pushing away. The
    new surfaces overwrite the surface cache. (Parameters: fixoverlaps.growFactor,
    preprocess.surfaces.usemeshsurfaces)
run.convertattachments = 0          # converts the attachment specification from index-
    based specification to points in space
run.generateFootReplacement = 0     # generates the foot replacement (in memory, not on
    disk), this is used in combination with run.secondsetup. This is always used, and
    takes little time, so don't bother turning it off.
run.secondsetup = 0                 # generates the FEBio setup file (Parameters:
    secondsetup.fixoverlaps, secondsetup.removeCgalRubbish, secondsetup.jointpath, run.
    secondsetup.outputname)
run.stageBasedSetup = 1
run.startsimulation = 1             # runs the FEBio setup file specified in
    stageBasedSetup.outputname

# programs used to get some statistics to put in the paper
# if any of these programs are activated, ONLY that one will be run, and nothing else
run.calculatePenetrationStats = 0     # generates a file penetrations.csv, which
    calculates the amount of overlap all objects have with eachother
run.checkAllObjectsForValidity = 0    # tests for each object if point (0,0,0) is inside (
    this should be false for all objects), if this is true, something is wrong with the
    object, because the Carve library cannot create a correct intersection test
    datastructure. Results saved in validity.csv
run.countMeshElements = 0             # generates mesh and surface statistics and saves to
     meshsizes.csv (only objects in setup.txt)

# secondsetup run settings
run.secondsetup.outputname = secondsetup_arnopush     # name of the febio setup file (
    without .feb), file will be placed in simulations directory
secondsetup.jointpath = jointpath3.txt                          # input file to load
    jointmovements from. Relative to Data\JointPath

# stagebased setup
stageBasedSetup.jointpath = jointpaint_prototype.txt

# programs to execute some code tests (only useful when testing the code)
# tests settings
run.tests = 0                         # if this is 1, only tests will be run, and all
    other programs specified above will be ignored
run.test.carve = 0
run.test.cubemesh = 0                 # creates a cube mesh and does some mesh operations
    on it
run.test.smoothing = 0                # tests smoothing algorithm
run.test.connectedcomponents = 0      # tests connected component class
run.test.takeTheseTwoFaces = 0        # takes two faces from a surface and puts them in a
    different file. Parameters specified below under takeTheseTwoFaces.*
run.test.soleus = 0                   # does some tests on the soleus muscle
run.test.createTendon = 0             # tests the tendon creation code
run.test.skin = 0                     # prototype of skin generation
run.test.pushAway = 0                 # tests the pushing of two surfaces. Parameters
    specified below under test.pushAway.*
run.test.meshing = 0                  # tests the meshing of an object. Parameter below
    under test.meshing.objectId
run.test.sets = 0                     # tests the union-find wrapper-class
run.test.surfacevertexrelations = 0
```

```
run.test.maths = 0
run.test.calcthickness = 0
run.test.minkowski = 0
run.test.ccsMatrix = 0
run.test.mtl4 = 0
run.test.smartShrink = 0
run.test.smartShrink2 = 0
run.test.convexHull = 0
run.test.normalizedMovement = 0
run.test.bsurface = 0
run.test.meshContraction = 1
run.test.connectivitySurgery = 1
run.test.surfaceVolume = 0
run.test.multigridContractionSolver = 0
run.test.volumeMeshShrink = 0
run.test.selfIntersectRemoval = 0
run.test.cgalDelaunay = 0
run.test.substraction = 0
run.test.rdt2 = 0
run.test.pusharno = 0

# test parameters
test.meshing.objectId = vastusintermedius MUSCLE RIGHT        # parameter for run.test.
    meshing program
test.pushAway.master = Stages/Smooth1/Right_femur_0.vtk           # parameters for run.
    test.pushAway program
test.pushAway.slave = Stages/Smooth1/Right_sartorius_0.vtk          #
test.pushAway.growfactor = 1
takeTheseTwoFaces.filename = debug/skin.vtk           # parameters for run.test.
    takeTheseTwoFaces
takeTheseTwoFaces.face1 = 10760                       #
takeTheseTwoFaces.face2 = 10759                       #
test.smartShrink.filename = Muscles/Right_vastusintermedius_0.vtk # Muscles/
    Right_AdductorLongus_0.vtk #
test.smartShrink.iterationCount = 1000
test.smartShrink.writeIterationEvery = 10
test.smartShrink.merge.epsilon = 0.1
test.convexHull.filename = debug/spier.vtk
test.volumeMeshShrink.filename = Debug/testSubstraction/result.vtk
test.selfIntersectRemoval.filename = Muscles/Right_vastusintermedius_0.vtk  # Debug/
    simplified vastusintermedius.vtk
test.cgalDelaunay.filename = Debug/torus.vtk
test.rdt2.filename = Debug/rdt2test.vtk
test.rdt2.triangleIndex = 15762
test.carve.filename1 = Muscles/Right_bicepsfemoris_0.vtk
test.carve.filename2 = Bones/Right_femur_0.vtk
test.smoothing.filename1 = Muscles/Right_vastusintermedius_0.vtk
test.smoothing.iterationCount = 8
test.smoothing.stepsInEachIteration = 25
test.pusharno.filename1 = Muscles/Right_amagnus_0.vtk
test.pusharno.filename2 = Bones/Right_pelvis_0.vtk
test.pusharno.grow = 1
test.pusharno.mutual = 0

test.shrinkBySkeleton.iterationCount = 1
test.shrinkBySkeleton.stepsInEachIteration = 50


# debug settings
debug.twoManifold.writeErrorFiles = 0     # saves the surface object to the file FullModel
    /Debug/not2manifold.vtk when a non-2 manifold surface is detected. Not used anymore,
    because the check was only made for when run.preprocess.meshsurfaces was still used,
    that generated non-2-manifold objects.

# febio settings
attachment.maxpointdistance = 5       # maximum distance between the calculated attachment
     point (done during run.convertattachments) and the mesh vertex potentially assigned
    to it.

selfIntersectRemoval.smoothing.iterationCount = 0
selfIntersectRemoval.smoothing.areaSize = 0
selfIntersectRemoval.smoothing.stepSize1 = 0.5
selfIntersectRemoval.smoothing.stepSize2 = -0.49999
```

```
# resolve degenerate triangles 2
rdt2.epsilon.edges = 0.1
rdt2.epsilon.angles = 0.005

# Stages

attachments.inputStage = AttachmentsRAW
tendons.inputStage = AttachmentsRAW/TendonIndices

smoothThemUp.smoothSteps = 1000
smoothThemUp.taubinSmoother = 1
smoothThemUp.taubin.substeps = 1
smoothThemUp.taubin.lambda = 0.33
smoothThemUp.taubin.mu = -0.331
smoothThemUp.inputStage = SurfacesRAW
smoothThemUp.outputStage = Smooth1

removeUnwantedComponents.inputStage = Smooth1
removeUnwantedComponents.outputStage = UnwantedComponentsRemoved

generateFootTendons.inputStage = UnwantedComponentsRemoved
generateFootTendons.outputStage = FootTendonsGenerated
footTendonGeneration.maxFaceAngle = 60
footTendonGeneration.maxZdistance = 10
footTendonGeneration.attachmentLength = 20

resolveDegenerateTriangles.epsilon = 0.1
selfIntersectRemoval.epsilon = 0.00000000001
resolveSelfIntersections.inputStage = FootTendonsGenerated
resolveSelfIntersections.outputStage = SIR

substractAllFromAllSIR.inputStage = SIR
substractAllFromAllSIR.outputStage = SubstractAllFromAllSIR
substractAllFromAllSIR.growFactor = 0.50
substractAllFromAllSIR.growSteps = 50
substractAllFromAllSIR.useCarve = 1
growAndSmooth.smoothSteps = 2
growAndSmooth.taubin.substeps = 1
growAndSmooth.taubin.lambda = 0.33
growAndSmooth.taubin.mu = -0.331


generateMeshes.inputStage = SubstractAllFromAllSIR
generateMeshes.outputStage = Meshes

volumeMeshCleanup1.inputStage = Meshes
volumeMeshCleanup1.outputStage = MeshesCleanedUp1

createAttachmentConvexHulls.growFactor = 1
createAttachmentConvexHulls.inputStage = Smooth1
createAttachmentConvexHulls.outputStage = Attachments

createTendonConvexHulls.growFactor = 0.5
createTendonConvexHulls.inputStage = Smooth1
createTendonConvexHulls.outputStage = Tendons

meshAttachmentIndices.inputStageAttachments = Attachments
meshAttachmentIndices.inputStageMeshes = Meshes
meshAttachmentIndices.minimumIndexCount = 100
meshAttachmentIndices.outputStage = MeshAttachmentIndices

meshTendonIndices.inputStageTendons = Tendons
meshTendonIndices.inputStageMeshes = Meshes
meshTendonIndices.outputStage = MeshTendonIndices

stageBasedSetup.inputStageMeshes = Meshes
stageBasedSetup.outputname = hamstringcombi_1.3  # __kneemotionoptimize1  #
    stageBasedSetup_6.39
stageBasedSetup.outputname.autoinc = 1
stageBasedSetup.bonesAttached = 1

# meshcontraction settings
```

```
meshcontraction.laplaceoperator = 1
meshcontraction.W_hMode = program       # values can be: paper (method from paper) or
    program (method deduced from the decompiled code)
meshcontraction.logObjects = 0
meshcontraction.logIteration = 11
test.meshContraction.filename = Muscles/Right_vastusintermedius_0.vtk
test.meshContraction.volumeThreshold = 1e-5
test.meshcontraction.maxIterations = 300

# febio debug
febio.debug = 1
febio.debug.elementId = 10364

# See febio documentation for details
febio.title = Experiment              # name of the simulation (inside the file, NOT the
    filename, which can be specified in run.secondsetup.outputname)
febio.dtol = 0.001                    # convergence tolerance on displacements
febio.etol = 0.01                     # convergence tolerance on energy
febio.rtol = 0                        # convergence tolerance on residual
febio.lstol = 0.9                     # convergence tolerance on line search
febio.max_refs = 15                   # max number of stiffness reformations
febio.max_ups = 10                    # max number of BFGS stiffness updates
febio.optimize_bw = 0                 # optimize banwidth of stiffness matrix
febio.restart = 0                     # generate restart file
febio.cmax = 1E+5                     # max condition number of the stiffness matrix
febio.analysis = static               # dynamic or static
febio.print_level = PRINT_PROGRESS
febio.linear_solver = pardiso
febio.pressure_stiffness = 0
febio.time_stepper.opt_iter = 10
febio.timesteps = 200                 # number of timesteps per second
febio.time_stepper.dtmin = 0.001      # minimum timestep size
febio.time_stepper.dtmax = 0.1        # maximum timestep size
febio.time_stepper.max_retries = 100  # maximum number of times the timestepper can reduce
     the size of the timestep
febio.restarter.failcount = 10

# overlap settings
fixoverlaps.growFactor = 1                # how many mm should the pushing surface grow
    before pushing, used during run.preprocess.resolveOverlaps
secondsetup.fixoverlaps = 0               # apply overlap check after meshes have been
    generated, used during run.secondsetup
secondsetup.removeCgalRubbish = 1         # remove disconnected elements generated by CGAL
     mesh algorithm. This options was added for surfaces with really bad quality, usually
    it is not nessecary to apply this option.
preprocess.surfaces.usemeshsurfaces = 0   # specifies if the output of run.preprocess.
    meshsurfaces should be the input for run.preprocess.resolveOverlaps

# mesh surface generation
meshsurface.smoothfactor = 0              # amount of smoothing used before mesh is
    generated in run.preprocess.meshsurfaces

# mesh generation settings
# see CGAL documentation http://www.cgal.org/Manual/3.5/doc_html/cgal_manual/Mesh_3/
    Chapter_main.html#introsec:param
muscle.facetAngle = 30
muscle.facetSize = 5
muscle.facetApproximation = 0.5
muscle.cellRadiusEdgeRatio = 1.25
muscle.cellSize = 3

skin.facetAngle = 30
skin.facetSize = 2
skin.facetApproximation = 1
skin.cellRadiusEdgeRatio = 4
skin.cellSize = 4

bone.facetAngle = 30
bone.facetSize = 15
bone.facetApproximation = 0.75
bone.cellRadiusEdgeRatio = 4
bone.cellSize = 100
```

```
fast.facetAngle = 15
fast.facetSize = 40
fast.facetApproximation = 1
fast.cellRadiusEdgeRatio = 40
fast.cellSize = 20

# material settings
# muscle material settings
# see febio documentation for meanings
muscle.material.type = neohookean
    muscle.material.neohookean.density =1
    muscle.material.neohookean.youngs_modulus = 120
    muscle.material.neohookean.poissons_ratio = 0.4

    muscle.material.isotropic_elastic.density =1
    muscle.material.isotropic_elastic.youngs_modulus = 0.075
    muscle.material.isotropic_elastic.poissons_ratio = 0.49

    muscle.material.mooneyrivlin.density = 1
    muscle.material.mooneyrivlin.c1 = 0.06
    muscle.material.mooneyrivlin.c2 = 0.12
    muscle.material.mooneyrivlin.k = 1800

tendon.material.type = neohookean
    tendon.material.neohookean.density =1
    tendon.material.neohookean.youngs_modulus = 1200
    tendon.material.neohookean.poissons_ratio = 0.4

    tendon.material.isotropic_elastic.density =1
    tendon.material.isotropic_elastic.youngs_modulus = 1.7
    tendon.material.isotropic_elastic.poissons_ratio = 0.49

    tendon.material.mooneyrivlin.density = 1
    tendon.material.mooneyrivlin.c1 = 60000
    tendon.material.mooneyrivlin.c2 = 10000
    tendon.material.mooneyrivlin.k = 80000

# contact settings
# see febio documentation for detailed explanations

febio.collisionavoidance = 1      # write the sliding interfaces to the febio file.
    Usually on.

sliding.type = sliding_with_gaps  # options: sliding_with_gaps, facet_to_facet or sliding2
sliding.laugon = 0                # use augmented lagrangian method
sliding.penalty = 5               # in case of laugon: scale langrange multiplier
    increment
                                  #    otherwise: multiplier to penetration distance to
                                      resolve intersection
sliding.auto_penalty = 1          # automatically determine penalty value, probably not
    useful when using laugon
sliding.two_pass = 0              # use two passes (always use this)
sliding.tolerance = 1             # aug. lagrangian solution tolerance 0=turnoff
sliding.gaptol = 0                # tolerance of gap value, gap should be smaller than
    this value, in absolute measure 0=turnoff
sliding.minaug = 0                # minimum number of augmentations 0=turnoff
sliding.maxaug = 10               # maximum number of augmentations, when maxaug is
    reached, FEBio will move to the next timestep,
                                  #    regardless of force and gap tolerances have been
                                      met
sliding.fric_coeff = 0            # friction coefficient, between 0 and 1
sliding.fric_penalty = 0          # friction penalty, only works with sliding_with_gaps
sliding.ktmult = 1                # scale factor for tangential stiffness
sliding.seg_up = 0                # unimportant, leave at 0

# tied interfaces are only used for tieing tendons to muscles
tied.penalty = 100000
tied.laugon = 0
tied.tolerance = 0.00001
```

Abstract

Studying human motion using musculoskeletal models is a common practice in the field of biomechanics. By using such models, recorded subject's motions can be analyzed in successive steps from kinematics and dynamics to muscle control. However simulating muscle deformation and interaction is not possible, but other methods such as a *finite element* (FE) simulation are very well suited to simulate deformation and interaction of objects. We present a pipeline that can combine these two, by automatically generating a FE simulation based on subject-specific segmented MRI data, and a motion performed by the same subject. The pipeline resolves several types of data inconsistencies: noise in the dataset is removed by smoothing, objects that contain self-intersecting parts are corrected, missing tendon geometries are generated automatically and overlaps between objects are resolved. Much effort was made to resolve overlaps in a meaningful way of which several methods are discussed. This report shows the different steps of the pipeline, such as solving overlaps in the segmented surfaces, generating the volume mesh and the connection to a musculoskeletal simulation. The pipeline is validated by recreating an experiment done on live subjects where passive hamstring resistance was measured and by comparing experimental results.