UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

FIB

Fast Soft-Tissue Deformations with FEM

Martín Garcia, Pol

Master's Thesis

Master in Innovation and Research in Informatics
Computer Graphics and Virtual Reality
June 2022

Supervisor: Susín Sánchez, Toni
Dpt. MAT

Abstract

Soft body simulation has been a very active research area in computer animation since Baraff and Witkin's 1998 work on cloth simulation, which led Pixar to start using such techniques in all of its animated movies that followed.

Many challenges in these simulations come from different roots. From a numerical point of view, deformable systems are large sparse problems that can become numerically unstable at surprising rates and may need to be modified at each time-step. From a mathematical point of view, hyperelastic models defined by continuum mechanics need to be derived, established and configured. And from the geometric side, physical interaction with the environment and self-collisions may need to be detected and introduced into the solver.

It is a fact that the Computer Graphics academia primarily focuses on offline methods, both for rendering and simulation. At the same time, the advances from the industry mainly apply to real-time rendering. However, we wondered how such high-quality simulation methods would map to a real-time use case.

In this thesis, we delve into the simulation system used by Pixar's *Fizt2* simulator, based on the Finite Element Method, and investigate how to apply the same techniques in real-time while preserving robustness and fidelity, altogether providing the user with some interaction mechanisms.

A 3D engine for simulating deformable materials has been developed following the described models, with an interactive interface that allows the definition and configuration of scenes and later interaction with the simulation.

Acknowledgements

I would like to thank all the people that have helped me, directly or indirectly, with this thesis during these stressful times.

First and foremost, I have to thank my thesis supervisor Toni Susín Sánchez. His guidance made doing this thesis possible, and I have to especially thank him for constantly pushing me to improve and for putting up with me in all the meetings where our conversation always strayed from the topic.

I also want to thank the people at the ViRVIG research group, and in particular Àlvar Vinacua Pla and Toni Chica Calaf. If it were not for their involvement with me, this thesis and my undergrad thesis would have never happened, as working with them pushed me further to become interested in computer graphics.

Finally, because being able to develop this thesis required more than academic assistance, I also want to thank my family and friends who provided me with the support and encouragement that I needed to carry on and finish the thesis. They have been too kind, supporting and caring, for words to describe.

Contents

1 Introduction **1**

1.1 Motivation and Objectives . . . 1
1.2 State of the Art . . . 2
1.3 Document structure . . . 3
1.4 Notation and Tensors . . . 4

2 Deformation evaluation **5**

2.1 Continuum mechanics fundamentals . . . 5
2.1.1 Deformation map . . . 5
2.1.2 Deformation gradient . . . 6
2.1.3 Deformation energy . . . 7
2.1.4 Stress tensors . . . 8
2.1.5 Deformation forces . . . 9
2.2 Strain evaluation . . . 10
2.2.1 *Cauchy-Green* strain . . . 10
2.2.2 Strain Invariants . . . 11
2.2.3 *Cauchy-Green* Strain Invariants . . . 11
2.2.4 S-centric invariants . . . 12
2.2.5 Lamé Parameters . . . 12
2.3 Deformation Energy Hessian . . . 13
2.3.1 Hessian from S-centric invariants energy . . . 14
2.4 Hyperelastic Constitutive Models . . . 14
2.4.1 *Neo-Hookean* . . . 14
2.4.2 Corotational . . . 15
2.4.3 Stable-*Neo-Hookean* . . . 16

3 The Finite Elements Method **17**

3.1 Tetrahedral elements . . . 17
3.2 Linear tetrahedral elements on energy density gradients . . . 18
3.3 Computing the deformation gradient . . . 19
3.4 Gradient of the deformation gradient . . . 21
3.5 FEM system formulation . . . 22
3.5.1 Explicit FEM . . . 22
3.5.2 Implicit *Backwards-Euler* FEM . . . 23
3.5.3 Damping forces . . . 24

4 Practical simulation **26**

4.1 Collisions and interaction . . . 26
4.1.1 External forces . . . 26
4.1.2 Velocity Constraints . . . 27
4.1.3 Position alteration . . . 28
4.1.4 Collision handling . . . 28

4.1.5 Friction . . . 29
4.1.6 Floating-point error . . . 30
4.2 Sparse matrices . . . 31
4.3 Preconditioned Conjugate gradient . . . 31
4.3.1 Preconditioning . . . 33
4.4 Adaptive time-steps . . . 33
4.5 Analytic Eigenanalysis of Energies . . . 34
4.5.1 Stable *Neo-Hookean* analysis . . . 35
4.5.2 Forcing positive-definite Hessians . . . 37

5 Implementation details **39**
5.1 Intersection queries . . . 39
5.2 Eigen Sparse considerations . . . 39
5.2.1 Sparse matrix block address indexing . . . 39
5.2.2 Avoiding memory reallocations . . . 40
5.2.3 Preconditioned Conjugate Gradient . . . 41
5.3 Optimized PPCG . . . 41
5.4 Parallelization . . . 42
5.5 CPU-GPU bandwidth . . . 42
5.6 Fast 3×3 SVD . . . 43

6 Results **44**
6.1 Implementation Screenshots . . . 44
6.2 Optimizations . . . 44
6.2.1 General analysis . . . 46
6.2.2 Parallelization . . . 48
6.2.3 Preconditioned Conjugate Gradient . . . 49
6.3 Stability . . . 49
6.3.1 Dynamic time-stepping . . . 50
6.3.2 Floating-point precision . . . 51
6.3.3 Degenerate tetrahedron meshes . . . 51
6.4 Energy models . . . 52
6.4.1 Performance . . . 52
6.4.2 Stability and behavior . . . 54
6.5 Conclusions . . . 55
6.6 Future work . . . 56

A Linear algebra **58**
A.1 Frobenius norm . . . 58
A.2 Tensor flattening . . . 58
A.3 Tensor double contraction . . . 59
A.4 Singular Value Decomposition . . . 60
A.5 Polar Decomposition . . . 60
A.6 Divergence of a vector field . . . 60
A.7 Tetrahedron volume . . . 61

B Derivations **62**

B.1 S-centric invariants 62

B.1.1 Invariant I_1 62

B.1.2 Invariant I_2 63

B.1.3 Invariant I_3 and $\frac{\partial J}{\partial \mathbf{F}}$ 63

B.1.4 Summary 63

B.2 Corotational Energy 64

B.3 Stable Neo-Hookean Energy 65

B.4 Analytic Eigensystems of Arbitrary isotropic energies 65

Bibliography **68**

1. Introduction

The field of computer animation with simulation started gaining traction since Terzopoulos' work in 1987 [31], where he defines the foundations for all the techniques we use nowadays. However, simulations were too unstable to be used robustly with large time steps, and too costly to run with small time steps.

That changed with Baraff and Witkin [1], who first introduced simulation with implicit integration with models that simplified managing such systems. It enabled the usage of constraints to enforce one-way-interaction (solid-deformable) and penalization forces for two-way interaction and self-collisions (deformable-deformable). This work allowed such simulations in production environments as the previous authors joined Pixar shortly after to work in *Monsters, Inc.* (2001).

Since then, computational physical simulation has been making its way into many different fields, becoming an essential tool. The times when simulations were only used to make engineering simulations are gone forever.

On the other side, interactive simulation is a topic of added complexity, but many fields benefit from it, from games to medical training. However, we need to consider also the requirements involved with such applications to create usable systems.

User interaction is one crucial issue that needs to be handled carefully. Offline simulators usually react to a scripted environment. However, an interactive application must respond to the user input, which cannot be predicted, while maintaining a stable and robust simulation online.

Efficiency is another essential point that needs to be assessed for the simulation, rendering, collision system, and others. To provide some context, Pixar's *Fitz2* simulator spends an average of 56.5% of the time of a frame simulation computing and solving collisions in the scene [15].

1.1 Motivation and Objectives

Interactive simulation is a fascinating topic because of its immediate results in complex phenomena and interactions. Multiple works try to handle the issues it entails, mainly by introducing simplifications in the Finite Elements Method or providing alternatives to the method altogether.

Personally, I became interested in interactive simulation after my Bachelor thesis [17] on simulation with the *Material Point Method*. This powerful technique offers a simpler algorithm to the competition that can simulate very complex phenomena in record times, which I also implemented in the GPU. However, each frame took seconds (if not minutes) to be simulated.

The knowledge and experience gained, together with my advisor's guidance, let me research a different method that is widely used across the industry, the Finite Elements Method.

The project aims to replicate, in real-time, some of the phenomena that production-level simulators can produce, while solving the inherent problems of interaction and real-time constraints.

Thus, every concept explained in this thesis has been implemented into `C++` code in a custom-made engine for validation of the involved methods.

1.2 State of the Art

The original Baraff and Witkin [1] paper can still be considered state-of-the-art, as it introduced all the basis of what we use today. It presented an implicit backward Euler formulation of the system to solve, easy to configure constraints directly encoded in the solver, and a custom preconditioned Conjugate Gradients algorithm to solve the linear system. It also introduced some guidelines to make the simulated objects react robustly to the environment and to themselves.

It is interesting to highlight that the Baraff-Witkin model was not explicitly formulated as a Finite Elements Method model; however, Theodore Kim [14] recently showed a modern analysis of the very same Baraff-Witkin model, discussing the adaptability of the method with customizable energies (both isotropic and anisotropic).

With acceleration in mind, many new simulated systems arose from the previous works, mainly focused on geometric multigrid simplifications. We would highlight Tamstorf et al. [30], which includes a new linear system formulation to include the constraints while not worsening its condition number.

For interactive simulation, the parametrization of the whole scene in a pre-compute step seems to be one of the best alternatives to FEM simulations, as Dough and Kayvon showed in [12]. Also, Zhisong et al. [6] present a GPU architecture for FEM simulations; meanwhile, Georgii et al. [7] simplify the formulation into a mass-spring system with an explicit integration in the GPU and use particle systems, which are alternatives that scale efficiently with the computing power of a graphics card. Meier et al. [19] offer a survey on real-time simulation for surgery simulation that still applies.

Other exciting approaches are from Barbi et al. [2] on editing an established animation with a deformable body by evaluating the cost of an edition, Jeřábková et al. [13] on interactively cutting deformable bodies, and Hildebrandt et al. [10] on key-frame control of a simulated body in a simplified system.

Collision detection for deformable models also needs to be considered. Due to the nature of deformable solids, we would need to recompute an acceleration data structure for collision detection after each modification. The alternatives rely on GPU parallelized implementations of massive pairwise intersection tests like Zhang et al. [32], or simplifications of the surface and recovery via interpolation like Civit et al. [5].

Multiple courses on deformable simulation are published and available. Theodore Kim and David Eberle's course [15] offers excellent explanations of the whole method and uncovers some of the optimizations and characteristics used in Pixar's *Fizt2* solver. A more gentle introduction to the maths and concepts behind computer animation is presented by Bargteil et al. [3]. And finally, a comprehensive guide to the FEM was presented by Sifakis and Barbic [27] with accurate and rigorous derivations of the whole method.

1.3 Document structure

The structure of this thesis is the following:

- Chapter 1 introduces the context and the previous works that define this thesis.
- Chapter 2 presents the necessary concepts to understand the formalisms for simulation of deformable materials, and the continuous dynamics knowledge used to achieve it.
- Chapter 3 explains the FEM formulation of the deformable system, and how we use it to simulate the materials.
- Chapter 4 enhances the basic FEM with utilities for practical simulation requirements, like collision interaction and performance optimizations.
- Chapter 5 describes how several features have been implement to achieve interactive simulation.
- Chapter 6 provides a final discussion and analysis of the topics discussed and implemented in this thesis.

Finally, the appendices review several mathematical concepts, derivations, and extra information to fully comprehend this thesis.

- Appendix A reviews several classical concepts of linear algebra and explains some tensorial notation and algebra that are used in this thesis. We highly recommend at least skimming through it.
- Appendix B provides some derivations and descriptions of some formulations used in the thesis.

1.4 Notation and Tensors

Here we describe the notation used in this thesis on its mathematical derivations to ease understanding and avoid confusion about the multiple elements. The table 1.1 briefs the different mathematical notations.

Type	Notation	Examples
Scalar	italics	a, t, v_x, Ψ
Vector	vector	$\vec{v}, \vec{u}$
Normalized vector	hat	$\hat{n}, \hat{e}_1$
Matrix	boldface	$\mathbf{F}, \boldsymbol{\sigma}$

Table 1.1: Brief of the notation used in the thesis.

All vectors used in the documents are column vectors by default:

$$\vec{v} = \begin{pmatrix} v_x \\ v_y \\ v_z \end{pmatrix} \tag{1.1}$$

For the matrices, access to the elements m_{ij} of matrix $\mathbf{M}$ is indexed by row i and column j. In the case of 3×3 matrices:

$$\mathbf{M} = \begin{bmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \end{bmatrix} \tag{1.2}$$

This thesis also deals with tensors up to the fourth-order. To compute with ease products between tensors of a different order, and simplify its analysis, we flatten such tensors following the notation of Golub and Van Loan [8], with the $\text{vec}(\cdot)$ operator. Much more insight into the vectorization operator can be seen in appendix A.2.

To give some intuition, for a second-order tensor of 2×2.

$$\mathbf{M} = \begin{bmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \end{bmatrix} \longleftrightarrow \text{vec}(\mathbf{M}) = \begin{pmatrix} m_{11} \\ m_{12} \\ m_{21} \\ m_{22} \end{pmatrix} \tag{1.3}$$

We also define the double contraction operator for tensors ":", which is a generalization of the dot product for tensors. The result is a tensor of less order than the operands. It requires that the two operands have the same dimensions in their two lowest orders. See appendix A.3 for a more in-depth explanation.

$$\mathbf{A} : \mathbf{B} = \sum_i \sum_j a_{ij} b_{ij} \tag{1.4}$$

In the case of 2×2 matrices it behaves as follows:

$$\mathbf{A} : \mathbf{B} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} : \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} = a_{11}b_{11} + a_{12}b_{12} + a_{21}b_{21} + a_{22}b_{22} \tag{1.5}$$

2. Deformation evaluation

This chapter presents the necessary background to understand how to mathematically represent deformations of a solid, following the rules of continuous dynamics, and how we measure and control the elasticity of a material.

We will only consider homogeneous solids whose material domain Ω, which represents the undeformed solid geometry, is well defined and has constant topology (no fractures).

2.1 Continuum mechanics fundamentals

Here we describe the behavior of an ideal continuum material domain Ω, according to its infinitesimal elements, along some time t. When we want to identify the undeformed configuration of the material domain it will be noted as Ω or Ω^0, and Ω^t will determine the configuration at time t.

For the sake of clarity, at first, we can assume that $\Omega \subset \mathcal{R}^3$. However, the elements of Ω can be identified by any number of degrees of freedom.

2.1.1 Deformation map

To see the evolution of the deformable body, we define a deformation map $\vec{\phi} : \Omega^0 \to \Omega^t$. Ideally, this function will be able to map any position of the undeformed configuration to its deformed location at a specific time t.

$$\vec{x} = \vec{\phi}(\vec{X}) \tag{2.1}$$

In the equation 2.1 we map some reference (undeformed) configuration $\vec{X} \in \Omega$, which are called material coordinates, to its corresponding new deformed location $\vec{x}$, called spatial coordinates. We will always identify material coordinates with capital letters, and spatial coordinates with lower case. This is shown in figure 2.1.

It is important to note that the deformation map is bijective, and one can recover the material coordinates from the spatial ones $\vec{X} = \vec{\phi}^{-1}(\vec{x})$.

Notice that we omit the time t in the function. One has to assume that time is a parameter **implicitly** expressed in the deformation map. When it becomes necessary, it will be noted explicitly.

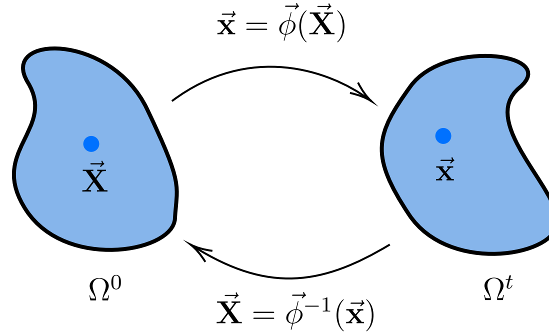


Figure 2.1: Deformation mapping function $\vec{\phi}$, which maps undeformed material coordinates in Ω^0 into spatial coordinates in a spatial configuration in Ω^t after some time t. Note that $\vec{\phi}$ is bijective.

2.1.2 Deformation gradient

Because the deformation map is an affine function, as we assume that the material does not break, its derivative is straightforward to derive:

$$\mathbf{F}(\vec{X}) = \frac{\partial \vec{\phi}}{\partial \vec{X}}(\vec{X}) \tag{2.2}$$

The deformation gradient $\mathbf{F}(\vec{X})$ is a function over the undeformed space and time that represents the infinitesimal deformation around $\vec{X}$. This deformation gradient, even though it is a function, in a 3-dimensional space can usually be interpreted as a $\mathcal{R}^{3\times3}$ matrix transformation.

In other words, $\mathbf{F}$ is the Jacobian matrix of the deformation map, which varies across Ω. To simplify the expressions to follow, we will express the gradient as a matrix $\mathbf{F}$.

To be more precise with the definition of the deformation gradient, if we define the material coordinate's components as $\vec{X} = (X, Y, Z)$ and the deformation map's as $\vec{\phi}(\vec{X}) = (\phi_x(\vec{X}), \phi_y(\vec{X}), \phi_z(\vec{X}))$, then the deformation gradient $\mathbf{F}$ in 3D is:

$$\mathbf{F} = \frac{\partial(\phi_x, \phi_y, \phi_z)}{\partial(X, Y, Z)} = \begin{bmatrix} \frac{\partial \phi_x}{\partial X} & \frac{\partial \phi_x}{\partial Y} & \frac{\partial \phi_x}{\partial Z} \\ \frac{\partial \phi_y}{\partial X} & \frac{\partial \phi_y}{\partial Y} & \frac{\partial \phi_y}{\partial Z} \\ \frac{\partial \phi_z}{\partial X} & \frac{\partial \phi_z}{\partial Y} & \frac{\partial \phi_z}{\partial Z} \end{bmatrix} \tag{2.3}$$

The deformation gradient is **an essential concept**, and almost everything dealing with the simulations discussed in this thesis will deal with it, as it defines a local infinitesimal deformation and how to produce it. In other words, and as depicted in figure 2.2, we can describe the following relationship.

$$d\vec{x} = \mathbf{F}\, d\vec{X} \tag{2.4}$$

Another key concept is that the deformation gradient $\mathbf{F}$ also explains the infinitesimal change in volume. This is explained by the determinant of $\mathbf{F}$, denoted J. Given a

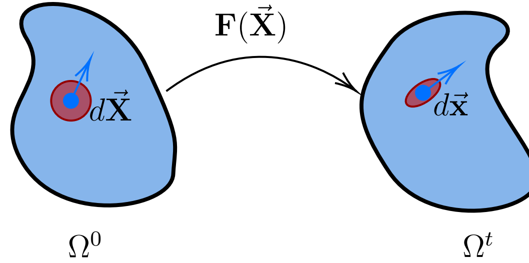


Figure 2.2: Deformation gradient $\mathbf{F}$, which expresses infinitesimal deformation of some material differential $d\vec{X}$, identified in red. Notice that the deformation gradient changes over space.

volume of the undeformed body V_0 and a volume after a deformation V, we can define:

$$dV = \det(\mathbf{F})\, dV_0 = J\, dV_0 \tag{2.5}$$

Note that when the deformation map $\vec{\phi}$ defines rotations and translations, $J = 1$. When $J < 1$ or $J > 1$, the material has been compressed or has been expanded.

Degenerate situations that cannot happen in real life need to be also considered. When $J < 0$, the material has been inverted altogether. If $J = 0$, we have a very degenerate case in which the portion of material has collapsed into a single point with no volume, which we must try to avoid.

2.1.3 Deformation energy

To run a simulation, we need to be able to compute forces from a deformation. One straightforward way to calculate forces intuitively is by defining the deformation's energy, called *strain* or elastic energy E, which will depend on the current deformation mapping $E[\vec{\phi}]$.

We will use the notation $E[\vec{\phi}]$, used by Sifakis [27], for the *strain* energy which means that the energy E is fully determined by the deformation mapping $\vec{\phi}$. Different parts of the material can suffer deformations of varying magnitude; thus, we define this *strain* energy locally.

To do this, we introduce an *energy density function* Ψ. This function of the deformation map measures the *strain* energy that the current infinitesimal deformation entails per unit of *undeformed* volume.

$$E[\vec{\phi}] = \int_{\Omega} \Psi[\vec{\phi}](\vec{X})\, d\vec{X} \tag{2.6}$$

Before we have discussed that the deformation given by $\vec{\phi}$ is defined by its deformation gradient $\mathbf{F}$. Thus, it is convenient to express always the *energy density* as a function of $\mathbf{F}$, i.e. $\Psi[\vec{\phi}](\vec{X}) = \Psi(\mathbf{F}(\vec{X})) = \Psi(\mathbf{F})$:

$$E[\vec{\phi}] = \int_{\Omega} \Psi(\mathbf{F})\, d\vec{X} \tag{2.7}$$

Note that the energy only depends on the current deformation of the material, and it is independent of the previous elastic energy. This property defines *hyperelastic* materials.

Using this *energy density* function gives us direct control over the simulated material. Giving an explicit formula for Ψ as a function of $\mathbf{F}$ is one of the two usual ways to define a material. Specific models will be seen in section 2.4.

The most direct expression for an elastic force $\vec{f}_e$ that we can obtain on any infinitesimal deformed portion of the material, where $\vec{x}$ denotes the degrees of freedom that define it, is the negative gradient of the *energy density* function Ψ, scaled by the original volume of $\vec{x}$ identified as $V_{\vec{X}}$:

$$\vec{f}_e(\vec{x}) = -V_{\vec{X}} \cdot \frac{\partial \Psi}{\partial \vec{x}}(\mathbf{F}) \tag{2.8}$$

At the moment we could only use this expression 2.8 to obtain forces from infinitesimal points. However, we will use the Finite Elements Method to discretize portions of the material and compute forces from the elastic energy described here.

2.1.4 Stress tensors

Because we cannot analyze our material's internal forces in infinitesimal points, we will handle such forces with traction $\vec{\tau}$ on infinitesimal surfaces.

Traction is defined as a force $\vec{f}$ acting on a deformed surface s with normal $\hat{n}$. In other words: the density of forces working on a particular surface.

$$\vec{\tau}^{(\hat{n})} = \frac{d\vec{f}}{ds} \tag{2.9}$$

However, we are more interested in defining this traction from the point of view of the undeformed surface S with normal $\hat{N}$.

$$\vec{T}^{(\hat{N})} = \frac{d\vec{f}}{dS} \tag{2.10}$$

Even more interesting is the fact that traction is not only valuable for defining forces on the surface of a material, but can define forces in interior sections of a body. For example, if we consider a domain Ω, for any internal point $\vec{X} \in \Omega \setminus \partial\Omega$ we could consider an infinitesimal cut through $\vec{X}$ perpendicular to $\hat{N}$ and compute the traction along with it.

We would like to have some tool that easily relates the force density and an arbitrary plane; these are *stress tensors*. For the topic that concerns us, we are interested in the first tensor of *Piola-Kirchhoff* $\mathbf{P}$, a unique matrix that explains traction for any boundary orientation.

$$\vec{T}^{(\hat{N})} = -\mathbf{P} \cdot \hat{N} \tag{2.11}$$

The first *Piola-Kirchhoff* stress tensor **P**, also called PK1, has some other interesting properties. In the first place, because it defines the force density in any direction, its divergence is equal to the force density (See divergence in appendix A.6). This is useful as it offers another way to obtain forces.

$$\vec{f} = (\nabla_{\vec{X}} \cdot \mathbf{P}) \cdot V_{\vec{X}} \tag{2.12}$$

Moreover, another essential property of the PK1 is that, for hyperelastic materials, it can be computed as a function of the deformation gradient **F** from the *elastic energy density* function's derivative.

$$\mathbf{P}(\mathbf{F}) = \frac{\partial \Psi}{\partial \mathbf{F}}(\mathbf{F}) \tag{2.13}$$

Using the PK1 is another typical way of defining a new material, which we can do by giving an explicit formula for **P** as a function of **F**. Together with, as stated previously, giving an explicit formula for Ψ as a function of **F** are the two typical styles of description for hyperelastic materials. Different materials are showcased in section 2.4

2.1.5 Deformation forces

Using the previous definition of forces from equation 2.8, we find ourselves aiming to compute forces given any deformed portion of material, with degrees of freedom $\vec{x}$, but we need to come up with an expression for $\frac{\partial \Psi}{\partial \vec{x}}(\mathbf{F})$. By applying the chain rule, we can expand this derivative as follows:

$$\frac{\partial \Psi}{\partial \vec{x}}(\mathbf{F}) = \frac{\partial \Psi}{\partial \mathbf{F}} \frac{\partial \mathbf{F}}{\partial \vec{x}}(\mathbf{F}) \tag{2.14}$$

This new transformation is incredibly powerful, as it uses $\frac{\partial \Psi}{\partial \mathbf{F}}$ which is equivalent to the PK1 **P** for hyperelastic materials. Notice that we have isolated the energy into a single term, independent of the degrees of freedom.

With this first part of the expression known, we only need to define $\frac{\partial \mathbf{F}}{\partial \vec{x}}$ which expresses the change of the deformation gradient **F** as its infinitesimal neighborhood changes (aka its degrees of freedom). As complex as it sounds, we will see that with the FEM, this term has a direct, energy-independent formula that always remains the same.

However, notice that $\frac{\partial \mathbf{F}}{\partial \vec{x}}$ is the derivative of a matrix by a vector, which will produce a tensor of order 3. Because we are dealing with a second-order tensor valued function with a scalar solution, its tensor derivative needs to use the ":" operator for tensor contraction. This operation is reviewed in appendix A.3, but essentially is a generalization of the dot product for tensors, where the result is a tensor of less order than the operands. In this case, the resulting $\frac{\partial \Psi}{\partial \vec{x}}(\mathbf{F})$ is a vector.

$$\frac{\partial \Psi}{\partial \vec{x}}(\mathbf{F}) = \frac{\partial \Psi}{\partial \mathbf{F}} : \frac{\partial \mathbf{F}}{\partial \vec{x}}(\mathbf{F}) \tag{2.15}$$

Knowing that operation, we can define the final equation to compute forces given any deformation by combining equations 2.8 and 2.15:

$$\vec{f}_e(\vec{x}) = -V_{\vec{X}} \cdot \frac{\partial \Psi}{\partial \mathbf{F}} : \frac{\partial \mathbf{F}}{\partial \vec{x}}(\mathbf{F}) \tag{2.16}$$

2.2 Strain evaluation

This section presents some terms used to construct *constitutive models* for hyperelastic materials (Section 2.4)), which model a mathematical description of the material behavior and expected response to deformations.

To design *constitutive models*, we need to be able to get descriptors to measure the magnitude of the deformation as a function of the deformation gradient $\mathbf{F}$. For example, we cannot use the raw matrix $\mathbf{F}$ as a metric because it also encodes perfect rotations. Ideally, we only want to evaluate the scaling that it encodes.

2.2.1 *Cauchy-Green* strain

One of the classical descriptors for strain is the right *Cauchy-Green* tensor $\mathbf{C}$:

$$\mathbf{C} = \mathbf{F}^T\mathbf{F} \tag{2.17}$$

This tensor is able remove the pure rotations from $\mathbf{F}$. To see how it does this one can imagine that we substitute $\mathbf{F}$ by its polar decomposition $\mathbf{F} = \mathbf{RS}$, where $\mathbf{R}$ is the rotational orthonormal component, and $\mathbf{S}$ the symmetric scaling component (See appendix A.5 for more information). Because $\mathbf{R}^T\mathbf{R} = \mathbf{I}$:

$$\mathbf{F}^T\mathbf{F} = (\mathbf{RS})^T(\mathbf{RS}) = \mathbf{S}^T\mathbf{R}^T\mathbf{RS} = \mathbf{S}^2 \tag{2.18}$$

Also, the eigenvalues of $\mathbf{C}$ are the squared three principal stretches that identify the deformation in 3 different orthonormal directions.

Cauchy-Green strain tensor

With the right *Cauchy-Green* tensor $\mathbf{C}$ we can define one of the most known strain descriptors, the *Cauchy-Green* strain tensor $\mathbf{E}$.

$$\mathbf{E} = \frac{1}{2}(\mathbf{F}^T\mathbf{F} - \mathbf{I}) \tag{2.19}$$

It succeeds in discarding the rotational components and keeps information on the strain (shear and stretch) but squared. One problem with the *Cauchy-Green* strain tensor $\mathbf{E}$ is that it is non-linear and will propagate this non-linearity into the *constitutive model* that uses it.

One alternative to the tensor $\mathbf{E}$ is to do its Taylor expansion to reach a linear expression. The resulting matrix is the *small strain tensor* ϵ:

$$\epsilon = \frac{1}{2}(\mathbf{F} + \mathbf{F}^T) - \mathbf{I} \tag{2.20}$$

Though linear and cheap to compute, the tensor ϵ is only reliable for very small motions. It will grow and produce noticeable wrong results as the deformations increase.

2.2.2 Strain Invariants

Instead of creating an *energy density* function with arbitrary terms, there is a set of usually used invariants to combine when creating such functions. These invariants are an already selected set, which comes with many already computed expressions to define *energy density* functions easily.

The usage of invariants is very appealing to build *energy density* functions because its derivatives and Hessians with respect to $\mathbf{F}$ have already been derived. These derivatives will become necessary when modeling an implicit solver in section 3.5.2.

Invariants also make it easier to analyze the expected behavior of an energy, and serve as guidance to better, more stable formulas.

2.2.3 *Cauchy-Green* Strain Invariants

The *Cauchy-Green* invariants describe different phenomena, and are generally easy to compute:

$$I_C = \mathrm{tr}(\mathbf{F}^T\mathbf{F}) \tag{2.21}$$

$$II_C = \mathrm{tr}((\mathbf{F}^T\mathbf{F})^2) \tag{2.22}$$

$$III_C = \det(\mathbf{F}^T\mathbf{F}) \tag{2.23}$$

On [15] there is an exemplary derivation of the meaning of each one of the invariants, but they can be roughly interpreted as the deformation in edges, areas and volume, respectively, of a cube affected by $\mathbf{F}$.

These invariants are very convenient to use because their derivatives with respect to $\mathbf{F}$ are already defined and ready to use. It makes it easier to later compute the derivatives of the PK1 $\mathbf{P}$ for later implicit integration.

The derivatives are the following. Notice that $\frac{\partial III_C}{\partial \mathbf{F}}$ is expensive to compute.

$$\frac{\partial I_C}{\partial \mathbf{F}} = 2\mathbf{F} \tag{2.24}$$

$$\frac{\partial II_C}{\partial \mathbf{F}} = 4\mathbf{F}\mathbf{F}^T\mathbf{F} \tag{2.25}$$

$$\frac{\partial III_C}{\partial \mathbf{F}} = 2\,III_C\,\mathbf{F}^{-T} \tag{2.26}$$

2.2.4 S-centric invariants

In Smith et al.[29] they propose a new set of invariants with more expressibility. Those invariants aim to be simpler, provide non-quadratic terms, and are also able to encode the *Cauchy-Green* invariants.

For this project, we used these invariants, and from here onwards, we will assume we are using these when mentioning invariants. They use the polar decomposition $\mathbf{F} = \mathbf{RS}$ (Appendix A.5).

$$I_1 = \text{tr}(\mathbf{S}) \tag{2.27}$$

$$I_2 = \text{tr}(\mathbf{F}^T\mathbf{F}) \tag{2.28}$$

$$I_3 = \det(\mathbf{F}) \tag{2.29}$$

In appendix B.1 we derive gradients and Hessians of this set of invariants. A compilation of the 3 gradients and 3 Hessians is in appendix B.1.4.

2.2.5 Lamé Parameters

To configure a *constitutive model*, one can use multiple parameters. However, it is typical to use a set of interconnected parameters known as the Lamé parameters for isotropic materials, λ and μ. The μ parameter controls length preservation, while λ controls volume preservation.

Because these parameters encode some obscure magnitudes, they can be encoded with an equivalent set of parameters. The Young modulus E and Poisson's ratio ν.

The Young modulus measures the stiffness of the material; it is defined by pressure (force per unit area) related to axial strain.

The Poisson ratio is a unitless measure that relates the deformation effect in perpendicular directions; in other words, ν is the fraction between the amount of strain in an axial direction that affects a transversal one. It is always defined in the range $[0, \frac{1}{2}]$.

The two set of parameters are related as follows:

$$\mu = \frac{E}{2(1+\nu)} \tag{2.30}$$

$$\lambda = \frac{E\nu}{(1+\nu)(1-2\nu)} \tag{2.31}$$

Note that when $\nu = \frac{1}{2}$, to compute λ we cause a division by zero. Specifying $\nu \sim \frac{1}{2}$ means that the energy must preserve the volume, but it will affect the stability of the simulation. This is the case of tissues, skin or muscle.

2.3 Deformation Energy Hessian

Up until now, we have derived a way to obtain forces from an *energy density* function. This is enough to simulate with explicit integration, as we have the energy and its first derivative, the PK1. However, simulating with this model can worsen the stability of the simulation very quickly. We will get into the details for this in the next chapter 3.

To get better simulations we will need the derivative of the force with respect to the deformed degrees of freedom, $\frac{\partial \vec{f}_e}{\partial \vec{x}}$. This derivative is a 2nd order tensor that explains how the force changes with the deformation. Deriving equation 2.8:

$$\frac{\partial \vec{f}_e}{\partial \vec{x}} = -V_0 \frac{\partial^2 \Psi}{\partial x^2} \tag{2.32}$$

It is important to remember that we are dealing with an unknown number of degrees of freedom at this moment; thus, we do not know the dimensionality of $\vec{x}$.

Because we do not know how to compute $\frac{\partial^2 \Psi}{\partial x^2}$, we need to find a way to write it in terms of something we do know or can compute: $\frac{\partial^2 \Psi}{\partial \mathbf{F}^2}$. There are many ways to get there, all with its compromises. For example, Müller et al. [22] compute this algorithmically by rotating the coordinate frame of each **F** every time and applying some "force offsets".

However, we take the analytical approach. We can do something very similar to what we did to obtain equation 2.16. By applying the chain rule on tensors, and assuming $\frac{\partial^2 \mathbf{F}}{\partial x^2} = 0$, we can get the following:

$$\frac{\partial^2 \Psi}{\partial x^2} = \frac{\partial \mathbf{F}}{\partial x}^T \frac{\partial^2 \Psi}{\partial \mathbf{F}^2} \frac{\partial \mathbf{F}}{\partial x} \tag{2.33}$$

We can only get equation 2.33 if we assume that $\frac{\partial \mathbf{F}}{\partial x}$ does not depend on $\vec{x}$. In other words, the deformation gradient is **linear** with respect to $\vec{x}$. This is very important, as it allowed us to simplify this expression. The reason for this simplification will be seen in section 3.2.

Thus, we require the second derivative of the *energy density* function, which is similar to the acceleration. This derivative, also called Hessian matrix, made of second-order partial derivatives, is defined equivalent to the derivative of the PK1 $\frac{\partial^2 \Psi}{\partial \mathbf{F}^2} = \frac{\partial \mathbf{P}}{\partial \mathbf{F}}$.

Also, the $\frac{\partial^2 \Psi}{\partial \mathbf{F}^2}$ is problematic because it is an order 4 tensor in $\mathcal{R}^{3\times3\times3\times3}$. To perform the product with the second-order tensors $\frac{\partial \mathbf{F}}{\partial x}$ we will flatten them. To see more on this operation, see appendix A.2.

$$\frac{\partial^2 \Psi}{\partial x^2} = \text{vec}\left(\frac{\partial \mathbf{F}}{\partial x}\right)^T \text{vec}\left(\frac{\partial^2 \Psi}{\partial \mathbf{F}^2}\right) \text{vec}\left(\frac{\partial \mathbf{F}}{\partial x}\right) \tag{2.34}$$

This is our final expression to compute the force derivative with respect to the deformed degrees of freedom. We only need to derive $\text{vec}(\frac{\partial^2 \Psi}{\partial \mathbf{F}^2})$, which is a matrix in $\mathcal{R}^{9\times9}$, from our selected *constitutive model*.

2.3.1 Hessian from S-centric invariants energy

Luckily, computing this order 4 tensor from an energy created from invariants is easy. If we develop the derivative $\frac{\partial^2\Psi}{\partial \mathbf{F}^2}$ and clump together the terms for each invariant, we get a straightforward expression to compute such tensor.

In the case of the S-centric invariants, from section 2.2.4, the derivatives and Hessians of each invariant with respect to the deformation gradient are simple expressions we can find derived in appendix B.1, and compiled in B.1.4.

To simplify the final expression we will use the flattened version of the derivatives, following Smith et al. notation [29]. To do this we define $\vec{\mathcal{F}} = \text{vec}(\mathbf{F})$. Thus, for all 3 invariants:

$$\frac{\partial I_i}{\partial \vec{\mathcal{F}}} = \text{vec}(\frac{\partial I_i}{\partial \mathbf{F}}) \qquad \frac{\partial^2 I_i}{\partial \vec{\mathcal{F}}^2} = \text{vec}(\frac{\partial^2 I_i}{\partial \mathbf{F}^2}) \tag{2.35}$$

Then, the final expression of the *energy density* Hessian is the following:

$$\text{vec}(\frac{\partial^2\Psi}{\partial \mathbf{F}^2}) = \sum_{i=1}^{3} \frac{\partial^2\Psi}{\partial I_i^2}\frac{\partial I_i}{\partial \vec{\mathcal{F}}}(\frac{\partial I_i}{\partial \vec{\mathcal{F}}})^T + \frac{\partial\Psi}{\partial I_i}\frac{\partial^2 I_i}{\partial \vec{\mathcal{F}}^2} \tag{2.36}$$

We have marked in blue the only entries we will have to derive by ourselves from the *energy density* function. Furthermore, these derivatives are scalars, which makes derivation even easier.

2.4 Hyperelastic Constitutive Models

There are an unlimited amount of potential hyperelastic *constitutive models*, and many papers are presented now and then with new exciting models. In this thesis, we have experimented with 2 of the most famous energies, *Neo-Hookean* and *Corotated*, and one relatively new energy that is more stable for large values of ν.

There are two different and interchangeable ways to describe a constitutive model: either as a formula for the PK1 stress tensor $\mathbf{P}$ as a function of the deformation gradient $\mathbf{F}$, or as a formula for the *energy density* Ψ as a function of also $\mathbf{F}$.

2.4.1 *Neo-Hookean*

There are many forms of *Neo-Hookean*, as it has been discussed and analyzed continuously since its original paper came out [20]. It is a type of energy that tries to preserve volume more than prevent elongations.

Here we will look at the known Bonet and Wood *Neo-Hookean* [4].

$$\Psi_{\text{BW08}} = \frac{\mu}{2}(||\mathbf{F}||_F^2 - 3) - \mu\log(J) + \frac{\lambda}{2}\log^2(J) \tag{2.37}$$

As a reminder, $||\cdot||_F$ is the Frobenius norm (See appendix A.1), and $J = \det(\mathbf{F})$ which explains the volume change (Equation 2.5).

Its PK1 tensor $\mathbf{P}_{\text{BW08}}$ is the derivative of Ψ_{BW08} with respect to $\mathbf{F}$, as seen on equation 2.13.

$$\mathbf{P}_{\text{BW08}}(\mathbf{F}) = \mu(\mathbf{F} - \frac{1}{J}\frac{\partial J}{\partial \mathbf{F}}) + \lambda \frac{\log J}{J}\frac{\partial J}{\partial \mathbf{F}} \tag{2.38}$$

Notice we have a new partial to solve $\frac{\partial J}{\partial \mathbf{F}}$. Luckily it is equivalent to $\frac{\partial I_3}{\partial \mathbf{F}}$, and is provided in appendix B.1.3.

Of course, Ψ_{BW08} can be rewritten in terms of the S-centric invariants of section 2.2.4:

$$\Psi_{\text{BW08}} = \frac{\mu}{2}(I_2 - 3) - \mu \log(I_3) + \frac{\lambda}{2}\log^2(I_3) \tag{2.39}$$

Notice that we cannot rewrite Ψ_{BW08} with the *Cauchy-Green* Invariants without introducing a $\sqrt{III_C}$ term, which makes things messier.

To build the *energy density* Hessian, we need to calculate the 3 derivatives and 3 Hessians of the PK1 with respect to the invariants.

$$\frac{\partial \Psi_{\text{BW08}}}{\partial I_1} = 0 \tag{2.40}$$

$$\frac{\partial \Psi_{\text{BW08}}}{\partial I_2} = \frac{\mu}{2} \tag{2.41}$$

$$\frac{\partial \Psi_{\text{BW08}}}{\partial I_3} = \frac{\lambda \log I_3 - \mu}{I_3} \tag{2.42}$$

$$\frac{\partial^2 \Psi_{\text{BW08}}}{\partial I_1^2} = 0 \tag{2.43}$$

$$\frac{\partial^2 \Psi_{\text{BW08}}}{\partial I_2^2} = 0 \tag{2.44}$$

$$\frac{\partial^2 \Psi_{\text{BW08}}}{\partial I_3^2} = \frac{\lambda - \lambda \log I_3 + \mu}{I_3^2} \tag{2.45}$$

Notice half of the terms end up being zero. Now, we insert those terms to equation 2.36 to get the final Hessian:

$$\text{vec}(\frac{\partial^2 \Psi_{\text{BW08}}}{\partial \mathbf{F}^2}) = \mu \mathbf{I}_{9\times 9} + \frac{\lambda - \lambda \log I_3 + \mu}{I_3^2}\frac{\partial I_3}{\partial \vec{\mathcal{F}}}(\frac{\partial I_3}{\partial \vec{\mathcal{F}}})^T + \frac{\lambda \log I_3 - \mu}{I_3}\frac{\partial^2 I_3}{\partial \vec{\mathcal{F}}^2} \tag{2.46}$$

Moreover, that is our final Hessian. The $\mu\mathbf{I}_{9\times 9}$ appears because $\frac{\partial^2 I_2}{\partial \vec{\mathcal{F}}^2} = 2\mathbf{I}_{9\times 9}$, it is one of the simpler derivatives of the S-centric invariants.

Now we could plug this Hessian into our simulator, and we would be good to go.

2.4.2 Corotational

The Corotational energy is one of the more widely used energies in the industry and academia. It was invented in mechanical engineering by Rankin and Brogan [24] in 1986, and rediscovered for computer animation in early 2000 by Müller et al. [21].

This *energy density* function is classically known as:

$$\Psi_{\text{Co}} = \frac{\mu}{2}||\mathbf{F} - \mathbf{R}||_F^2 + \frac{\lambda}{2}\text{tr}^2(\mathbf{S} - \mathbf{I}) \tag{2.47}$$

Notice that this energy does not have any term with J; thus, the biggest flaw of this energy is that it does not take into account the volume of the deformation, but has an approximation in its rightmost term. However, it is an stable energy, simple to compute, and has an appropriate behavior in almost all situations (the exception being when $\nu \sim \frac{1}{2}$).

This energy, its PK1 and Hessian can be expressed with the S-centric invariants as follows:

$$\Psi_{\text{Co}} = \frac{\mu}{2}(I_2 - 2I_1 + 3) + \frac{\lambda}{2}(I_1^2 - 6I_1 + 9) \tag{2.48}$$

$$\mathbf{P}_{\text{Co}} = (\lambda I_1 - 3\lambda - \mu)\frac{\partial I_1}{\partial \mathbf{F}} + \frac{\mu}{2}\frac{\partial I_2}{\partial \mathbf{F}} \tag{2.49}$$

$$\text{vec}(\frac{\partial^2 \Psi_{\text{Co}}}{\partial \mathbf{F}^2}) = \lambda \frac{\partial I_1}{\partial \vec{\mathcal{F}}}(\frac{\partial I_1}{\partial \vec{\mathcal{F}}})^T + (\lambda I_1 - 3\lambda - \mu)\frac{\partial^2 I_1}{\partial \vec{\mathcal{F}}^2} + \frac{\mu}{2}\frac{\partial^2 I_2}{\partial \vec{\mathcal{F}}^2} \tag{2.50}$$

The derivations for this energy, as well as the rewriting in terms of the S-centric invariants, can be found in appendix B.2.

2.4.3 Stable-*Neo-Hookean*

This model, presented by Smith et. al [28], tries to fix the stability and behavior problem of *Neo-Hookean* energies, like the one from section 2.4.1, when $\nu \sim \frac{1}{2}$.

Here, we present the version of this energy used at Pixar according to [15].

$$\Psi_{\text{SNH}} = \frac{\mu}{2}(I_2 - 3) - \mu(I_3 - 1) + \frac{\lambda}{2}(I_3 - 1)^2 \tag{2.51}$$

$$\mathbf{P}_{\text{SNH}} = \frac{\mu}{2}\frac{\partial I_2}{\partial \mathbf{F}} + (\lambda I_3 - \lambda - \mu)\frac{\partial I_3}{\partial \mathbf{F}} \tag{2.52}$$

$$\text{vec}(\frac{\partial^2 \Psi_{\text{SNH}}}{\partial \mathbf{F}^2}) = \frac{\mu}{2}\frac{\partial^2 I_2}{\partial \vec{\mathcal{F}}^2} + \lambda \frac{\partial I_2}{\partial \vec{\mathcal{F}}}(\frac{\partial I_2}{\partial \vec{\mathcal{F}}})^T + (\lambda I_3 - \lambda - \mu)\frac{\partial^3 I_1}{\partial \vec{\mathcal{F}}^2} \tag{2.53}$$

The derivation process for this energy is described in appendix B.3.

3. The Finite Elements Method

The Finite Elements method is a technique and a justification to faithfully approximate ordinary differential equations, such as those that describe movement.

This method is based on the idea that we can have a finite number of degrees of freedom to analyze a continuum and use interpolation to reconstruct the function everywhere.

Choosing the interpolation function significantly impacts the FEM form of the equations. A linear interpolation function will result in simpler systems, but it does not provide continuous derivative (C^1). Meanwhile, a quadratic interpolation function produces better results with C^1 continuity at the cost of more complexity.

We will rasterize the degrees of freedom into elements to work with them. An element is defined by a set of degrees of freedom and an interpolation function to retrieve any property defined inside of the element.

3.1 Tetrahedral elements

We have used tetrahedral elements for this thesis, the simplest elements one can use with volume.

Each tetrahedral element has 12 degrees of freedom: 4 nodes with 3 components each, as shown in the figure 3.1. Notice we call them nodes instead of vertices.

A tetrahedral mesh is a set of tetrahedrons that have shared nodes. These meshes can represent an arbitrary shape and volume, building an interconnected net of tetrahedrons.

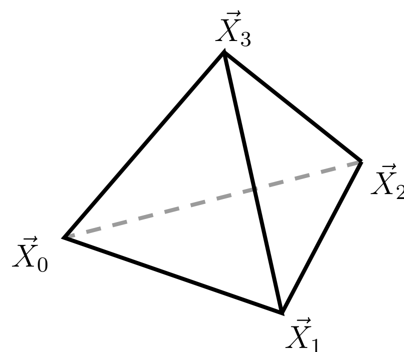


Figure 3.1: Tetrahedral element. Determined by 4 material coordinates $\vec{X}_i$.

Generating tetrahedral meshes is out of the scope of this thesis; however, we give some intuition about how they can be generated. We can create elements by extruding on an arbitrary surface mesh. If the mesh is watertight and a 2-manifold (i.e., there are no holes and all faces are orientable), we can fill the volume enclosed by the mesh with tetrahedrons using an algorithm akin to Delaunay triangulation.

In our particular case, we have used TetGen [26] to generate tetrahedral meshes.

Like with any geometry processing need, having good faces (or elements in this case) is preferred. We want to avoid very small elements, or with very acute angles, as those may produce numerical precision or/and stability issues.

3.2 Linear tetrahedral elements on energy density gradients

Elements for FEM need an interpolation function that defines them. Depending on the degree of the function, we get a trade-off between accuracy and complexity. As we have seen in the section 2.3, we used the fact that $\frac{\partial^2 \mathbf{F}}{\partial x^2} = 0$ to simplify an expression because the elements are linear, and thus $\mathbf{F}$ is too.

Because we are using **linear** tetrahedral elements, we will use barycentric coordinates as interpolation function to retrieve any property at any location inside the element.

Let us formalize this for the case of a tetrahedra, we define 3 barycentric coordinates $\vec{b} = \begin{pmatrix} u \\ v \\ w \end{pmatrix}$, and 4 basis functions:

$$\begin{aligned} \psi_0(\vec{b}) &= 1 - u - v - w \\ \psi_1(\vec{b}) &= u \\ \psi_2(\vec{b}) &= v \\ \psi_3(\vec{b}) &= w \end{aligned} \tag{3.1}$$

With the properties that $\sum_{i=0}^{3} \psi_i = 1$ and $\forall i, \psi_i \geq 0$. This constraints the function to define any interior property as a convex combination of the tetrahedron nodes, which defines a simplex shape.

Given the barycentric coordinates $\vec{b}$, any magnitude f_i defined at the 4 nodes can be linearly interpolated as:

$$f(\vec{b}) = \sum_{i=0}^{3} \psi_i(\vec{b}) f_i \tag{3.2}$$

We can use the barycentric interpolation to reconstruct any point inside the tetrahe-

dron both in the deformed and undeformed configuration:

$$\vec{X}(\vec{b}) = \sum_{i=0}^{3} \psi_i(\vec{b})\vec{X}_i \tag{3.3}$$

$$\vec{x}(\vec{b}) = \sum_{i=0}^{3} \psi_i(\vec{b})\vec{x}_i \tag{3.4}$$

3.3 Computing the deformation gradient

The deformation gradient **F**, defined in section 2.1.2, is one of the essential concepts in deformable material simulation, and now we want to get an analytic expression to compute it.

How we have seen in the previous equations 3.3 and 3.4, we can write $\vec{x}$ and $\vec{X}$ in terms of any barycentric coordinate $\vec{b}$, thus we can develop the deformation gradient definition:

$$\begin{aligned}
\mathbf{F} &= \frac{\partial \phi}{\partial \vec{X}} && (2.2)\\
&= \frac{\partial \vec{x}}{\partial \vec{X}} && (3.5)\\
&= \frac{\partial \vec{x}}{\partial \vec{b}}\frac{\partial \vec{b}}{\partial \vec{X}} && (3.6)\\
&= \frac{\partial \vec{x}}{\partial \vec{b}}\left(\frac{\partial \vec{X}}{\partial \vec{b}}\right)^{-1} && (3.7)
\end{aligned}$$

Having introduced the barycentric coordinates into the deformation gradient definition, we can substitute the interpolation function into the previous equations:

$$\begin{aligned}
\mathbf{F} &= \frac{\partial \vec{x}}{\partial \vec{b}}\left(\frac{\partial \vec{X}}{\partial \vec{b}}\right)^{-1} && (3.7)\\
&= \frac{\partial}{\partial \vec{b}}\left(\sum_{i=0}^{3} \psi_i(\vec{b})\vec{x}_i\right)\frac{\partial}{\partial \vec{b}}\left(\sum_{i=0}^{3} \psi_i(\vec{b})\vec{X}_i\right)^{-1} && (3.8)\\
&= \left(\sum_{i=0}^{3} \vec{x}_i \frac{\partial \psi_i(\vec{b})}{\partial \vec{b}}\right)\left(\sum_{i=0}^{3} \vec{X}_i \frac{\partial \psi_i(\vec{b})}{\partial \vec{b}}\right)^{-1} && (3.9)
\end{aligned}$$

We can see how both $\vec{X}$ and $\vec{x}$ do not depend on the value of $\vec{b}$, which allowed us to move the derivative inside the sum. What is even more interesting, is that we can

represent the previous formula 3.9 as a product of matrices:

$$\mathbf{F} = \left(\sum_{i=0}^{3} \vec{x}_i \frac{\partial \psi_i(\vec{b})}{\partial \vec{b}}\right) \left(\sum_{i=0}^{3} \vec{X}_i \frac{\partial \psi_i(\vec{b})}{\partial \vec{b}}\right)^{-1} \tag{3.9}$$

$$= \mathbf{xH}(\mathbf{XH})^{-1} \tag{3.10}$$

The matrices $\mathbf{X}$ and $\mathbf{x}$ are just the undeformed and deformed coordinates of the nodes put together in columns:

$$\mathbf{X} = \left[\begin{array}{c|c|c|c} \vec{X}_0 & \vec{X}_1 & \vec{X}_2 & \vec{X}_3 \end{array}\right] \tag{3.11}$$

$$\mathbf{x} = \left[\begin{array}{c|c|c|c} \vec{x}_0 & \vec{x}_1 & \vec{x}_2 & \vec{x}_3 \end{array}\right] \tag{3.12}$$

The Hessian matrix $\mathbf{H}$ seems very complex, but because our basis functions of equation 3.1 are simple and linear, it becomes a very simple matrix:

$$\mathbf{H} = \begin{bmatrix} \frac{\partial \psi_0}{\partial u} & \frac{\partial \psi_0}{\partial v} & \frac{\partial \psi_0}{\partial w} \\ \frac{\partial \psi_1}{\partial u} & \frac{\partial \psi_1}{\partial v} & \frac{\partial \psi_1}{\partial w} \\ \frac{\partial \psi_2}{\partial u} & \frac{\partial \psi_2}{\partial v} & \frac{\partial \psi_2}{\partial w} \\ \frac{\partial \psi_3}{\partial u} & \frac{\partial \psi_3}{\partial v} & \frac{\partial \psi_3}{\partial w} \end{bmatrix} \tag{3.13}$$

$$= \begin{bmatrix} -1 & -1 & -1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{3.14}$$

To finish up, we can multiply the hessian with $\mathbf{x}$ and $\mathbf{X}$, and set the computation of the deformation gradient as the product:

$$\mathbf{F} = \mathbf{xH}(\mathbf{XH})^{-1} \tag{3.10}$$

$$= \mathbf{D}_s \mathbf{D}_m^{-1} \tag{3.15}$$

Where $\mathbf{D}_s$ and $\mathbf{D}_m$ are the spatial and material parts of the deformation gradient, computed as follows:

$$\mathbf{D}_m = \mathbf{XH} = \left[\begin{array}{c|c|c} \vec{X}_1 - \vec{X}_0 & \vec{X}_2 - \vec{X}_0 & \vec{X}_3 - \vec{X}_0 \end{array}\right] \tag{3.16}$$

$$\mathbf{D}_s = \mathbf{xH} = \left[\begin{array}{c|c|c} \vec{x}_1 - \vec{x}_0 & \vec{x}_2 - \vec{x}_0 & \vec{x}_3 - \vec{x}_0 \end{array}\right] \tag{3.17}$$

Notice how lucky we are with the previous expressions; as $\mathbf{D}_m$ is defined by the undeformed configuration of the tetrahedron, we can calculate and cache $\mathbf{D}_m^{-1}$ since the beginning of the simulation. Thus, each time we need the deformation gradient $\mathbf{F}$, we only need to put together $\mathbf{D}_s$ and multiply by our stored $\mathbf{D}_m^{-1}$.

3.4 Gradient of the deformation gradient

The term $\frac{\partial \mathbf{F}}{\partial \vec{x}}$ has appeared before on the *deformation energy* Hessians of section 2.3. It is a required term we needed but could not define. Now that we have stated we are using linear tetrahedral elements, we can define it.

Because $\vec{x}$ has 12 degrees of freedom (4 nodes of 3 components each), $\frac{\partial \mathbf{F}}{\partial \vec{x}}$ is a 3rd order tensor of 12 matrices of 3×3.

For the sake of clarity, we define $\vec{x} = \begin{pmatrix} \vec{x}_0 \\ \vec{x}_1 \\ \vec{x}_2 \\ \vec{x}_3 \end{pmatrix} = \begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_{11} \end{pmatrix}$. Thus, the gradient of the deformation gradient is:

$$\frac{\partial \mathbf{F}}{\partial \vec{x}} = \begin{bmatrix} \left[\frac{\partial \mathbf{F}}{\partial x_0}\right] \\ \left[\frac{\partial \mathbf{F}}{\partial x_1}\right] \\ \vdots \\ \left[\frac{\partial \mathbf{F}}{\partial x_{11}}\right] \end{bmatrix} \tag{3.18}$$

In order to compute such tensor, we can make us of the previous equation 3.15 $\mathbf{F} = \mathbf{D}_s \mathbf{D}_m^{-1}$, and derivate the right hand side:

$$\frac{\partial \mathbf{F}}{\partial \vec{x}} = \frac{\mathbf{D}_s}{\partial \vec{x}} \mathbf{D}_m^{-1} \tag{3.19}$$

Notice that only $\mathbf{D}_s$ depends on $\vec{x}$. Thus, we can define $\frac{\partial \mathbf{F}}{\partial \vec{x}}$ in terms only of the partials of $\frac{\mathbf{D}_s}{\partial \vec{x}}$. Because of our linear interpolation function, all 12 matrices are very simple:

$$\begin{aligned}
&\frac{\mathbf{D}_s}{\partial x_0} = \begin{pmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \quad \frac{\mathbf{D}_s}{\partial x_1} = \begin{pmatrix} 0 & 0 & 0 \\ -1 & -1 & -1 \\ 0 & 0 & 0 \end{pmatrix} \quad \frac{\mathbf{D}_s}{\partial x_2} = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{pmatrix} \\
&\frac{\mathbf{D}_s}{\partial x_3} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \quad \frac{\mathbf{D}_s}{\partial x_4} = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \quad \frac{\mathbf{D}_s}{\partial x_5} = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \\
&\frac{\mathbf{D}_s}{\partial x_6} = \begin{pmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \quad \frac{\mathbf{D}_s}{\partial x_7} = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix} \quad \frac{\mathbf{D}_s}{\partial x_8} = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{pmatrix} \\
&\frac{\mathbf{D}_s}{\partial x_9} = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \end{pmatrix} \quad \frac{\mathbf{D}_s}{\partial x_{10}} = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} \quad \frac{\mathbf{D}_s}{\partial x_{11}} = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}
\end{aligned} \tag{3.20}$$

With these previous partials we can reconstruct $\frac{\partial \mathbf{F}}{\partial \vec{x}}$ by multiplying each of these with $\mathbf{D}_m^{-1}$. This means that $\frac{\mathbf{F}}{\partial x_i} = \frac{\mathbf{D}_s}{\partial x_i} \mathbf{D}_m^{-1}$, for all $i \in [0, 12)$, which defines our final tensor $\frac{\mathbf{F}}{\partial \vec{x}}$, the gradient of the deformation gradient.

Because of the significant amount of zeros, this computation can be optimized by building directly the final tensor $\frac{\mathbf{F}}{\partial \vec{x}}$, instead of doing the matrix products.

3.5 FEM system formulation

As any motion based system, our FEM formulation will be based in the equations of motion, which states that some positions $\vec{x}$ will evolve over time t as follows:

$$\frac{\partial^2 \vec{x}}{\partial t^2} = \mathbf{M}^{-1} \vec{f}(\vec{x}, \frac{\partial \vec{x}}{\partial t}) \tag{3.21}$$

Where $\mathbf{M}$ is the mass matrix, and $\vec{f}$ are all the forces that take effect, as a function of $\vec{x}$ and $\frac{\partial \vec{x}}{\partial t}$. Notice also the velocity term $\vec{v} = \frac{\partial \vec{x}}{\partial t}$.

The forces $\vec{f}$ include different terms, and we define them here as:

$$\vec{f}(\vec{x}, \frac{\partial \vec{x}}{\partial t}) = \vec{f}_e(\vec{x}, \frac{\partial \vec{x}}{\partial t}) + \vec{f}_d(\vec{x}, \frac{\partial \vec{x}}{\partial t}) + \vec{f}_{\text{ext}} \tag{3.22}$$

Each component is defined as follows:

- $\vec{f}_e(\vec{x}, \frac{\partial \vec{x}}{\partial t})$: Elastic forces. Resulting from our *energy density* function. This force is defined in the previous equation 2.16.
- $\vec{f}_d(\vec{x}, \frac{\partial \vec{x}}{\partial t})$: Damping forces. These are other forces that seek to balance out our assumptions of hyperelasticity and try to cancel out oscillation by dissipating energy from the system. It will be defined in the section 3.5.3.
- $\vec{f}_{\text{ext}}$: External forces that affect the system externally. This may mean gravity or collisions, for example.

To simulate equation 3.21 we need to transform it into a first-order differential equation:

$$\frac{\partial}{\partial t} \begin{pmatrix} \vec{x} \\ \vec{v} \end{pmatrix} = \begin{pmatrix} \vec{v} \\ \mathbf{M}^{-1} \vec{f}(\vec{x}, \vec{v}) \end{pmatrix} \tag{3.23}$$

3.5.1 Explicit FEM

The explicit formulation of the problem using FEM is the most simple and requires no complex analysis. We only need to set up the problem.

We want to simulate a the system at time $t+1$ from the state of the same system at the previous time t, where the interval of time between both states is Δt. Because of that, instead of focusing on the state of the system at a certain time $\vec{x}^{(t)}$ and $\vec{v}^{(t)}$, we will focus in the differences needed to reach a next step:

$$\vec{x}^{(t+1)} = \vec{x}^{(t)} + \Delta \vec{x} \tag{3.24}$$

$$\vec{v}^{(t+1)} = \vec{v}^{(t)} + \Delta \vec{v} \tag{3.25}$$

The explicit system can be defined by approximating equation 3.23 with Δt:

$$\begin{pmatrix} \Delta\vec{x} \\ \Delta\vec{v} \end{pmatrix} = \Delta t \begin{pmatrix} \vec{v}^{(t)} + \Delta\vec{v} \\ \mathbf{M}^{-1}\vec{f}(\vec{x}^{(t)}, \vec{v}^{(t)}) \end{pmatrix} \tag{3.26}$$

Where the state of the system is defined by:

- $\vec{x}^{(t)}$: Vector in $\mathcal{R}^{3n}$ with the positions of the n nodes at time t.
- $\vec{v}^{(t)}$: Vector in $\mathcal{R}^{3n}$ with the velocities of the n nodes at time t.

The mass matrix $\mathbf{M}$ is a diagonal matrix in $\mathcal{R}^{3n\times 3n}$ with the respective masses of each node. If the mass of the i-th node is m_i, then diag($\mathbf{M}$) =(m_0, m_0, m_0, m_1, m_1, m_1, ..., m_{n-1}, m_{n-1}, m_{n-1}).

Notice that forces $\vec{f}(\vec{x}^{(t)}, \vec{v}^{(t)})$ are simple to compute, as they depend on the current system state. The elastic forces $\vec{f_e}$ can be directly computed with their equation 2.16, using the original volumes of each tetrahedron to scale the force (See the appendix A.7 for the volume of a tetrahedron).

3.5.2 Implicit *Backwards-Euler* FEM

The implicit formulation of FEM is a little more complex and requires solving the following system:

$$\begin{pmatrix} \Delta\vec{x} \\ \Delta\vec{v} \end{pmatrix} = \Delta t \begin{pmatrix} \vec{v}^{(t)} + \Delta\vec{v} \\ \mathbf{M}^{-1}\vec{f}(\vec{x}^{(t)} + \Delta\vec{x}, \vec{v}^{(t)} + \Delta\vec{v}) \end{pmatrix} \tag{3.27}$$

This new formulation has the issue that is written in terms of the end result. Also, this equation 3.27 is non linear. We could use a non-linear solver, like the Newton-Rhapshon method, however the convergence of such method may be too slow for interactive applications.

We can fix the non-linearity by transforming the implicit formulation into the implicit *Backwards-Euler* formulation by performing a Taylor expansion on $\vec{f}$ of the first order. This formulation tries to reach a new state where taking a step backward ($-\Delta t$) brings the state back to the current state:

$$\vec{f}(\vec{x}^{(t)} + \Delta\vec{x}, \vec{v}^{(t)} + \Delta\vec{v}) \approx \vec{f}(\vec{x}^{(t)}, \vec{v}^{(t)}) + \frac{\partial\vec{f}}{\partial\vec{x}}(\vec{x}^{(t)}, \vec{v}^{(t)})\Delta\vec{x} + \frac{\partial\vec{f}}{\partial\vec{v}}(\vec{x}^{(t)}, \vec{v}^{(t)})\Delta\vec{v} \tag{3.28}$$

Such Taylor series induces in the worst case a quadratic error $O(\Delta\vec{x}^2 + \Delta\vec{v}^2)$, which is acceptable as $\Delta\vec{x}$ and $\Delta\vec{v}$ should be small. Also, the error is more tolerable because of the implicit formulation's stability.

As the force $\vec{f}$ and its derivatives are always evaluated with the state $(\vec{x}^{(t)}, \vec{v}^{(t)})$, let us write $\vec{f}^{(t)} = \vec{f}(\vec{x}^{(t)}, \vec{v}^{(t)})$ to compact the equations:

$$\vec{f}(\vec{x}^{(t)} + \Delta\vec{x}, \vec{v}^{(t)} + \Delta\vec{v}) \approx \vec{f}^{(t)} + \frac{\partial\vec{f}^{(t)}}{\partial\vec{x}}\Delta\vec{x} + \frac{\partial\vec{f}^{(t)}}{\partial\vec{v}}\Delta\vec{v} \tag{3.29}$$

Substituting our approximation 3.29 into the system 3.27 results in the following linear system:

$$\begin{pmatrix} \Delta\vec{x} \\ \Delta\vec{v} \end{pmatrix} = \Delta t \begin{pmatrix} \vec{v}^{(t)} + \Delta\vec{v} \\ \mathbf{M}^{-1}\left(\vec{f}^{(t)} + \frac{\partial \vec{f}^{(t)}}{\partial \vec{x}}\Delta\vec{x} + \frac{\partial \vec{f}^{(t)}}{\partial \vec{v}}\Delta\vec{v}\right) \end{pmatrix} \tag{3.30}$$

Given the previous system 3.30, we can substitute $\Delta\vec{x} = \Delta t\left(\vec{v}^{(t)} + \Delta\vec{v}\right)$ into the second equation:

$$\Delta\vec{v} = \Delta t \mathbf{M}^{-1}\left(\vec{f}^{(t)} + \frac{\partial \vec{f}^{(t)}}{\partial \vec{x}}\Delta t\left(\vec{v}^{(t)} + \Delta\vec{v}\right) + \frac{\partial \vec{f}^{(t)}}{\partial \vec{v}}\Delta\vec{v}\right) \tag{3.31}$$

With that we only have one variable, $\Delta\vec{v}$, we can regroup to build a linear system we can actually solve:

$$\left[\mathbf{M} - \Delta t\frac{\partial \vec{f}^{(t)}}{\partial \vec{v}} - \Delta t^2\frac{\partial \vec{f}^{(t)}}{\partial \vec{x}}\right]\Delta\vec{v} = \Delta t\,\vec{f}^{(t)} + \Delta t^2\frac{\partial \vec{f}^{(t)}}{\partial \vec{x}}\vec{v}^{(t)} \tag{3.32}$$

Which can be interpreted as solving a common linear system $\mathbf{A}\vec{x} = \vec{b}$, where our unknown variable is $\Delta\vec{v}$. Once the linear system is solved, we can update $\vec{v}^{(t+1)}$ and $\vec{x}^{(t+1)}$ accordingly.

An interesting and crucial fact about the previous 3.32 system, is that its left hand side is a symmetric matrix. In section 4.3 we review how to solve such system.

Finally, it is important to note that on the literature the derivative $\frac{\partial \vec{f}^{(t)}}{\partial \vec{x}}$ is denoted $\mathbf{K}$, the elasticity stiffness matrix. The damping term $\frac{\partial \vec{f}^{(t)}}{\partial \vec{v}}$ is historically denoted with $\mathbf{C}$.

3.5.3 Damping forces

Up until now, we have included the term $\frac{\partial \vec{f}^{(t)}}{\partial \vec{v}}$ into our system, but we have not discussed where it comes from or what is it.

In mechanics, the damping affects an oscillatory system to reduce the oscillation along time, usually with a decay rate ζ. The same concept can be applied in the simulation of deformable materials, where because of the simulation with discrete time steps, we can produce permanent oscillations.

Damping formulations are usually hand-crafted to avoid such issues, as those are based on assumptions. One of the most common damping models is *Rayleigh damping*, which can be configured with 2 coefficients α and β. The configuration of the damping parameters entails a complex discussion of oscillatory mechanics. See Liu and Gorman for an interesting analysis [16].

The damping term is defined as follows:

$$\frac{\partial \vec{f}^{(t)}}{\partial \vec{v}} = -\alpha\mathbf{M} - \beta\frac{\partial \vec{f}^{(t)}}{\partial \vec{x}} \tag{3.33}$$

To provide at least a little insight on how damping works and can be configured, given an oscillation angular velocity ω, the damping rate of *Rayleigh damping* is:

$$\zeta = \frac{1}{2}\left(\frac{\alpha}{\omega} + \beta\omega\right) \tag{3.34}$$

Because ζ depends on the current oscillation speed, it is impossible to guarantee good damping during all the simulation if no dynamic parameters are used. We want to aim for $\zeta \in (0, 1]$, as $\zeta = 0$ means that the system is undamped, while $\zeta = 1$ means it is critically damped, i.e., leads to the rest state the fastest. Values of ζ outside that range can lead to overshooting, and cause an increase in the oscillation.

4. Practical simulation

With the knowledge of the previous chapters and the last implicit system formulated in the equation 3.32, we have the necessary tools to start simulating any hyperelastic material.

However, full-fledged simulation systems require more features so that the simulated objects can interact with the scene and each other.

4.1 Collisions and interaction

This section delves into introducing collision reactions to the simulation system. Because the implicit linear system 3.32 only works with some deformable primitives, we need to add a method to take into account:

- External forces: Per node external forces which modify the behavior locally. External Forces can model gravity or friction on nodes in contact with some surface.
- Velocity Constraints: Impose velocity constraints in the nodes. Constraints are used to introduce interaction by forcing nodes to behave according to user input. Directional constraints can also be used to force a node to stay and move only along one plane.
- Position alteration: Modify nodes' positions directly while maintaining the system stable. Moving nodes explicitly introduces artificial corrections, like moving nodes along some collider.

4.1.1 External forces

Introducing external forces to each node is trivial, as forces are already part of the original formulation.

Remembering the system equation 3.32:

$$\left[\mathbf{M} - \Delta t \frac{\partial \vec{f}^{(t)}}{\partial \vec{v}} - \Delta t^2 \frac{\partial \vec{f}^{(t)}}{\partial \vec{x}}\right] \Delta \vec{v} = \Delta t\, \vec{f}^{(t)} + \Delta t^2 \frac{\partial \vec{f}^{(t)}}{\partial \vec{x}} \vec{v}^{(t)} \tag{3.32}$$

We can configure all the forces that affect a particle with the term $\vec{f}^{(t)}$. So this is already supported by our default system.

4.1.2 Velocity Constraints

Constraints were introduced by the original Baraff and Witkin paper [1] to have more control over the behavior of some selected nodes.

The idea is to set the constrained nodes with infinite mass as a way to force nodes not to be affected by the system. Another way to see it is: if we formulated the system for a single node with inverse masses $\frac{\partial^2 \vec{x}}{\partial t^2} = \frac{1}{m}\vec{f}$, if $m = \infty$ then the acceleration of the node is 0.

However, we would like to apply constraints in arbitrary planes and directions. To do so we will constraint each i-th node, $i \in [0, n)$, with the following 3×3 matrix $\mathbf{S}_i$, which emulates the previous inverse masses example:

$$\mathbf{S}_i = \begin{cases} \mathbf{I} & \text{if no DoF is constrained} \\ \mathbf{I} - \hat{p}_i\hat{p}_i^T & \text{if 1 DoF } \hat{p}_i \text{ is constrained} \\ \mathbf{I} - \hat{p}_i\hat{p}_i^T - \hat{q}_i\hat{q}_i^T & \text{if 2 DoFs, } \hat{p}_i \text{ and } \hat{q}_i\text{, are constrained} \\ \mathbf{0} & \text{if all 3 DoFs are constrained} \end{cases} \tag{4.1}$$

This reflects any 3D constraint, and allows us to fix the particle and allow it to move along a plane or only a direction. We can bring together all small matrices $\mathbf{S}_i$ into symmetric matrix $\mathbf{S} \in \mathcal{R}^{3n \times 3n}$, where its diagonal 3×3 blocks correspond to the respective $\mathbf{S}_i$.

Constraints work together with a forced change of velocity $\vec{z}_i$ for each node. This variable allows us to force a node to reach the desired velocity at the end of the step. If a node i has been constrained in all its degrees of freedom, then $\Delta\vec{v}_i = \vec{z}_i$. In other words, we are solving the system while guaranteeing that $(\mathbf{I} - \mathbf{S})\Delta\vec{v} = (\mathbf{I} - \mathbf{S})\vec{z}$.

In the original Baraff and Witkin [1], they applied such constraints inside the linear solver; however, doing so reduced the condition number of the matrix because constraints did set to zero its rows and columns of the system. However, we used the following newer approach by Tamstorf et al. *Pre-filtered Preconditioned* formulation [30], which is integrated into the formulation of the implicit FEM problem.

Tamstorf et al. handle the linear system as a minimization problem with satisfayability. With some clever algebra manipulation and common pattern substitutions, they are able to integrate the constraints in the system. If we interpret system 3.32 as $\mathbf{A}\vec{x} = \vec{b}$:

$$(\mathbf{SAS} + \mathbf{I} - \mathbf{S})\vec{y} = \mathbf{S}\vec{c} \tag{4.2}$$

$$\vec{y} = \vec{x} - \vec{z} \tag{4.3}$$

$$\vec{c} = \vec{b} - \mathbf{A}\vec{z} \tag{4.4}$$

The idea behind this system is that with $\mathbf{SAS}$ we project the system $\mathbf{A}$ into a space defined by $\mathbf{S}$. Also, the term $\mathbf{I} - \mathbf{S}$ is added to improve the condition number of the system without modifying the expected solution.

Solving this new system requires some extra matrix-vector products, and even if we optimize the products with the very sparse matrix **S**, this cost is non-negligible. However, solving this system guarantees a speed-up in the linear solver, which makes this worth it.

4.1.3 Position alteration

To handle collisions, we require to move nodes to the surface of the collider they have intersected. We cannot do that with the velocities because this would introduce energy to the system. However, moving a node explicitly is not enough as the node's neighbors are not notified of such movement, and will behave randomly.

Baraff and Witkin [1] proposed solving a slightly different system. Instead of computing $\Delta\vec{x}$ with equation 3.30, we add a correction term in equation 4.5:

$$\Delta\vec{x} = \Delta t(\vec{v}^{(t)} + \Delta\vec{v}) \tag{3.30}$$

$$\Delta\vec{x} = \Delta t(\vec{v}^{(t)} + \Delta\vec{v}) + \vec{r} \tag{4.5}$$

The $\vec{r}$ represents a correction term that encodes the extra displacement (position alteration) that a node must have applied at the end of the step.

If we repeat the derivation of section 3.5.2, instead of using equation 3.32 as our linear system, we use:

$$\left[\mathbf{M} - \Delta t\frac{\partial \vec{f}^{(t)}}{\partial \vec{v}} - \Delta t^2\frac{\partial \vec{f}^{(t)}}{\partial \vec{x}}\right]\Delta\vec{v} = \Delta t\,\vec{f}^{(t)} + \Delta t^2\frac{\partial \vec{f}^{(t)}}{\partial \vec{x}}\vec{v}^{(t)} + \Delta t\frac{\partial \vec{f}^{(t)}}{\partial \vec{x}}\vec{r} \tag{4.6}$$

The new rightmost term $\Delta t\frac{\partial \vec{f}^{(t)}}{\partial \vec{x}}\vec{r}$ of the equation 4.6 propagates the alteration to neighbor nodes, and makes its velocities adapt accordingly.

It is necessary to note that the velocity of the altered nodes will not be affected by such, and the position must be modified after solving the linear system with the equation 4.5.

4.1.4 Collision handling

Collisions with external elements to the simulation must be detected, tracked, and handled correctly.

For detection, Kim and Eberle [15] discuss the usage of proximity queries as a robust method to detect intersections. Proximity queries allow checking if a node is close to some collider at any point, only requiring a position. However, this method has some drawbacks:

- Generating SDF of the colliders is expensive to compute and store; simplifications can be done, and only consider offsets and insets of the collider faces, but this can produce very degenerate regions on sharp angles.

- Another point is that these queries are expensive because we need not only to classify points according to some 3D extruded collider faces, but we need to handle the boundaries of these offsets and insets, so that sets of neighboring faces do not overlap.
- Self-intersections of the sets can happen in non-neighboring faces and need to be handled accurately, maybe returning multiple collider faces as a response to the query.

Because of this, we ended up using ray intersection tests. Given a node i-th, that has evolved from $\vec{x}_i^{(t)}$ to $\vec{x}_i^{(t+1)}$, we can create a ray with direction $\frac{\vec{x}_i^{(t+1)}-\vec{x}_i^{(t)}}{||\vec{x}_i^{(t+1)}-\vec{x}_i^{(t)}||}$, and maximum length $||\vec{x}_i^{(t+1)} - \vec{x}_i^{(t)}||$.

Ray intersection tests are fast, reliable, and quickly accelerated by spatial data structures. Pharr et al.'s Physically Based Rendering book [23] has a thorough survey on ray intersection and acceleration data structures, which we have used.

Once an intersection has been found with an external collider, we enforce a constraint on the node that traversed the collider, move the node above the surface with *position alteration*, and keep track of the node.

Nodes that are tracked are kept constrained only in the perpendicular direction of the collider. This means that the node can move tangentially across the collider freely.

The constraints are removed when we detect that the last step's residual constraint forces do not restrict the node from staying on the surface by pulling it above. This is depicted in the figure 4.1.

The residual constraint forces $\vec{e}$ for all nodes can be easily computed from the original system of equation 3.32 interpreted as $\mathbf{A}\Delta\vec{v} = \vec{b}$:

$$\vec{e} = \mathbf{A}\Delta\vec{v} - \vec{b} \tag{4.7}$$

If a particle is constrained, its residual $\vec{e}_i$ may be different from zero because we have solved for the system of section 4.1.2. This error $\vec{e}$ corresponds to the extra force being enforced by the constraint.

Notice that this incurs in 1 step of delay to free the constraint. However, in practice, it does not seem to have a significant effect if the time steps are small enough and the model has a considerable amount of elements.

4.1.5 Friction

Friction is a type of damping where the kinetic energy of the system is dissipated via sliding against bodies. There are many ways of modeling this behaviour, where the most accurate and accepted is Coulomb damping. However such damping requires computing the surface area if you want accurate responses.

We take the simplification that each node in contact with a collider will be subject to some friction damping proportional to its velocity. This is that a node will have a

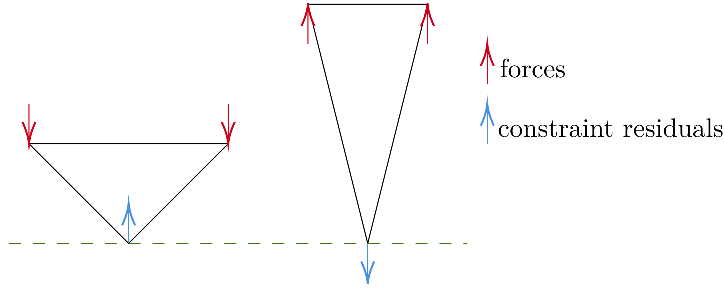


Figure 4.1: Constraint force residuals in different configurations. In the left case, the triangle is squished, and the constrain applies an upward force to keep the bottom node from moving down. In the right case, the triangle is being moved away, and the constraint prevents the bottom node from separating by pulling down. The constraint is removed in the right case.

friction damping force applied:

$$\vec{f}_{\text{fric}} = -k\vec{v} \tag{4.8}$$

Where $k \in [0, 1]$ is a constant of friction defined by the material the node is sliding against.

Because we want to use this force in our implicit system, we need its derivative:

$$\frac{\partial \vec{f}_{\text{fric}}}{\partial v} = -k\mathbf{I} \tag{4.9}$$

However, this equation expresses a damping force applied in any direction. Because we only want to damp the tangential direction of the surface, we project the damping into the plane defined by the normal of the surface $\hat{n}$ we are sliding against:

$$\frac{\partial \vec{f}_{\text{fric}}}{\partial v} = -k\left(\mathbf{I} - \hat{n}\hat{n}^T\right) \tag{4.10}$$

We can easily add this simple model of friction into our system, with almost no cost. However, setting the damping constant $k \to 1$ can make the system unstable as the damping equals the exact velocity we are trying to produce.

4.1.6 Floating-point error

Floating-point numbers are a discretization of real numbers, which may cause many different problems. In the case of collisions, we need to be very careful with how to apply the *position alteration*.

Suppose we move the nodes at the exact boundary of a face of a collider; at later frames, because of the tangential movement, the node may have gone through the collider due to floating-point error. This will lead to missing collisions in the future or block the node underneath the collider.

We need to keep track of the collider affecting a node and apply the *position alteration* with some small offsets to fix this.

It is essential to keep track of the collider and the node that collided against it. They are used to check when to release the collision, and to ascertain that the node is always at the right side of the collider.

4.2 Sparse matrices

The described system of equations 3.32 or 4.6 that we have to solve is very sparse, which means the number of non-zero entries of the matrix $\mathbf{A}$ is much lower than the number of zeroes. Instead of using dense matrices, we can use sparse ones.

A sparse matrix is a data structure that represents a matrix but only stores its non-zero components. Linear algebra libraries have optimized operations of all kinds with sparse matrices, making it desirable to use them.

Sparse matrices are optimal to use in their compressed form, where their non-zero entries are stored tight and contiguously in memory.

One drawback of sparse matrices is that they cannot have $\mathcal{O}(1)$ random access, and use binary searches to locate its entries. Another drawback is that when a matrix is compressed, one cannot insert new elements without paying $\mathcal{O}(m)$, where m is its number of non-zero entries.

To build the FEM system, we want to reutilize as much memory as possible and be able to insert and add the values into the sparse matrix without reallocating memory. In section 5.2 we will see how to explode properties of the FEM problem for efficient $\mathcal{O}(1)$ random access into the sparse matrix.

4.3 Preconditioned Conjugate gradient

To solve any linear system $\mathbf{A}\vec{x} = \vec{b}$, we require some algorithms to do so, called linear solvers. There are many different solvers for as many different problems; however, we will focus on the Conjugate Gradient Method (CG).

The perks of using CG is that it is an iterative solver that scales very well with sparse systems of linear equations. Here we will provide some insight into how it works and why it is so valuable. However, an in-depth introduction to the method is provided by Shewchuk [25].

CG is very similar in its concept to the classical Newton-Raphson method to solve non-linear equations. In Newton-Raphson method, given a function f from which we want to find its zeros, and its derivative f', we chose an initial point $x_{(0)}$, and move along the derivative to a next $x_{(1)}$ which should be closer to $f(x) = 0$. The method

is defined as follows at the i-th iteration:

$$x_{(i+1)} = x_{(i)} - \frac{f(x_{(i)})}{f'(x_{(i)})} \tag{4.11}$$

We are dealing with linear systems $\mathbf{A}\vec{x} = \vec{b}$, and instead of a gradient we could use matrix $\mathbf{A} \in \mathcal{R}^{n\times n}$'s eigenvectors as search directions. If we went through the eigenvectors in order, we could get closer to the solution very fast with the eigenvectors with a larger eigenvalue.

If, and only if, the matrix $\mathbf{A}$ is **positive-definite and symmetric**, we will always reach the best solution in at most n steps. Otherwise, convergence may be impossible in some cases.

To get some intuition of this requirement, positive-definiteness of matrix $\mathbf{A}$ means that :

- For any positive vector $\vec{x}$, $\vec{x}^T\mathbf{A}\vec{x} \geq 0$
- All the eigenvalues λ of $\mathbf{A}$ are positive.
- If $\mathbf{A}$ is also symmetric, the system $\mathbf{A}\vec{x} = \vec{b}$ has a unique solution, reachable from anywhere by moving along the eigenvectors $\vec{e}$.

However, computing eigenvectors is too expensive. Instead, we will start in a specific direction and continue in new orthogonal directions, using Gram-Schmidt orthogonalization. At most, we will still reach the solution in n steps, but not as fast as if we were moving along the eigenvectors.

The overall iterative algorithm is as follows:

$$\vec{d}_{(0)} = \vec{r}_{(0)} = \vec{b} - \mathbf{A}\vec{x}_{(0)} \quad \text{(Initial conditions)} \tag{4.12}$$

$$\alpha_{(i)} = \frac{\vec{r}^T_{(i)}\vec{r}_{(i)}}{\vec{d}^T_{(i)}\mathbf{A}\vec{d}_{(i)}} \quad \text{(Align direction with solution)} \tag{4.13}$$

$$\vec{x}_{(i+1)} = \vec{x}_{(i)} + \alpha_{(i)}\vec{d}_{(i)} \quad \text{(Update variable)} \tag{4.14}$$

$$\vec{r}_{(i+1)} = \vec{r}_{(i)} - \alpha_{(i)}\mathbf{A}\vec{d}_{(i)} \quad \text{(Get residual/error)} \tag{4.15}$$

$$\beta_{(i+1)} = \frac{\vec{r}^T_{(i+1)}\vec{r}_{(i+1)}}{\vec{d}^T_{(i)}\vec{r}_{(i)}} \quad \text{Gram-Schmidt} \tag{4.16}$$

$$\vec{d}_{(i+1)} = \vec{r}_{(i+1)} + \beta_{(i+1)}\vec{d}_{(i)} \quad \text{(orthonormal new direction)} \tag{4.17}$$

We might get very close to the solution before completing the n steps. We will stop the process when the residual's norm is below a threshold $||\vec{r}_{(i)}||^2 < \epsilon^2$.

This method will converge with a speed relative to the condition number κ of the matrix $\mathbf{A}$. If matrix $\mathbf{A}$ is sparse, and has m non zero entries, it will converge in $\mathcal{O}(m\sqrt{\kappa})$.

If we start with a good guess at the first iteration $\vec{x}_{(0)}$, the convergence will be much faster. In the simulation case, using the previous step's final solution is a good starting value.

4.3.1 Preconditioning

Preconditioning is a technique used to improve the condition number of a matrix even more. If we have some matrix $\mathbf{P}$ which is "similar" to $\mathbf{A}$, positive-definite, symmetric and easy to invert, then we can solve the following system instead:

$$\mathbf{P}^{-1}\mathbf{A}\vec{x} = \mathbf{P}^{-1}\vec{b} \tag{4.18}$$

In the case that $\mathbf{P}$ is similar in solutions to $\mathbf{A}$, $\kappa(\mathbf{P}^{-1}\mathbf{A}) \ll \kappa(\mathbf{A})$. However, $\mathbf{P}^{-1}\mathbf{A}$ might not be positive definite or symmetric. We can rewrite the CG algorithm into the *Untransformed Preconditioned Conjugate Gradient Method* by using some matrix identities and algebra manipulation.

$$\vec{r}_{(0)} = \vec{b} - \mathbf{A}\vec{x}_{(0)} \qquad \text{(Initial conditions)} \tag{4.19}$$

$$\vec{d}_{(0)} = \mathbf{P}^{-1}\vec{r}_{(0)} \qquad \text{(Initial preconditioned direction)} \tag{4.20}$$

$$\alpha_{(i)} = \frac{\vec{r}^T_{(i)}\mathbf{P}^{-1}\vec{r}_{(i)}}{\vec{d}^T_{(i)}\mathbf{A}\vec{d}_{(i)}} \qquad \text{(Align direction with solution)} \tag{4.21}$$

$$\vec{x}_{(i+1)} = \vec{x}_{(i)} + \alpha_{(i)}\vec{d}_{(i)} \qquad \text{(Update variable)} \tag{4.22}$$

$$\vec{r}_{(i+1)} = \vec{r}_{(i)} - \alpha_{(i)}\mathbf{A}\vec{d}_{(i)} \qquad \text{(Get residual/error)} \tag{4.23}$$

$$\beta_{(i+1)} = \frac{\vec{r}^T_{(i+1)}\mathbf{P}^{-1}\vec{r}_{(i+1)}}{\vec{d}^T_{(i)}\mathbf{P}^{-1}\vec{r}_{(i)}} \qquad \text{Gram-Schmidt preconditioned} \tag{4.24}$$

$$\vec{d}_{(i+1)} = \mathbf{P}^{-1}\vec{r}_{(i+1)} + \beta_{(i+1)}\vec{d}_{(i)} \qquad \text{(orthonormal new direction)} \tag{4.25}$$

For the simulations, we have used a simple Diagonal Preconditioner, i.e. $\mathbf{P}$ is a diagonal matrix, whose $\mathbf{P}^{-1}$ entries $p_{ii} = \frac{1}{a_{ii}}$.

4.4 Adaptive time-steps

In order to avoid instabilities with the system, we try to keep the Δt to a minimum, in a compromise between Δt magnitude and number of steps.

This needs to be handled carefully when dealing with interactive systems, as the framerate is variable and might have spikes. Also, adding more simulation steps in a single frame without proper care can make the system unstable, and the framerate can be affected.

In the first place, we need to stabilize the time in-between frames. To do so, we use the average of the time in-between the last few frames. This allows us to avoid sudden time spikes caused either by our simulation or operative system-related issues.

On the other hand, we would like to run at a specific framerate, where 100% of the computing power in a frame is being actively used. If a simulation step only requires a third of the frame computation time, we could run at least 2 steps with smaller Δt, making the simulation much more stable.

Our approach is to infer how much margin of time we have from the previous frame and check if a simulation step would fit in this margin. We increase the number of simulation steps at most once per frame if the average simulation step of the previous frame would fit in the remaining margin of time.

If the frame time is less than the objective frame rate, we decrease the number of simulation steps by one.

4.5 Analytic Eigenanalysis of Energies

As we have seen in section 4.3, we require that the matrix $\mathbf{A}$ of the system we are trying to solve is symmetric and positive-definite. However, our left hand side $\left[\mathbf{M} - \Delta t \frac{\partial \vec{f}^{(t)}}{\partial \vec{v}} - \Delta t^2 \frac{\partial \vec{f}^{(t)}}{\partial \vec{x}}\right]$ albeit symmetric, may not be positive-definite.

In the case that the matrix $\mathbf{A}$ does not fulfil such property, multiple solutions of the system exist. This can cause permanent oscillations, as the solver is indecisive towards which solution to go at each step, or an explosion to infinity.

Of course, analysing the eigenvalues of the full system is unfeasible. Nevertheless, because it is built from the composition of the Hessians $\frac{\partial^2 \Psi}{\partial \mathbf{F}^2}$, and no negative terms appear after substitution, the positive-definitiveness of the system is determined by those Hessians $\frac{\partial^2 \Psi}{\partial \mathbf{F}^2}$ because the sum of positive-definite matrices is guaranteed to be a positive-definite matrix.

The energies are also defined by the product of gradients $\frac{\partial \Psi}{\partial \mathbf{F}} \frac{\partial \Psi}{\partial \mathbf{F}}^T$, however this outer product is its own eigenvector and defined by 3 eigenvectors of $\frac{\partial^2 \Psi}{\partial \mathbf{F}^2}$.

By analysing the Hessians analytically we can do 2 different things:

- Study under which circumstances our system may stop being positive-definite.
- If a non positive-definite $\frac{\partial^2 \Psi}{\partial \mathbf{F}^2}$ is found, we can project the Hessian into a positive-definite one by clamping its eigenvalues to positive values, and reconstructing $\frac{\partial^2 \Psi}{\partial \mathbf{F}^2}$.

Luckily for us, a generalization of the eigensystems for any isotropic energy Hessian, defined with respect of the *Cauchy* or *S-centric* invariants, has already been derived by Smith et al. [29]. We use it to obtain the 9 eigenvalues and 9 $\mathcal{R}^9$ eigenvectors of $\frac{\partial^2 \Psi}{\partial \mathbf{F}^2}$. The eigenvectors can be consulted in appendix B.4.

The 9 general analytic eigenvalues are the following (the eigenvectors are left out). Notice, it uses σ as the singular values of $\mathbf{F}$ (See appendix A.4 for more information on the SVD).

$$\lambda_{0\ldots 2} = \text{eigenvalues}(\mathbf{A}) \tag{4.26}$$

$$\lambda_3 = \frac{2}{\sigma_x + \sigma_y}\frac{\partial\Psi}{\partial I_1} + 2\frac{\partial\Psi}{\partial I_2} + \sigma_z\frac{\partial\Psi}{\partial I_3} \tag{4.27}$$

$$\lambda_4 = \frac{2}{\sigma_y + \sigma_z}\frac{\partial\Psi}{\partial I_1} + 2\frac{\partial\Psi}{\partial I_2} + \sigma_x\frac{\partial\Psi}{\partial I_3} \tag{4.28}$$

$$\lambda_5 = \frac{2}{\sigma_x + \sigma_z}\frac{\partial\Psi}{\partial I_1} + 2\frac{\partial\Psi}{\partial I_2} + \sigma_y\frac{\partial\Psi}{\partial I_3} \tag{4.29}$$

$$\lambda_6 = 2\frac{\partial\Psi}{\partial I_2} - \sigma_z\frac{\partial\Psi}{\partial I_3} \tag{4.30}$$

$$\lambda_7 = 2\frac{\partial\Psi}{\partial I_2} - \sigma_x\frac{\partial\Psi}{\partial I_3} \tag{4.31}$$

$$\lambda_8 = 2\frac{\partial\Psi}{\partial I_2} - \sigma_y\frac{\partial\Psi}{\partial I_3} \tag{4.32}$$

The general symmetric matrix $\mathbf{A}$, which is a matrix that encodes the scaling only components of the energy, is defined as follows with the $\{i, j, k\}$ notation $(i \neq j \neq k)$.

$$a_{ii} = 2\frac{\partial\Psi}{\partial I_2} + \frac{\partial^2\Psi}{\partial I_2^2} + 4\sigma_i^2\frac{\partial^2\Psi}{\partial I_2^2} + \sigma_j^2\sigma_k^2\frac{\partial^2\Psi}{\partial I_3^2} + 4\sigma_i\frac{\partial^2\Psi}{\partial I_1\partial I_2} + 4I_3\frac{\partial^2\Psi}{\partial I_2\partial I_3} + 2\frac{I_3}{\sigma_i}\frac{\partial^2\Psi}{\partial I_3\partial I_1} \tag{4.33}$$

$$a_{ij} = \sigma_k\frac{\partial\Psi}{\partial I_3} + \frac{\partial^2\Psi}{\partial I_1^2} + \sigma_i\sigma_j\frac{\partial^2\Psi}{\partial I_2^2} + \sigma_k I_3\frac{\partial^2\Psi}{\partial I_3^2} + 2(I_1 - \sigma_k)\frac{\partial^2\Psi}{\partial I_1\partial I_2} + 2\sigma_k(I_2 - \sigma_k^2)\frac{\partial^2\Psi}{\partial I_2\partial I_3} + \sigma_k(I_1 - \sigma_k)\frac{\partial^2\Psi}{\partial I_3\partial I_1} \tag{4.34}$$

4.5.1 Stable *Neo-Hookean* analysis

Given the Stable *Neo-Hookean* energy of section 2.4.3:

$$\Psi_{\text{SNH}} = \frac{\mu}{2}(I_2 - 3) - \mu(I_3 - 1) + \frac{\lambda}{2}(I_3 - 1)^2 \tag{2.51}$$

The eigenvalues of $\frac{\partial^2 \Psi_{\text{SNH}}}{\partial \mathbf{F}^2}$ and its general scaling system are:

$$\lambda_3 = \mu + \sigma_z(\lambda(I_3 - 1) - \mu) \tag{4.35}$$
$$\lambda_4 = \mu + \sigma_x(\lambda(I_3 - 1) - \mu) \tag{4.36}$$
$$\lambda_5 = \mu + \sigma_y(\lambda(I_3 - 1) - \mu) \tag{4.37}$$
$$\lambda_6 = \mu - \sigma_z(\lambda(I_3 - 1) - \mu) \tag{4.38}$$
$$\lambda_7 = \mu - \sigma_x(\lambda(I_3 - 1) - \mu) \tag{4.39}$$
$$\lambda_8 = \mu - \sigma_y(\lambda(I_3 - 1) - \mu) \tag{4.40}$$

$$a_{ii} = \mu + \lambda\frac{I_3^2}{\sigma_i^2} \tag{4.41}$$
$$a_{ij} = \sigma_k(\lambda(2I_3 - 1) - \mu) \tag{4.42}$$

Analysing the matrix $\mathbf{A}$ is overly-complicated because guaranteeing that its eigenvalues are positive requires asserting that for any positive vector $\vec{v}$, $\vec{v}^T\mathbf{A}\vec{v} \geq 0$:

$$\begin{aligned} v^T\mathbf{A}\vec{v} = &- x(\sigma_z y(\mu - \lambda(2I_3 - 1)) - x(\mu + (I_3^2\lambda)/\sigma_x^2) + \sigma_y z(\mu - \lambda(2I_3 - 1))) \\ &- y(\sigma_z x(\mu - \lambda(2I_3 - 1)) - y(\mu + (I_3^2\lambda)/\sigma_y^2) + \sigma_x z(\mu - \lambda(2I_3 - 1))) \\ &- z(\sigma_y x(\mu - \lambda(2I_3 - 1)) - z(\mu + (I_3^2\lambda)/\sigma_z^2) + \sigma_x y(\mu - \lambda(2I_3 - 1))) \end{aligned} \tag{4.43}$$

However, we can consider the extremes of the material parametrization. When $\nu \to \frac{1}{2}$ then $\lambda \gg \mu$. Thus, we can modify the previous system by removing μ, and setting $\lambda = 1$ as we know that λ will always be positive, and it allows us to work with a simpler expression:

$$\begin{aligned} v^T\mathbf{A}\vec{v} \approx \;& x(I_3^2 x/\sigma_x^2 + \sigma_z y(2I_3 - 1) + \sigma_y z(2I_3 - 1)) \\ &+ y(\sigma_z x(2I_3 - 1) + I_3^2 y/\sigma_y^2 + \sigma_x z(2I_3 - 1)) \\ &+ z(\sigma_y x(2I_3 - 1) + \sigma_x y(2I_3 - 1) + I_3^2 z/\sigma_z^2) \end{aligned} \tag{4.44}$$

Here it is easier to see that whenever $I_3 \geq \frac{1}{2}$ the matrix is guaranteed to be positive definite. Remember that $I_3 = \det \mathbf{F} = \sigma_x\sigma_y\sigma_z$, which implies that if $I_3 > 0$ then all singular values are positive. Trying to derive much more from this complex system is difficult.

We can also analyse the other eigenvalues. For the $\lambda_{3\ldots5}$ we check when the eigenvalue will be zero.

$$\lambda_3 = \mu + \sigma_z(\lambda(\sigma_x\sigma_y\sigma_z - 1) - \mu) \geq 0 \tag{4.45}$$

Solving for σ_x yields the following root:

$$\sigma_x \geq \frac{\sigma_z(\lambda + \mu) - \mu}{\lambda\sigma_y\sigma_z^2} \tag{4.46}$$

If we do the same assumption than before that the material is volume preserving, we can assume the following.

$$\sigma_x \geq \frac{1}{\sigma_y\sigma_z} \tag{4.47}$$

This is very revelating, as it tells us that the system will remain stable while the compression along one direction is more than the inverse area of its cross-section.

In other words, if an element is compressed by n units in one direction, its cross-section area must be at most n^2 units for the Hessian to be positive-definite.

Something similar happens with the eigenvalues $\lambda_{6\ldots8}$, but slightly worse:

$$\begin{aligned}\lambda_6 &= \mu - \sigma_z(\lambda(I_3 - 1) - \mu) &(4.48)\\ &= \mu - \sigma_z(\lambda(\sigma_x\sigma_y\sigma_z - 1) - \mu) &(4.49)\end{aligned}$$

And solving $\lambda_6 \geq 0$ for σ_x:

$$\sigma_x \geq \frac{\sigma_z(\lambda + \mu) + \mu}{\lambda\sigma_y\sigma_z^2} \tag{4.50}$$

Notice that to guaranty such inequality, for great compressions along the cross-section we need to ensure a minimum elongation along our axis.

In conclusion, this material is always stable under moderate stretching, and stable under compression if its volume at most halved, or the compression inverse cross-section area is at most the compression factor along a direction.

4.5.2 Forcing positive-definite Hessians

Given the previous analytic analysis of the energy eigensystem, we could use it to detect and prevent non-positive-definite Hessians on the different elements.

Given the computed 9 eigenvalues, if we detect at least one of them is negative we clamp the eigenvalues to the positive space, and reconstruct the Hessian from these new eigenvalues and analytic eigenvectors.

This type of re-projection is not perfect, if there are multiple eigenvalues at zero there will be a deficiency in the number of orthogonal eigenvectors, which can led to problems.

To reconstruct a matrix $\mathbf{A}$ from a set of n eigenvectors $\vec{e}_i$ with eigenvalues λ_i we can write the following system:

$$\mathbf{A}\left(\begin{array}{c|c|c} \vec{e}_0 & \ldots & \vec{e}_{n-1} \end{array}\right) = \left(\begin{array}{c|c|c} \lambda_0\vec{e}_0 & \ldots & \vec{\lambda}_{n-1}e_{n-1} \end{array}\right) \tag{4.51}$$

We can rewrite it as follows, where $\mathbf{E}$ is a square matrix with the eigenvectors, and $\mathbf{D}$ a diagonal matrix with the eigenvalues.

$$\mathbf{AE} = \mathbf{ED} \tag{4.52}$$

Because our eigenvectors must be linearly independent, we consider $\mathbf{E}$ to be invertible, and thus our matrix $\mathbf{A}$ can be reconstructed:

$$\mathbf{A} = \mathbf{EDE}^{-1} \tag{4.53}$$

Furthermore, if our eigenvectors are also orthonormal i.e. $||\vec{e}_i|| = 1$, then $\mathbf{E}$ is an orthonormal matrix and easily invertible with its transpose:

$$\mathbf{A} = \mathbf{E}\mathbf{D}\mathbf{E}^T \tag{4.54}$$

From an algorithmic point of view, this means that matrix $\mathbf{A}$ can be constructed with a simple loop of additions, with no need to build $\mathbf{E}$:

$$\mathbf{A} = \sum_{i=0}^{n-1} \lambda_i \, \vec{e}_i \vec{e}_i^T \tag{4.55}$$

5. Implementation details

For this thesis, we have implemented an interactive simulator using the techniques, energies and approximations described in this document.

This simulator has been implemented with raw `C++`, with some libraries for linear algebra (GLM and Eigen), and OpenGL for the visualization. In this chapter we will only discuss some interesting or remarkable optimizations implemented in our solvers and simulators.

5.1 Intersection queries

As discussed in the section 4.1.4, we use ray intersection queries for collision detection.

These rays are slightly unique because their direction does not need to be normalized. Such alteration allows us to efficiently compute rays with a maximum span without requiring us to calculate distances or normalize vectors.

Ray queries are the only component of the simulator that uses floating-point numbers of single precision. This is to prevent issues when the rays are parallel to a surface, or its origin/end are too close, because less number resolution gives us larger margins to take these errors into account.

Furthermore, all ray queries also return the intersected primitive. This is used for more reliable distance tests, and to maintain or release collisions if required.

5.2 Eigen Sparse considerations

The Eigen [9] linear algebra library has its own set of routines specialized in working with sparse information. These are fine-tuned to get the maximum performance with an API that tries to replicate "on paper" formulation, which poses some limitations, obscures potential optimizations, and conceals other issues.

Ideally, we would use Sparse matrices with 3×3 block elements. However, these types of matrices are not supported by Eigen; thus, we have to use the standard Sparse Matrices and use them as best we can. Sparse matrices were described in section 4.2.

5.2.1 Sparse matrix block address indexing

Because in our simulation the mesh of tetrahedrons is not modified, and no extra forces are to be inserted between nodes, the system size is constant. We can store the memory addresses for each element into the sparse matrix for efficient $\mathcal{O}(1)$ random access.

At the beginning of the simulation, we preallocate a sparse matrix with entries in all 3×3 blocks of each pair of nodes that share an element and the blocks on the diagonal of the matrix. It is of utmost importance to build the matrix in compressed format, and building it from triplets has proven to be the most efficient.

Once the matrix is built, we can iterate through it and store for each block of 3×3 the pointers to the first elements of the 3 columns into a hash table (or a simple vector) indexed by the pair of node indices corresponding to the block. Because the sparse matrix is compressed, the following 2 elements of each stored pointer identify the other components down the column of the block.

This optimization was essential to get the $\mathcal{O}(1)$ random access to update the system at each iteration in real-time and set the sparse system to 0 without modifying its structure.

If we did not use this optimization and instead used Eigen's default random indexing and `setZero()` method to clean the matrix, we would have the following issues:

- Each random access to the matrix requires a binary search. So this is logarithmic access time even for consecutive values.
- The `setZero()` method cleans the whole structure of the matrix, so elements have to be inserted into the matrix again.
- Inserting elements into a matrix also makes the matrix uncompressed and will cause extremely expensive memory reallocations. Plus having to compress the matrix again.

5.2.2 Avoiding memory reallocations

Aside from the issue of filling a matrix, many other seemingly harmless statements with Eigen hide memory allocations and reallocations. In particular, most of the statements that use the assignment operator may cause the allocation of a new matrix before assigning it to the left-hand side.

This is because Eigen has the concept of "aliasing" when the same variable we are assigning into is being used on the right-hand side. Eigen cannot detect when to consider aliasing or not; thus, with some non-component-wise operations, it operates over a copy, then stores and deletes such copy.

There are some combinations of operations create better instructions than others because of Eigen's lazy evaluation, but this is not easy to assess because it can only be checked after compilation.

Avoiding such allocations is tricky and requires careful analysis and debugging of the statements. One can debug it by adding break-points in Eigen's `Core/util/Memory.h` allocations, and reorder the expressions until the desired behavior is reached.

The usage of `MatrixBase<Derived>::noalias()` is also recommended on matrices that are not used on the right-hand side of the assignment. However, sometimes this is not as fast as it should be, and explicit, allocated once, matrices may be better used.

5.2.3 Preconditioned Conjugate Gradient

Eigen's Conjugate gradient solver is impossible to improve algorithmically because the algorithm is so simple (Shown in section 4.3). However, applying the diagonal preconditioner can be improved, as Eigen's implementation does a linear search to find the diagonal entries of the sparse matrix.

Because we know that our system has all entries in its diagonal, we can optimize the preconditioner computation, making the computation of $\mathbf{P}^{-1}\vec{r}_{(i)}$ trivial with a simple loop.

Also, we can use our own implementation to reuse as much memory as possible in-between iterations of the Conjugate Gradient.

It is also interesting to note how important it is to start the Conjugate Gradient with a good initial guess. Eigen can also solve with an initial guess; however, it requires some memory copies and allocations.

5.3 Optimized PPCG

The application for constraints of our implementation for the Pre-filtered Preconditioned Conjugate Gradient (Section 4.1.2) can be heavily optimized; in particular, the computation for our final system $\mathbf{SAS}^T + \mathbf{I} - \mathbf{S}$.

Because the symmetric matrix $\mathbf{S}$ will only have entries in its 3×3 blocks in the diagonal, we could do something similar to our Block address cache (Section 5.2.1) and preallocate all entries of this diagonal with zeros. However, this would imply managing another sparse matrix, building it each step, and computing those products as many times.

We can do better by exploding the nature of the $\mathbf{SAS}^T$ operation, the shape of $\mathbf{S}$, and our storage of constraints. If we consider our elements of $s_{ij} \in \mathbf{S}$ and $a_{ij} \in \mathbf{A}$ to be blocks $s_{ij}, a_{ij} \in \mathcal{R}^{3\times 3}$ (or a tensor of 4th order in $\mathcal{R}^{n\times n\times 3\times 3}$) we can write our $\mathbf{SAS}^T$ operation as:

$$\left(\mathbf{SAS}^T\right)_{ij} = \sum_{k=0}^{n-1}\left(s_{ik}\sum_{l=0}^{n-1} a_{kl} s_{jl}^T\right) \qquad (\mathbf{S} \text{ is diagonal}) \qquad (5.1)$$

$$= s_{ii} a_{ij} s_{jj}^T \qquad (\mathbf{S} = \mathbf{S}^T) \qquad (5.2)$$

$$= s_{ii} a_{ij} s_{jj} \qquad (5.3)$$

This is a powerful optimization, as we do not need the matrix $\mathbf{S}$ at all, and only have to consider the equation 5.3 when node i-th or j-th are constrained. Otherwise, $\left(\mathbf{SAS}^T\right)_{ij} = a_{ij}$.

We can apply this optimization by traversing the matrix linearly, and for each matrix component check if it is under any constraint (using a hash table, for example, in $\mathcal{O}(1)$) and do the simple product. Of course, when iterating the diagonal we can also add $\mathbf{I} - \mathbf{S}$ to complete the PPCG system.

5.4 Parallelization

The construction of the sparse system can be easily parallelized. The computation of the energy contribution of each element is independent of the other elements; thus, we can parallelize the energies computation, together with the derivation of the forces, gradients and Hessians.

Because we have our static sparse matrix indexed, we can use atomic additions to add up the contribution of each element into the matrix without data races, and without triggering reallocations of the matrix.

Also, we can parallelize our computation for the Optimized PPCG of section 5.3. Given that the matrix $\mathbf{A}$ is sparse, we can parallelize the computation of the columns of the final PPCG system, because to compute $\left(\mathbf{SAS}^T\right)_{ij}$ we only need to access $\mathbf{A}_{ij}$.

Iterating a sparse matrix by columns is equivalent to doing a simple iteration over the non-zero values of the matrix, not requiring any complex behavior, which also is thread-safe.

5.5 CPU-GPU bandwidth

One of the problems of interactive applications is when it is not bounded algorithmically, but by memory bandwidth. No matter how much computing power we have, the program will not execute faster. Rendering can suffer from this issue.

GPUs are very fast in processing (and rendering) information already present in their VRAM; however, sending information from the CPU to the GPU can be slow. Because we are running our simulation on the CPU, we need to send the updated nodes to the GPU for rendering each frame.

Given that we only render the surface of the tetrahedron models, it is an essential consideration to only send to the GPU the vertices belonging to the surface of the models.

Also, given our dynamic time stepping (Section 4.4) we could have several updates per frame. It is important to only update the GPU memory once after several sub-steps to avoid saturating the bus.

5.6 Fast 3×3 SVD

In 2011 McAdams et al. [18] published an algorithm to compute the singular value decomposition of 3×3 matrices minimizing branching and floating-point operations.

This SVD relies on approximating the eigenanalysis of the squared input matrix with some Jacobi iterative method and then reconstructing from this the SVD. Of course, the number of Jacobi iterations determines the precision of the output; however, state that 4 iterations are usually enough for good convergence.

Such implementation is much faster and requires less memory than Eigen's SVD Jacobi solver, and numerous implementations exist on the web. We have used the implementation provided by Yuanming Hu at taichi [11].

6. Results

In this last chapter, we present the results obtained with the implemented simulator. We also discuss how the different energies behave in multiple scenarios and how their behavior may impact an interactive simulation.

6.1 Implementation Screenshots

In the figure 6.1 we show some images of the implemented simulator, representing and simulating different phenomena. In more detail:

- Subfigure 6.1a: Scene used for most benchmarks. Each sphere has almost 700 nodes and slightly over 1900 elements, and two of its vertices have been constrained.
- Subfigure 6.1b: Spheres rolling and bouncing on high friction surfaces, rebounding from one surface to the next.
- Subfigure 6.1c: Simple armadillo with several surface nodes constrained. The constrained nodes are interactive and can be translated and rotated around their centroid. The armadillo has 285 nodes and 787 elements.
- Subfigure 6.1d: Complex version of the armadillo, with an irregular surface, reacting to a collider. This armadillo has 961 nodes and 2880 elements.

The screenshots also show the interface of the program. Its benchmarking UI, lists of objects in the scene, simulation configuration, etc.

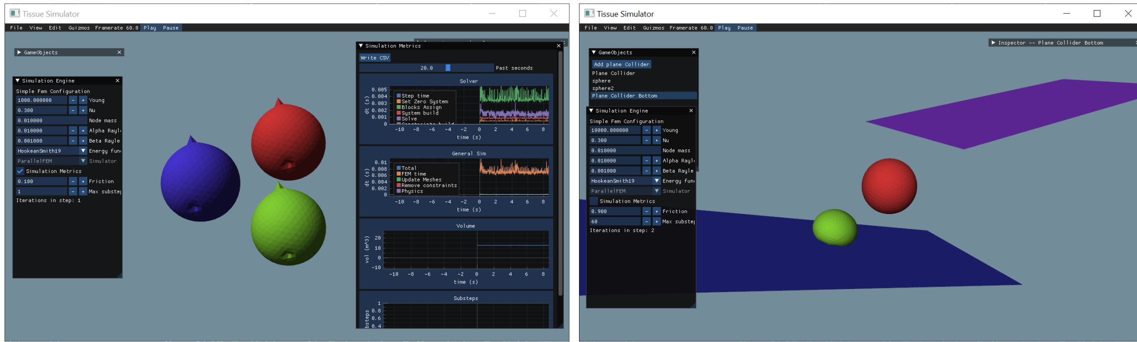
On the upper menu, one can also configure the engine's frame rate and threading configuration, load custom models into the scene, and load/store the scenes.

6.2 Optimizations

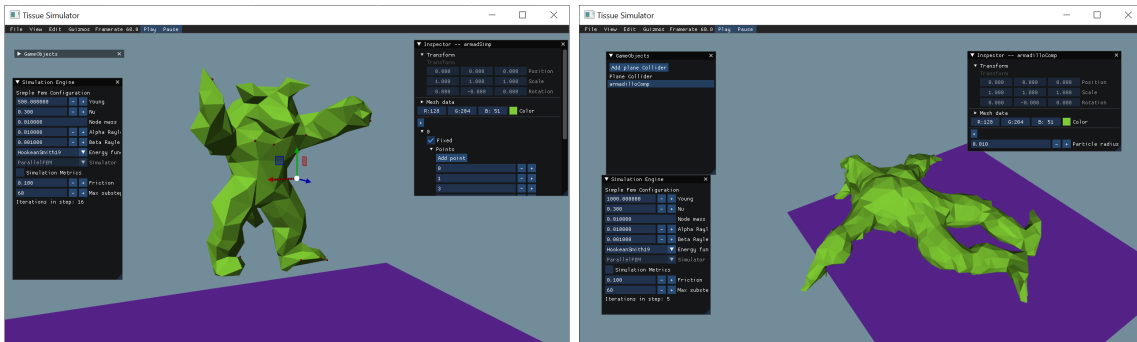
All optimizations presented in the chapter 5 have been benchmarked and are presented here.

In the first place, we need to dissect a complete simulation step into multiple smaller self-contained portions:

- *Blocks assign*: Computation of the energies for each element, adding Hessians contribution to $\frac{\partial \vec{f}}{\partial \vec{x}}$ and the forces $\vec{f}$ to the right-hand side of the system.



(a) Benchmarks (b) Friction and bounce



(c) Interaction (d) Complex response

Figure 6.1: Screenshots of different simulation scenes of varying complexity and phenomena. The models shown are provided with the software. Here is shown some of the UI; for example, on the left is the simulation configuration. On the image 6.1a you can see the benchmarking utility, that shows the evolution of a simulation and allows to export its contents to CSV. On the bottom images, you can see on the right the inspector, which permits to configure the transformation, mesh data, and add interaction groups to the game object.

- *System finish*: Complete the unconstrained system with position alteration of the equation 4.6. Apply Rayleigh damping and friction damping.

$$\left[\mathbf{M} - \Delta t \frac{\partial \vec{f}^{(t)}}{\partial \vec{v}} - \Delta t^2 \frac{\partial \vec{f}^{(t)}}{\partial \vec{x}}\right] \Delta \vec{v} = \Delta t\, \vec{f}^{(t)} + \Delta t^2 \frac{\partial \vec{f}^{(t)}}{\partial \vec{x}} \vec{v}^{(t)} + \Delta t \frac{\partial \vec{f}^{(t)}}{\partial \vec{x}} \vec{r} \quad (4.6)$$

- *Constraints*: Build the velocity constrained system of equation 4.2.

$$(\mathbf{SAS} + \mathbf{I} - \mathbf{S})\vec{y} = \mathbf{S}\vec{c} \quad (4.2)$$

- *Solve*: Solve the previous system with Conjugate Gradients.
- *Update Mesh*: Send the new surface vertices positions to the GPU's VRAM.
- *Remove constraints*: Check the system's constraints, and update/remove them if necessary.
- *Physics*: Check intersections and collisions for each surface node with the environment.
- *Step*: Total step time, considering all previous items, and some extra negligible safety checks.

All benchmarks have been done in a desktop computer with the following specifications:

- AMD Ryzen 5 2600, 6 Cores (12 hyperthreading)
- NVIDIA GeForce GTX 1060 6GB
- 16GB RAM

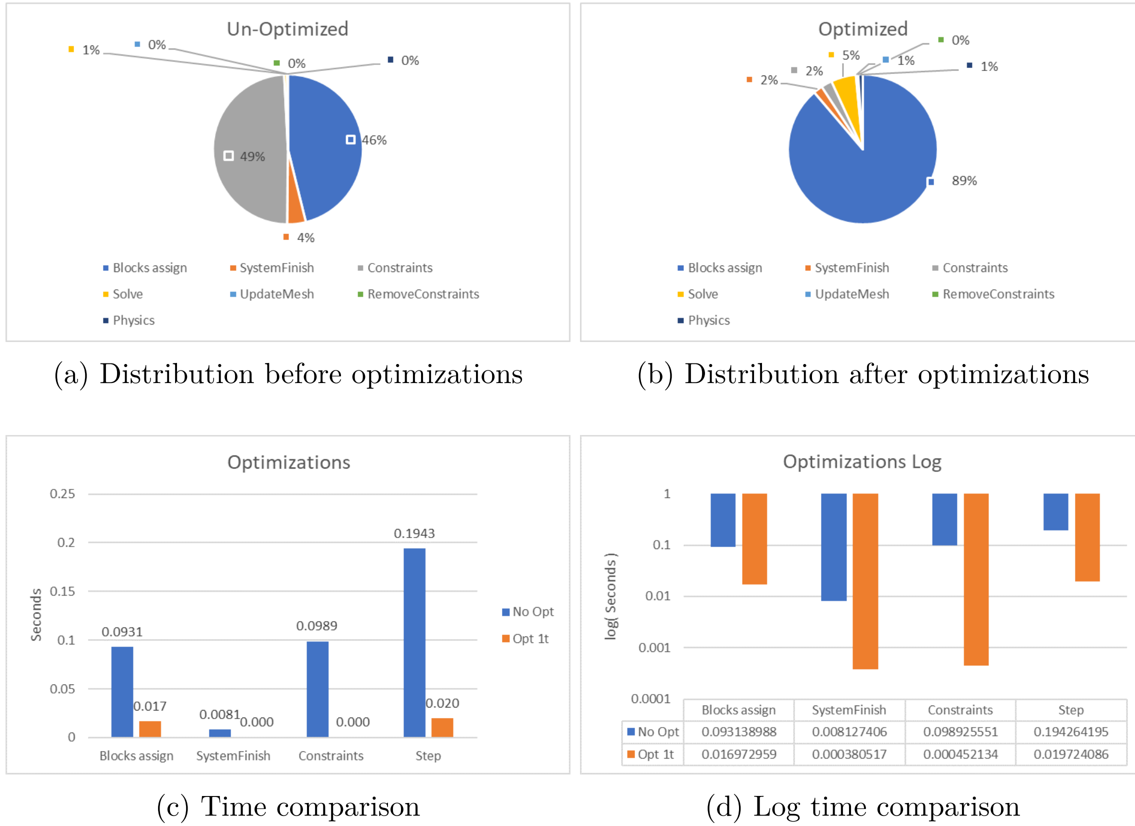
With such a machine, all tests easily fit in memory and are not constrained by the rendering.

The benchmark scene consisted of running at least 230 frames and averaging the results, with $\Delta t = \frac{1}{100}$, of the scene in the figure 6.1a, with almost 6000 elements and 2100 nodes. No dynamic substeps were used.

6.2.1 General analysis

Before the optimizations, as the figure 6.2a shows, the application of constraints was the most expensive step, followed by the computation of energies and assignation to the matrix. These were our two focuses for optimization.

The cost of the *Constraints* step is due to how expensive it is to build the sparse matrix $\mathbf{S}$ in every step, even if it is very similar to the identity matrix. Even avoiding reallocations, compressing the matrix for efficient products is too expensive.



(a) Distribution before optimizations (b) Distribution after optimizations

(c) Time comparison (d) Log time comparison

Figure 6.2: Charts for comparison of optimization implementations, using only 1 thread. Used the benchmark scene explained at the beginning of the section 6.1. In the two bottom charts, lower is better.

Blocks assign	SystemFinish	Constraints	Step
x5.487	x21.358	x218.79	x9.849

Table 6.1: Speedups table of the applied optimizations, using only 1 thread. *Step* represents the overall pipeline.

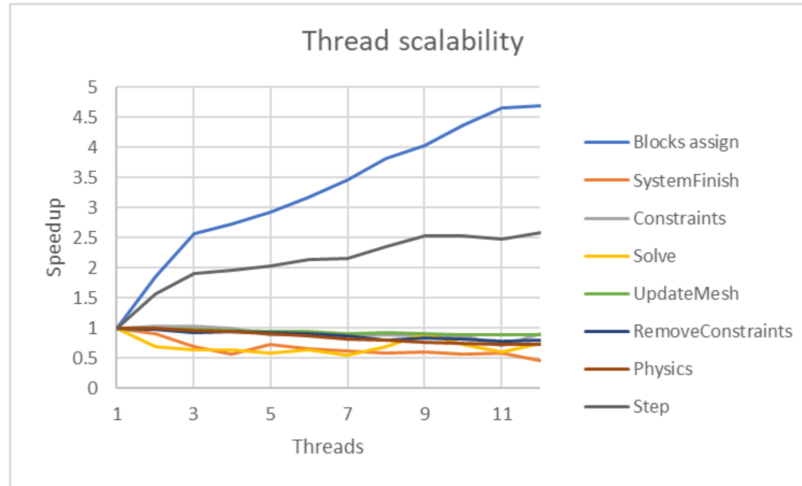


Figure 6.3: Threading speedups of the multiple steps of the pipeline. Using the benchmark scene explained at the beginning of the section 6.1. *Step* represents the overall pipeline.

As you can see in the figures 6.2c and 6.2d, avoiding computing the matrix **S** reduces the impact of this step enormously. This is a solution that can escalate to any scene. In the case of the benchmark scene, this was reflected in a speedup of x218.79 (Table 6.1).

In the *Blocks assign* step, we added our static indexed sparse matrix by blocks (Section 5.2.1), which showed in a speedup of x5.487 by just avoiding the binary searches in the sparse matrix. The cost decreased from 0.093 seconds to 0.017 seconds.

We modified the formulations for the system to avoid extra allocations in the *SystemFinish* step, which is reflected in a speedup of x21.35, reducing the cost from 0.008 seconds to 0.00038 seconds. This optimization shows how important it is to analyze how your algebra library operates with your variables.

All these optimizations effectively reduced the overall step cost by x9.85, from 0.194 seconds to 0.0197 seconds.

However, even with these optimizations, we still have some margin for improvement. As the new cost distribution expresses, as shown in the figure 6.2b, the *Blocks assign* are our new bottleneck taking up 89% of the step time. Parallelization serves to mitigate such issues.

6.2.2 Parallelization

Introducing parallelization is necessary to use all the available resources of our processors. We have added parallelization with *OpenMP* in multiple portions of our code.

In the case of the bottleneck step *Blocks assign*, we have used atomic operations to update the static indexed sparse matrix.

	Eigen	Eigen with guess	Custom
Time	0.0071 s	0.00439 s	0.00168 s
Speedup	x1	x1.614	x4.226

Table 6.2: Speedups of Conjugate Gradients implementations.

The figure 6.3 shows the different speedups of the different portions of the algorithm. Notice how *Blocks assign* greatly benefits from our parallelization, carrying on the speedup to the overall step.

However, notice that the speedup is not 1:1 with the number of threads. This is due to the massive amount of atomic additions and the memory penalization induced by random access.

Moreover, all the other pipeline parts do not seem to benefit from the parallelization. Even if in smaller scenes we have data showing some speedups, with this benchmark scene it does not show. This may be because non-parallelizable operations overshadow the parallelized ones, adding to the penalization of opening and handling parallel regions.

6.2.3 Preconditioned Conjugate Gradient

As explained in the section 5.2.3, we can improve Eigen's implementation of the conjugate gradient by using a more intelligent sparse matrix access, avoiding dynamic memory allocations, and applying the preconditioner directly. Thus, we wanted to see how using the previous step solution as a guess affected our solve time.

In table 6.2 you can see the costs and speedups of using Eigen's `solve` and `solveWithGuess` methods of the iterative solver, and our implementation of the Conjugate Gradient applying all optimizations mentioned above.

Even if using a good guess for the first iteration of the Conjugate Gradients brings a speedup of at least x1.614, avoiding dynamic allocations and unnecessary copies seems to be the decisive factor. This shift in importance is explained by the fast convergence of the Conjugate Gradients, and the cost befalling on all other operations of the solve step.

6.3 Stability

In an interactive simulation we need to guarantee stability as much as possible, because if the simulation grows unstable we do not have a fallback system to go back on track.

Stability depends on many different factors, like the delta time used, the implementation, and the models used. Also, stability can be easily measured by computing the volume of the deformed bodies over time. Here we discuss some of the behaviors these factors produce.

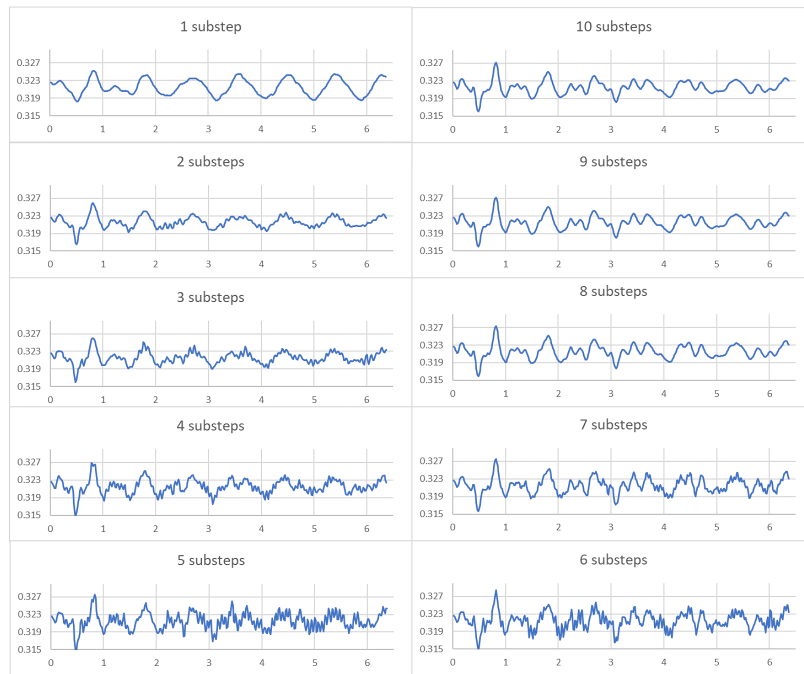


Figure 6.4: Volume over time of a mostly static simulation, under a different number of substeps at 60 FPS. Notice how vibrations increase and decrease along with the number of substeps. Observe that the *10 substeps* chart seems to have stabilized, and has more detail than the *1 substep*.

6.3.1 Dynamic time-stepping

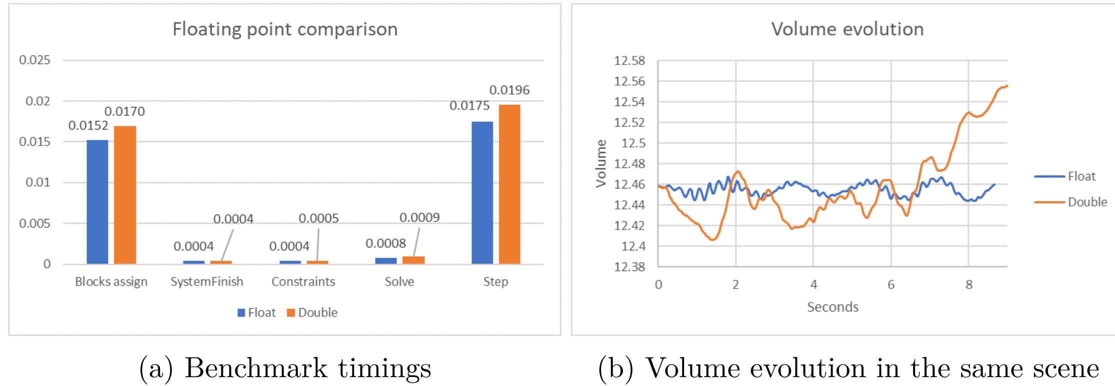
Dynamic time-stepping increases the frequency of the simulation steps in a second significantly. Smaller time steps always have a positive impact on stability.

In environments with an FPS upper bound, this dynamic-time stepping lets us use as much of the frame time for the simulation. For example, if we are running at 60 FPS, we have a budget of 16ms to use. According to our optimized times using 10 threads, one single step takes almost 8ms. If we only run one step at each frame, we have a 50% utilization of the frame. However, using 2 substeps, we have a 100% utilization.

Of course, having such time-stepping be dynamic helps only the framerate. When large deformations happen, the system's condition number can grow, and the solver may take longer. In these cases, we would like smaller time steps to ensure that the system can recover; however, because the solving is slower, we assign larger and fewer time steps in a frame, which may lead to instabilities.

Also, we have identified one strange phenomenon: vibrations. Using smaller sub-steps causes the model to vibrate, and the vibration augments and diminishes in a span of substeps. This is depicted in the figure 6.4.

This interesting fact also happens with higher frame rates. It seems due to some inherent issue with our *Backwards Euler* formulation, where the error produced behaves in some harmonic fashion. This may have something to do with the deformation modes.



(a) Benchmark timings

(b) Volume evolution in the same scene

Figure 6.5: Comparison of single and double precision in the simulator. Speedup of x1.12 using single precision over double. However, they show very different behaviors, where using floats causes stiffness.

Thus, it is important to fine-tune the maximum or minimum simulation substeps if the vibrations start getting noticed. However, they do not seem to substantially impact the stability, as the vibrations are small and do not lead to explosions.

6.3.2 Floating-point precision

Using single and double floating-point precision directly impacts stability and performance.

In the figure 6.5a you can see how using floats is a little above a 10% faster than double precision. This speedown is not only due to having to deal with more resolution in the reals, but in having to allocate and manage double the memory, and because we need to have random access in the sparse matrices, the cost of the access increases also with memory because of the cache size.

However, the floating-point precision's most significant impact is in the simulated bodies' behavior. Having less resolution in our reals causes the simulation to become stiffer and do not deform nearly as much as if we were using double precision, which can be seen in the figure 6.5b.

A simulation using floats will preserve the object's shape much better and be closer to the real/original shape. But if we want to simulate large deformations, it is mandatory to use double-precision, as the floats are unable to provide enough resolution to encode the small forces and gradients that overall cause the exciting phenomena.

6.3.3 Degenerate tetrahedron meshes

Correctness on the input tetrahedralization of the bodies to simulate is crucial for a feasible simulation.

We define a correct tetrahedralization where all tetrahedrons are as regular as possible. Using the definition of a Delaunay triangulation, we would like to have a tetrahedralization where its minimum corner-angles have been maximized.

A tetrahedrons model with very elongated and thin tetrahedrons is prone to grow unstable in just a few iterations unless tiny time steps are used. Thus, it is necessary to use correct tetrahedron models which have been processed to attain such good tetrahedrons.

One must note that, given a surface mesh with n vertices, its straightforward tetrahedralization with n nodes will most likely have degenerate tetrahedrons. We have used *TetGen* [26] to add *Steiner points* and create better meshes for simulation with more than n nodes.

6.4 Energy models

We have implemented the 3 different energy models explained in the section 2.4, and another using the eigenanalysis of the Stable Neo-Hookean energy shown in the section 4.5.1.

All these energies present many different behaviors, as they produce different gradients and Hessians, which require different computations.

6.4.1 Performance

To compare the different energies, we have set up a scene that deals with both an undeformed and deformed body, consisting of a sphere that falls and bounces multiple times against a flat surface.

In the figure 6.6 you can see the results for the 4 different energies. Notice that in all simulations the first bounce occurs at second 1.

All 4 charts have in common that the most critical substeps are the *Blocks assign*, *Solve*, and *Constraints*. The other substeps, even the *SystemFinish*, remain constant throughout the simulation.

It is easy to see that the *Neo Hookean* energy by Smith et al. is not only the fastest to compute, but the most stable. Consecutive iterations have similar costs, making this energy very suitable for the dynamic time-stepping feature because we can trust that the predicted cost of the next frame will be similar to the current one.

As for the *Corrotational*, it is slightly more expensive than the previous one. However, notice that the solve step is the cheapest among all energies. This is due to being a "relaxed" energy, as it does not take volume into account, the forces introduced seem to be softer, which leads to more stable systems.

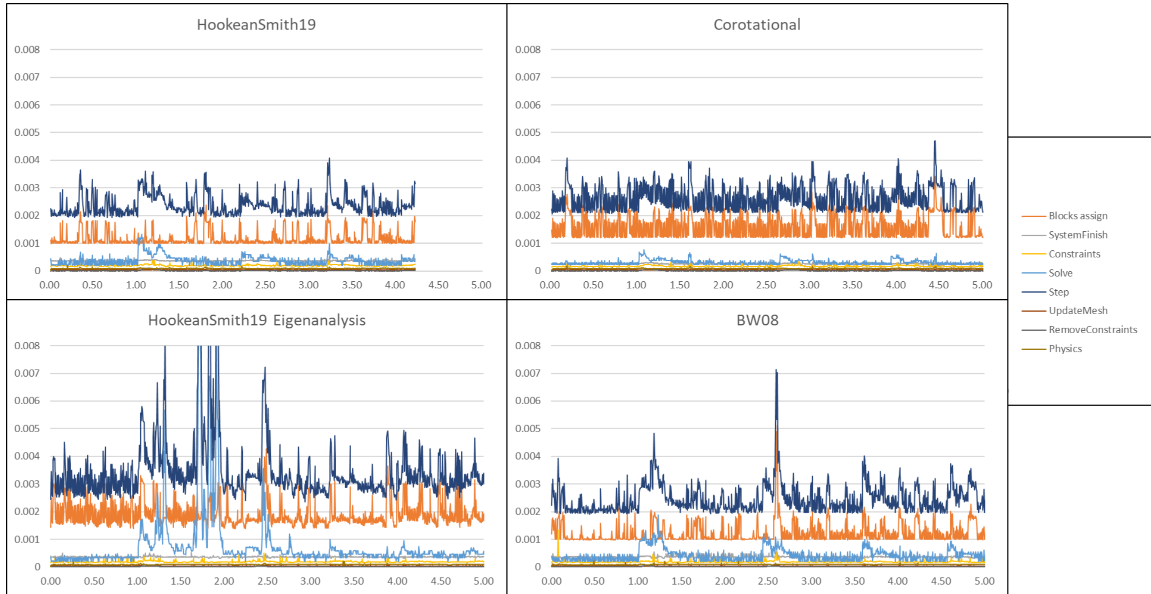


Figure 6.6: Performance comparison of the different energies. The scene consists of a ball suspended in the air that bounces against a flat surface, with a fixed frame rate of 100Hz.

The noticeable fluctuations in the *Corotational* and *Eigenanalysis* energies are not due to the energy per-se, but to the computation of the fast Singular Value Decomposition of McAdams (Section 5.6), which may converge faster or slower depending on the input matrix. It is a necessary cost to pay for up to a x2 speedup in the SVD computation. Eigen's SVD does not produce such fluctuations, though.

The *Eigenanalysis* implementation of the *HookeanSmith19* energy, where we compute the eigenmatrices and eigenvalues, snap those back to the positive domain, and reconstruct the Hessian, has serious issues. Even if it can maintain the simulation, it is not stable nor fast to compute. Trying to avoid producing not positive-definite matrices ends up causing even more unstable systems, which defeats the purpose of this approach.

Thus, fixing the Hessian through the eigenanalysis does not work for our case. We suppose that in attempting to fix a few of the eigenvectors, the condition number of the system augments considerably, forcing the solver to take many more iterations. Either this, or we would need to reconstruct the Hessian with a different, more delicate approach.

Finally, our old *Neo-Hookean* by Bonet and Wood still is able to compete with the other, more new, energies. However, it does not react well to external forces or interactions. An inherent problem of this energy causes these huge spikes, and that is that it uses $\log(\det \mathbf{F})$: when the sphere bounces against the surface, some elements may compress a lot, and $\det(\mathbf{F}) \to 0$, which causes the logarithm to grow and add too much energy to the system. Also, because of this logarithm, this energy will explode if an inversion of an element happens.

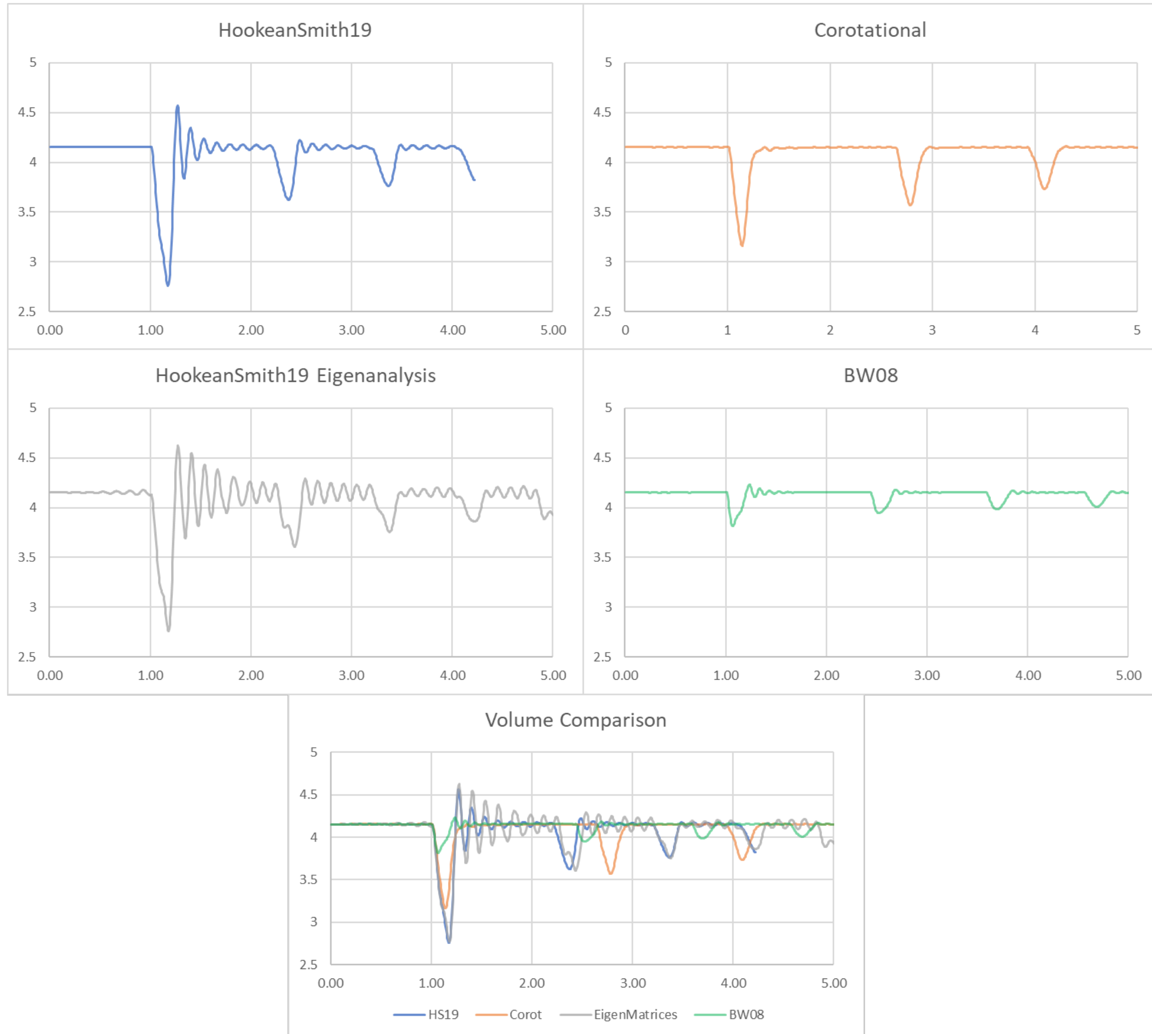


Figure 6.7: Volume comparison of the different energies along a simulation consisting of a bounce. All different energies correctly preserve the volume along time (no volume disappears); however, they have different behaviors. In the bottom chart you see the 4 energies' charts overlapped to compare the different oscillations better.

6.4.2 Stability and behavior

As we did in the section 6.3, one of the ways to consider stability and analyze the behavior of an energy is through the evolution of the simulated bodies' volumes. In the figure 6.7 we see the change in the volume for the same simulation of our performance analysis and figure 6.6.

The first thing that comes to our attention is that the Bonet and Wood *Neo-Hookean* simulates the smallest of deformations, being overly conservative. No huge compressions happen, and it balances the strain along with the deformation very fast. Notice also that the deformation disappears quickly after a deformation, and the body recovers its original shape in the air.

On the other hand, the *Neo-Hookean* energies derived from Smith et al. have the complete opposite behavior. Large compressions happen that take a while to become stabilized. While the compression tries to stabilize, some wiggling happens, and the perpendicular direction of the compression becomes elongated, and vice versa. This wiggling produces a very particular phenomenon.

In particular, its *Eigenanalysis* version propagates this deformation oscillations too much, making the material buckle from time to time, and explode. Notice also that before the impact at second 1, some oscillations have started building up. This is a prompt to see that this energy generates deformations out of nothing, which can be terrible in the long run.

Furthermore, the *Eigenanalysis* energy is terrible when an inversion or a huge compression occurs. These are the huge spikes in the figure 6.7. This energy also exploded all the time on our interactive tests, even without any external input. Very tiny time steps were necessary even to maintain stability.

The *Corotational* energy is an in-between. Because it only takes into account the elongation of the elements, and not its volume, it clearly behaves very spring-like. On a collision, it just compresses the material (with a considerable deformation) and decompresses it returning immediately to its rest shape. Notice also that because of this, all the energy of the deformation goes back only to the expansion, and thus the jump is more extensive, and the sphere is in the air for a longer time.

6.5 Conclusions

In this thesis, we presented a comprehensive guide to simulation for animation using the Finite Elements Method with tensors, discussed several optimizations and simplifications to use them in interactive environments, and compared different energies in such environments. Furthermore, a simulator has been implemented with everything discussed in the thesis.

In the first place, we show that it is plausible to use production-level techniques in real-time simulation, being able to simulate more than 5000 elements at 100Hz.

Next, we presented several optimizations used, their pros and cons, to achieve such performance. We did an in-depth analysis of the Eigen `C++` library with the objective to have the most performant implementation possible with such a library, analyzing what was worth coding from scratch, and what was worth using from the library.

With the data available in Kim and Eberle [15], our performance per element per second is superior than Pixar's Fizt2. Of course, take this statement with a pinch of salt, as Fizt2 is much more complex than what we did in this thesis. However, taking only into account the times of the systems assembly, our simulation performs just as fast.

For interactive environments with fixed maximum frames per second (for example, with forced V-sync), we presented a dynamic time-stepping solution that is able to use as much as possible of the frame time with simulation steps. However, care must be taken with this feature, as unstable energies with spikes of slowdowns can have an even more significant impact.

Finally, we discuss some exciting energy models used in our system. The *Neo-Hookean* by Smith et al. and a classical *Corotational* proved to be highly reliable; however not equivalent as they produce very different behaviors. On the other side, an analytic eigenanalysis to fix Hessians did not successfully produce good enough results.

Thus, this thesis successfully achieved most of its initial objectives, having analyzed the inner workings of current production-level simulators, implementing such techniques, and showing its capabilities.

6.6 Future work

We did not explore all the intended initial features and ideas for this thesis due to the lack of time. Here we present some of these ideas, and others that arose during the development of the thesis.

- Full Collision system: The current implementation is not accelerated with spatial data structures and only supports planes as colliders. A complete collision system with many collider primitives, even triangle meshes, accelerated with a Bounding Volumes Hierarchy would be a great addition to test the collision impact in the simulation at its fullest.
- Self-Collisions: Collisions of a body with itself are very complex and require detecting the self-collision, and applying either spring forces, or solving some minimization problem to correct them. Many algorithms (both global and local) exist, and discussing them for real-time uses would produce a fascinating discussion.
- Intel oneMKL: Intel provides the *Math Kernel Library*, which includes optimized math routines. Using them would bring a new spectrum of optimizations for the linear algebra of the solver. Also, Eigen is compatible with oneMKL, so it would be interesting to analyze how it is used.
- Skin tissue with bones: Attaching a model representing muscle or skin tissue to a set of bones and joints, animated using some kinematics, would be used to make the skin react to the movements, showing the contraction of muscles on the different joints. We could use it to animate a human face, where the skin reacts automatically to different expressions.

- 3D Triangle elements: We have only used tetrahedrons as elements. It is tricky to use triangles in 3 dimensions because their parameterization is 2D, and the correspondent deformation gradient is $\mathcal{R}^{2\times 2}$. To build a 3D deformation gradient we need to apply some other approaches; that would allow us, for example, to simulate triangulated cloth.

- Materials: The current simulation uses a single global material, and assumes all nodes have the same weight. Being able to configure all these parameters per simulated object would allow us to have a new myriad of different simulations.

- Smooth mouse interaction: Current interaction requires selecting groups of vertices. Applying local deformations, or grabbing portions of the simulated body, with a click of the mouse and affecting a region around it would be a great addition to the simulator.

- Analyze vibrations with small steps: As shown in the figure 6.4, using smaller delta-times causes some type of harmonic vibrations. An in-depth analysis of this phenomenon would be very revelating.

Appendix A. Linear algebra

A.1 Frobenius norm

The Frobenius norm, identified with the operator $||\cdot||_F$, evaluates the matrix similar to the 2-norm of a vector.

For a matrix $\mathbf{M} \in \mathcal{R}^{n\times m}$:

$$||\mathbf{M}||_F = \sqrt{\sum_i^n \sum_j^m m_{ij}^2} \tag{A.1}$$

It is common to use the squared Frobenius norm, which gets rid of the square root.

$$||\mathbf{M}||_F^2 = \sum_i^n \sum_j^m m_{ij}^2 \tag{A.2}$$

A.2 Tensor flattening

Flattening of tensors following the notation of Golub and Van Loan [8], with the vec($\cdot$) operator.

For order 2 tensors, $\text{vec}(\cdot) : \mathcal{R}^{n\times m} \rightarrow \mathcal{R}^{nm}$:

$$\text{vec}(\mathbf{M}) = \begin{pmatrix} m_{11} \\ m_{21} \\ \vdots \\ m_{nm} \end{pmatrix} \tag{A.3}$$

For a second-order tensor of 2×2.

$$\mathbf{M} = \begin{bmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \end{bmatrix} \longleftrightarrow \text{vec}(\mathbf{M}) = \begin{pmatrix} m_{11} \\ m_{12} \\ m_{21} \\ m_{22} \end{pmatrix} \tag{A.4}$$

For order 4 tensors it is slightly more complex.$\text{vec}(\cdot) : \mathcal{R}^{n\times m\times k\times l} \rightarrow \mathcal{R}^{kl\times nm}$. The best way to understand this operation is with an example. Given the order 4 tensor $\mathcal{A}$, we arrange it as a matrix of matrices:

$$\mathcal{A} = \begin{bmatrix} \begin{bmatrix} a & c \\ b & d \end{bmatrix} & \begin{bmatrix} i & k \\ j & l \end{bmatrix} \\ \begin{bmatrix} e & g \\ f & h \end{bmatrix} & \begin{bmatrix} m & o \\ n & p \end{bmatrix} \end{bmatrix} = \begin{bmatrix} [\mathbf{A_{11}}] & [\mathbf{A_{12}}] \\ [\mathbf{A_{21}}] & [\mathbf{A_{22}}] \end{bmatrix} \tag{A.5}$$

Then, the vectorization of such tensor is a double unfolding that returns the following matrix:

$$\text{vec}(\mathcal{A}) = [\text{vec}(\mathbf{A_{11}})|\text{vec}(\mathbf{A_{21}})|\text{vec}(\mathbf{A_{12}})|\text{vec}(\mathbf{A_{22}})] \tag{A.6}$$

$$= \begin{bmatrix} a & e & i & m \\ b & f & j & n \\ c & g & k & o \\ d & h & l & p \end{bmatrix} \tag{A.7}$$

The vectorization of a third rank tensor is equivalent.

A.3 Tensor double contraction

The double contraction operator for tensors ":" is a generalization of the dot product for tensors, where the result is a tensor of less order than the operands.

For 2-dimensional matrices, we can compute $\mathbf{A} : \mathbf{B}$ which will result in a scalar value. To do this, $\mathbf{A}$ and $\mathbf{B}$ need to have precisely the same size. It is a multiplication component by component, followed by an addition.

$$\mathbf{A} : \mathbf{B} = \sum_i \sum_j a_{ij} b_{ij} \tag{A.8}$$

Example in the case of 2×2 matrices:

$$\mathbf{A} : \mathbf{B} = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} : \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} = a_{11}b_{11} + a_{12}b_{12} + a_{21}b_{21} + a_{22}b_{22} \tag{A.9}$$

This operator can scale up with higher-order tensors, where only the most inner 2 dimensions must be equal. It behaves equally, but reduces in 2 the order of the highest order tensor.

For example, applying this operation to a 4th order tensor $\mathcal{A}$ with a 2nd order tensor $\mathbf{B}$ leads to the following (again, defining $\mathcal{A}$ as a matrix of matrices):

$$\mathcal{A} : \mathbf{B} = \begin{bmatrix} [\mathbf{A_{11}}] & [\mathbf{A_{12}}] \\ [\mathbf{A_{21}}] & [\mathbf{A_{22}}] \end{bmatrix} : \mathbf{B} \tag{A.10}$$

$$= \begin{bmatrix} [\mathbf{A_{11}}] : \mathbf{B} & [\mathbf{A_{12}}] : \mathbf{B} \\ [\mathbf{A_{21}}] : \mathbf{B} & [\mathbf{A_{22}}] : \mathbf{B} \end{bmatrix} \tag{A.11}$$

The same operation, but for 3rd order tensors, is the same.

$$\mathcal{A} : \mathbf{B} = \begin{bmatrix} [\mathbf{A_1}] \\ [\mathbf{A_2}] \end{bmatrix} : \mathbf{B} \tag{A.12}$$

$$= \begin{bmatrix} [\mathbf{A_1}] : \mathbf{B} \\ [\mathbf{A_2}] : \mathbf{B} \end{bmatrix} \tag{A.13}$$

This very same double contraction can be computed with vectorization, and simple matrix-vector products, i.e. $\mathcal{A} : \mathbf{B} = \text{vec}(\mathcal{A})^T \text{vec}(\mathbf{B})$.

A.4 Singular Value Decomposition

The Singular Value Decomposition (SVD) of a matrix $\mathbf{A}$ of real numbers is a product of 3 matrices:

$$\mathbf{A} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T \tag{A.14}$$

From these matrices, $\mathbf{\Sigma}$ is a diagonal matrix with the singular values, which express elongation across some orthonormal directions.

Both $\mathbf{U}$ and $\mathbf{V}^T$ are orthonormal matrices, only encoding rotations. Because of this, the SVD is not unique, as there may be multiple rotations that end up with an equivalent factorization.

It is important to note that in this thesis we assume that all inversions are encoded in the singular values, i.e. $\det(\mathbf{U}) = \det(\mathbf{V}^T) = 1$. If you are using an SVD that allows this kind of inversion, you can detect if $\mathbf{U}$ or $\mathbf{V}$ encode an inversion and apply the inversion to the matrix on one dimension and its singular value.

A.5 Polar Decomposition

The Polar Decomposition of a matrix $\mathbf{A}$ is a product of two matrices:

$$\mathbf{A} = \mathbf{R}\mathbf{S} \tag{A.15}$$

Where $\mathbf{R}$ is an orthonormal matrix encoding a rotation, and $\mathbf{S}$ a symmetric matrix with the scaling or shearing applied to the matrix.

It is important to note that the Polar Decomposition can be computed from the SVD.

$$\mathbf{R} = \mathbf{U}\mathbf{V}^T \tag{A.16}$$

$$\mathbf{S} = \mathbf{V}\mathbf{\Sigma}\mathbf{V}^T \tag{A.17}$$

A.6 Divergence of a vector field

The operator $\nabla\cdot$ indicates the computation of the divergence of a vector field. This concept defines the tendency of the vectors to "flow" inside or outside a point of interest.

The divergence maps a vector field to a scalar field, representing the magnitude of the flow in the vector field. It is defined for a 3D vector field as:

$$\text{div}(\vec{\Phi}) = \nabla \cdot \vec{\Phi} \equiv \frac{\partial \Phi_x}{\partial x} + \frac{\partial \Phi_y}{\partial y} + \frac{\partial \Phi_z}{\partial z} \tag{A.18}$$

Where $\vec{\Phi}$ is an arbitrary vector field.

If the vector field is a function of multiple variables, say $\vec{\Phi}(\vec{x}, t)$ a function of a position and time, we can specify that we are computing the divergence with respect to one of the variables with a subindex:

$$\nabla_{\vec{x}} \cdot \vec{\Phi}(\vec{x}, t) \equiv \frac{\partial \Phi_x}{\partial x}(\vec{x}, t) + \frac{\partial \Phi_y}{\partial y}(\vec{x}, t) + \frac{\partial \Phi_z}{\partial z}(\vec{x}, t) \tag{A.19}$$

A.7 Tetrahedron volume

There are multiple ways to compute the volume of a tetrahedron. We define a tetrahedron with vertices $\vec{a}$, $\vec{b}$, $\vec{c}$ and $\vec{d}$.

Our approach here will make use one of the properties of the determinant of a matrix: the determinant of a square matrix $\mathbf{R}^{n \times n}$ gives the signed n-dimensional volume of a parallelepiped, with its first coordinate in the origin.

Because we want the volume of a tetrahedron, we divide the volume of a parallelepiped by 6, as this is the number of tetrahedrons needed to subdivide a parallelepiped.

$$V = \frac{1}{6} \left| \det \left[\begin{array}{c|c|c} \vec{a} - \vec{d} & \vec{b} - \vec{d} & \vec{c} - \vec{d} \end{array} \right] \right| \tag{A.20}$$

Appendix B. Derivations

B.1 S-centric invariants

Given the 3 S-centric invariants of Smith et al. [29].

$$I_1 = \mathrm{tr}(\mathbf{S}) \tag{B.1}$$
$$I_2 = \mathrm{tr}(\mathbf{F}^T\mathbf{F}) \tag{B.2}$$
$$I_3 = \det(\mathbf{F}) \tag{B.3}$$

We will derive the 3 invariants in the following sections:

B.1.1 Invariant I_1

The first derivative of I_1 with respect to the deformation gradient is straightforward:

$$\frac{\partial I_1}{\partial \mathbf{F}} = \frac{\partial \mathrm{tr}(\mathbf{S})}{\partial \mathbf{F}} \tag{B.4}$$
$$= \frac{\partial \mathrm{tr}(\mathbf{R}\mathbf{F}^T)}{\partial \mathbf{F}} \tag{B.5}$$
$$= \mathbf{R} \tag{B.6}$$

The Hessian is much more complex, as $\frac{\partial^2 I_1}{\partial \mathbf{F}^2} = \frac{\partial \mathbf{R}}{\partial \mathbf{F}}$. For some years, this derivative has been a problem, as numerical methods were used to approximate it. However, Smith et al. [29] provide an analytic expression for it.

The analytic expression is derived from the eigendecomposition of $\frac{\partial \mathbf{R}}{\partial \mathbf{F}}$, and its eigenmatrices. This derivation is out of the scope of this thesis, and we redirect to the original paper for a very thorough derivation. Here are the terms needed, given the singular-value-decomposition of $\mathbf{F} = \mathbf{U\Sigma V^T}$, where $\Sigma = (\sigma_x, \sigma_y, \sigma_z)$.

$$\lambda_0 = \frac{2}{\sigma_x + \sigma_y} \tag{B.7}$$
$$\lambda_1 = \frac{2}{\sigma_y + \sigma_z} \tag{B.8}$$
$$\lambda_2 = \frac{2}{\sigma_x + \sigma_z} \tag{B.9}$$

$$\mathbf{Q}_0 = \frac{1}{\sqrt{2}}\mathbf{U}\begin{pmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}\mathbf{V}^T \tag{B.10}$$
$$\mathbf{Q}_1 = \frac{1}{\sqrt{2}}\mathbf{U}\begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & -1 & 0 \end{pmatrix}\mathbf{V}^T \tag{B.11}$$
$$\mathbf{Q}_2 = \frac{1}{\sqrt{2}}\mathbf{U}\begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \\ -1 & 0 & 0 \end{pmatrix}\mathbf{V}^T \tag{B.12}$$

The final Hessian is:

$$\mathrm{vec}(\frac{\partial I_1^2}{\partial \mathbf{F}^2}) = \sum_{i=0}^{2} \lambda_i \mathrm{vec}(\mathbf{Q_i})\mathrm{vec}(\mathbf{Q_i})^T \tag{B.13}$$

B.1.2 Invariant I_2

The derivatives of I_2 with respect to the deformation gradient are both simple:

$$\frac{\partial I_2}{\partial \mathbf{F}} = \frac{\partial \mathrm{tr}(\mathbf{F}\mathbf{F}^T)}{\partial \mathbf{F}} \tag{B.14}$$

$$= 2\mathbf{F} \tag{B.15}$$

And the Hessian:

$$\mathrm{vec}(\frac{\partial I_2^2}{\partial \mathbf{F}^2}) = \mathrm{vec}(\frac{\partial 2\mathbf{F}}{\partial \mathbf{F}}) \tag{B.16}$$

$$= 2\mathrm{vec}(\begin{pmatrix} \mathbf{I}_{3\times3} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{I}_{3\times3} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{I}_{3\times3} \end{pmatrix}) \tag{B.17}$$

$$= 2\mathbf{I}_{9\times9} \tag{B.18}$$

B.1.3 Invariant I_3 and $\frac{\partial J}{\partial \mathbf{F}}$

The gradient of J can be derived from Jacobi's formula of the derivative of the determinant of a matrix, i.e. $\frac{|\partial \mathbf{M}|}{\partial m_{ij}} = \mathrm{adj}(\mathbf{M})_{ji}$.

In another words, if $\mathbf{F} = \left(\vec{f_0} \,\middle|\, \vec{f_1} \,\middle|\, \vec{f_2} \right)$, $\vec{f_i}$ the columns of $\mathbf{F}$:

$$\frac{\partial J}{\partial \mathbf{F}} = \left(\vec{f_1} \times \vec{f_2} \,\middle|\, \vec{f_2} \times \vec{f_0} \,\middle|\, \vec{f_0} \times \vec{f_1} \right) \tag{B.19}$$

The Hessian of J is a little more complex and requires quite a bit of symbolic analysis. However, it can be built from the cross-product matrices of the columns of $\mathbf{F}$.

A cross product matrix of a vector $\vec{a}$ is a matrix $\overset{\times}{\mathbf{a}}$ where given any vector $\vec{b}$, then $\overset{\times}{\mathbf{a}}\vec{b} = \vec{a} \times \vec{b}$.

The final Hessian of J is the following flattened tensor:

$$\mathrm{vec}(\frac{\partial I_3^2}{\partial \mathbf{F}^2}) = \begin{pmatrix} \mathbf{0} & -\overset{\times}{\mathbf{f}}_2 & \overset{\times}{\mathbf{f}}_1 \\ \overset{\times}{\mathbf{f}}_2 & \mathbf{0} & -\overset{\times}{\mathbf{f}}_0 \\ -\overset{\times}{\mathbf{f}}_1 & \hat{\mathbf{f}}_0 & \mathbf{0} \end{pmatrix} \tag{B.20}$$

B.1.4 Summary

As a summary here are the invariants:

$$I_1 = \text{tr}(\mathbf{S}) \tag{B.21}$$
$$I_2 = \text{tr}(\mathbf{F}^T\mathbf{F}) \tag{B.22}$$
$$I_3 = \det(\mathbf{F}) \tag{B.23}$$

And its gradients and Hessians.

$$\frac{\partial I_1}{\partial \mathbf{F}} = \mathbf{R} \tag{B.24}$$
$$\frac{\partial I_2}{\partial \mathbf{F}} = 2\mathbf{F} \tag{B.25}$$
$$\frac{\partial I_3}{\partial \mathbf{F}} = \left(\vec{f_1} \times \vec{f_2} \;\middle|\; \vec{f_2} \times \vec{f_0} \;\middle|\; \vec{f_0} \times \vec{f_1} \right) \tag{B.26}$$

$$\text{vec}(\frac{\partial I_1^2}{\partial \mathbf{F}^2}) = \sum_{i=0}^{2} \lambda_i \text{vec}(\mathbf{Q_i})\text{vec}(\mathbf{Q_i})^T \tag{B.27}$$
$$\text{vec}(\frac{\partial I_2^2}{\partial \mathbf{F}^2}) = 2\mathbf{I}_{9\times 9} \tag{B.28}$$
$$\text{vec}(\frac{\partial I_3^2}{\partial \mathbf{F}^2}) = \begin{pmatrix} \mathbf{0} & -\overset{\times}{\mathbf{f}}_2 & \overset{\times}{\mathbf{f}}_1 \\ \overset{\times}{\mathbf{f}}_2 & \mathbf{0} & -\overset{\times}{\mathbf{f}}_0 \\ -\overset{\times}{\mathbf{f}}_1 & \overset{\times}{\mathbf{f}}_0 & \mathbf{0} \end{pmatrix} \tag{B.29}$$

B.2 Corotational Energy

Given the corotational energy:

$$\Psi_{\text{Co}} = \frac{\mu}{2}||\mathbf{F} - \mathbf{R}||_F^2 + \frac{\lambda}{2}\text{tr}^2(\mathbf{S} - \mathbf{I}) \tag{B.30}$$

We would need to rewrite the energy in terms of the S-centric invariants through some algebraic manipulation. Let's start with the left-most term:

$$||\mathbf{F} - \mathbf{R}||_F^2 = ||\mathbf{F}||_F^2 + ||\mathbf{R}||_F^2 - 2\text{tr}(\mathbf{F}^T\mathbf{R}) \tag{B.31}$$
$$= I_2 - 2I_1 + 3 \tag{B.32}$$

To simplify the trace term we applied the following identity $\text{tr}(\mathbf{F}^T\mathbf{R}) = \text{tr}(\mathbf{S}) = I_1$. As for the $||\mathbf{R}||_F^2$ term, the squared Frobenius norm of an orthonormal matrix is the number of dimensions, in this case: 3.

The right-most term can be rewritten as follows:

$$\text{tr}^2(\mathbf{S} - \mathbf{I}) = \text{tr}(\mathbf{S} - \mathbf{I})\text{tr}(\mathbf{S} - \mathbf{I}) \tag{B.33}$$
$$= (\text{tr}\mathbf{S} - \text{tr}\mathbf{I})(\text{tr}\mathbf{S} - \text{tr}\mathbf{I}) \tag{B.34}$$
$$= (\text{tr}\mathbf{S} - 3)(\text{tr}\mathbf{S} - 3) \tag{B.35}$$
$$= \text{tr}^2\mathbf{S} - 6\text{tr}\mathbf{S} + 9 \tag{B.36}$$
$$= I_1^2 - 6I_1 + 9 \tag{B.37}$$

With those 2 simplifications, the final energy written with the S-centric invariants is the following:

$$\Psi_{\text{Co}} = \frac{\mu}{2}(I_2 - 2I_1 + 3) + \frac{\lambda}{2}(I_1^2 - 6I_1 + 9) \tag{B.38}$$

The gradients and Hessians are the following:

$$\frac{\partial \Psi_{\text{Co}}}{\partial I_1} = \lambda I_1 - 3\lambda - \mu \tag{B.39}$$

$$\frac{\partial \Psi_{\text{Co}}}{\partial I_2} = \frac{\mu}{2} \tag{B.40}$$

$$\frac{\partial \Psi_{\text{Co}}}{\partial I_3} = 0 \tag{B.41}$$

$$\frac{\partial^2 \Psi_{\text{Co}}}{\partial I_1^2} = \lambda \tag{B.42}$$

$$\frac{\partial^2 \Psi_{\text{Co}}}{\partial I_2^2} = 0 \tag{B.43}$$

$$\frac{\partial^2 \Psi_{\text{Co}}}{\partial I_3^2} = 0 \tag{B.44}$$

B.3 Stable Neo-Hookean Energy

Given the *Stable Neo-Hookean* energy in thers of the S-centric invariants:

$$\Psi_{\text{SNH}} = \frac{\mu}{2}(I_2 - 3) - \mu(I_3 - 1) + \frac{\lambda}{2}(I_3 - 1)^2 \tag{B.45}$$

The gradients and Hessians are the following:

$$\frac{\partial \Psi_{\text{SNH}}}{\partial I_1} = 0 \tag{B.46}$$

$$\frac{\partial \Psi_{\text{SNH}}}{\partial I_2} = \frac{\mu}{2} \tag{B.47}$$

$$\frac{\partial \Psi_{\text{SNH}}}{\partial I_3} = \lambda I_3 - \lambda - \mu \tag{B.48}$$

$$\frac{\partial^2 \Psi_{\text{SNH}}}{\partial I_1^2} = 0 \tag{B.49}$$

$$\frac{\partial^2 \Psi_{\text{SNH}}}{\partial I_2^2} = 0 \tag{B.50}$$

$$\frac{\partial^2 \Psi_{\text{SNH}}}{\partial I_3^2} = \lambda \tag{B.51}$$

B.4 Analytic Eigensystems of Arbitrary isotropic energies

Following [29] and [15], we present here the generalization of the eigenvalues and eigenvectors of any isotropic energy, defined with respect of the S-centric invariants, and given the singular value decomposition $\mathbf{F} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T$.

The eigenvalues $\lambda_{0...8}$ are:

$$\lambda_{0...2} = \text{eigenvalues}(\mathbf{A}) \tag{B.52}$$

$$\lambda_3 = \frac{2}{\sigma_x + \sigma_y}\frac{\partial \Psi}{\partial I_1} + 2\frac{\partial \Psi}{\partial I_2} + \sigma_z \frac{\partial \Psi}{\partial I_3} \tag{B.53}$$

$$\lambda_4 = \frac{2}{\sigma_y + \sigma_z}\frac{\partial \Psi}{\partial I_1} + 2\frac{\partial \Psi}{\partial I_2} + \sigma_x \frac{\partial \Psi}{\partial I_3} \tag{B.54}$$

$$\lambda_5 = \frac{2}{\sigma_x + \sigma_z}\frac{\partial \Psi}{\partial I_1} + 2\frac{\partial \Psi}{\partial I_2} + \sigma_y \frac{\partial \Psi}{\partial I_3} \tag{B.55}$$

$$\lambda_6 = 2\frac{\partial \Psi}{\partial I_2} - \sigma_z \frac{\partial \Psi}{\partial I_3} \tag{B.56}$$

$$\lambda_7 = 2\frac{\partial \Psi}{\partial I_2} - \sigma_x \frac{\partial \Psi}{\partial I_3} \tag{B.57}$$

$$\lambda_8 = 2\frac{\partial \Psi}{\partial I_2} - \sigma_y \frac{\partial \Psi}{\partial I_3} \tag{B.58}$$

Where $\mathbf{A} \in \mathcal{R}^{3\times 3}$ is defined as follows with the $\{i, j, k\}$ notation $(i \neq j \neq k)$.

$$a_{ii} = 2\frac{\partial \Psi}{\partial I_2} + \frac{\partial^2 \Psi}{\partial I_2^2} + 4\sigma_i^2 \frac{\partial^2 \Psi}{\partial I_2^2} + \sigma_j^2 \sigma_k^2 \frac{\partial^2 \Psi}{\partial I_3^2} + 4\sigma_i \frac{\partial^2 \Psi}{\partial I_1 \partial I_2} + 4 I_3 \frac{\partial^2 \Psi}{\partial I_2 \partial I_3} + 2\frac{I_3}{\sigma_i}\frac{\partial^2 \Psi}{\partial I_3 \partial I_1} \tag{B.59}$$

$$\begin{aligned} a_{ij} = {} & \sigma_k \frac{\partial \Psi}{\partial I_3} + \frac{\partial^2 \Psi}{\partial I_1^2} + \sigma_i \sigma_j \frac{\partial^2 \Psi}{\partial I_2^2} + \sigma_k I_3 \frac{\partial^2 \Psi}{\partial I_3^2} + 2(I_1 - \sigma_k)\frac{\partial^2 \Psi}{\partial I_1 \partial I_2} \\ & + 2\sigma_k (I_2 - \sigma_k^2)\frac{\partial^2 \Psi}{\partial I_2 \partial I_3} + \sigma_k (I_1 - \sigma_k)\frac{\partial^2 \Psi}{\partial I_3 \partial I_1} \end{aligned} \tag{B.60}$$

The general eigenvectors, represented as a matrix $\mathbf{R}^{3\times 3}$ so that its vectorization is the actual eigenvector, are the following:

$$\mathbf{Q}_3 = \frac{1}{\sqrt{2}}\mathbf{U}\begin{pmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}\mathbf{V}^T \tag{B.61}$$

$$\mathbf{Q}_4 = \frac{1}{\sqrt{2}}\mathbf{U}\begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & -1 & 0 \end{pmatrix}\mathbf{V}^T \tag{B.62}$$

$$\mathbf{Q}_5 = \frac{1}{\sqrt{2}}\mathbf{U}\begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \\ -1 & 0 & 0 \end{pmatrix}\mathbf{V}^T \tag{B.63}$$

$$\mathbf{Q}_6 = \frac{1}{\sqrt{2}}\mathbf{U}\begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}\mathbf{V}^T \tag{B.64}$$

$$\mathbf{Q}_7 = \frac{1}{\sqrt{2}}\mathbf{U}\begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}\mathbf{V}^T \tag{B.65}$$

$$\mathbf{Q}_8 = \frac{1}{\sqrt{2}}\mathbf{U}\begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \end{pmatrix}\mathbf{V}^T \tag{B.66}$$

Smith et al calls $\mathbf{Q}_{3...5}$ the *twist* matrices, and $\mathbf{Q}_{6...8}$ the *flip* matrices.

The first 3 eigenvectors are a little more complicated. Given the corresponding on-diagonal scaling modes:

$$\mathbf{D}_0 = \frac{1}{\sqrt{2}}\mathbf{U}\begin{pmatrix}1 & 0 & 0\\ 0 & 0 & 0\\ 0 & 0 & 0\end{pmatrix}\mathbf{V}^T \quad \mathbf{D}_1 = \frac{1}{\sqrt{2}}\mathbf{U}\begin{pmatrix}0 & 0 & 0\\ 0 & 1 & 0\\ 0 & 0 & 0\end{pmatrix}\mathbf{V}^T \quad \mathbf{D}_2 = \frac{1}{\sqrt{2}}\mathbf{U}\begin{pmatrix}0 & 0 & 0\\ 0 & 0 & 0\\ 0 & 0 & 1\end{pmatrix}\mathbf{V}^T \tag{B.67}$$

The eigenvectors are a linear combination of the previous scaling modes. For $i \in [0, 2]$

$$\mathbf{Q_i} = \sum_{j=0}^{2} z_j \mathbf{D}_j \quad \text{where} \quad \begin{cases} z_0 = \sigma_x\sigma_z + \sigma_y\lambda_i \\ z_1 = \sigma_y\sigma_z + \sigma_x\lambda_i \\ z_2 = \lambda_i^2 - \sigma_z^2 \end{cases} \tag{B.68}$$

Bibliography

[1] D. Baraff and A. Witkin, "Large steps in cloth simulation", in *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, 1998, pp. 43–54.

[2] J. Barbič, F. Sin, and E. Grinspun, "Interactive editing of deformable simulations", *ACM Trans. Graph.*, vol. 31, no. 4, Jul. 2012.

[3] A. W. Bargteil, T. Shinar, and P. G. Kry, "An introduction to physics-based animation", in *SIGGRAPH Asia 2020 Courses*, 2020, pp. 1–57.

[4] J. Bonet and R. D. Wood, *Nonlinear Continuum Mechanics for Finite Element Analysis*, 2nd ed. Cambridge University Press, 2008.

[5] O. Civit-Flores and A. Susín, "Fast contact determination for intersecting deformable solids", in *Proceedings of the 8th ACM SIGGRAPH Conference on Motion in Games*, 2015, pp. 205–214.

[6] Z. Fu, T. J. Lewis, R. M. Kirby, and R. T. Whitaker, "Architecting the finite element method pipeline for the gpu", *Journal of computational and applied mathematics*, vol. 257, pp. 195–211, 2014.

[7] J. Georgii, F. Echtler, and R. Westermann, "Interactive simulation of deformable bodies on gpus.", in *SimVis*, 2005, pp. 247–258.

[8] G. H. Golub and C. F. Van Loan, *Matrix computations.* JHU press, 2013.

[9] G. Guennebaud, B. Jacob, *et al.*, *Eigen v3*, http://eigen.tuxfamily.org, 2010.

[10] K. Hildebrandt, C. Schulz, C. von Tycowicz, and K. Polthier, "Interactive space-time control of deformable objects", *ACM Trans. Graph.*, vol. 31, no. 4, Jul. 2012.

[11] Y. Hu, T.-M. Li, L. Anderson, J. Ragan-Kelley, and F. Durand, "Taichi: A language for high-performance computation on spatially sparse data structures", *ACM Transactions on Graphics (TOG)*, vol. 38, no. 6, pp. 1–16, 2019.

[12] D. L. James and K. Fatahalian, "Precomputing interactive dynamic deformable scenes", *ACM Trans. Graph.*, vol. 22, no. 3, pp. 879–887, Jul. 2003.

[13] L. Jeřábková, G. Bousquet, S. Barbier, F. Faure, and J. Allard, "Volumetric modeling and interactive cutting of deformable bodies", *Progress in Biophysics and Molecular Biology*, vol. 103, no. 2, pp. 217–224, 2010, Special Issue on Biomechanical Modelling of Soft Tissue Motion.

[14] T. Kim, "A finite element formulation of baraff-witkin cloth", in *Computer Graphics Forum*, Wiley Online Library, vol. 39, 2020, pp. 171–179.

[15] T. Kim and D. Eberle, "Dynamic deformables: Implementation and production practicalities", in *ACM SIGGRAPH 2020 Courses*, 2020, pp. 1–182.

[16] M. Liu and D. Gorman, "Formulation of rayleigh damping and its extensions", *Computers & Structures*, vol. 57, no. 2, pp. 277–285, 1995.

[17] P. Martin Garcia, "Aplicació del material point method a la deformació d'objectes", B.S. thesis, Universitat Politècnica de Catalunya, 2020.

[18] A. McAdams, A. Selle, R. Tamstorf, J. Teran, and E. Sifakis, "Computing the singular value decomposition of 3x3 matrices with minimal branching and elementary floating point operations", University of Wisconsin-Madison Department of Computer Sciences, Tech. Rep., 2011.

[19] U. Meier, O. López, C. Monserrat, M. C. Juan, and M. Alcaniz, "Real-time deformable models for surgery simulation: A survey", *Computer methods and programs in biomedicine*, vol. 77, no. 3, pp. 183–197, 2005.

[20] M. Mooney, "A theory of large elastic deformation", *Journal of applied physics*, vol. 11, no. 9, pp. 582–592, 1940.

[21] M. Müller, J. Dorsey, L. McMillan, R. Jagnow, and B. Cutler, "Stable real-time deformations", in *Proceedings of the 2002 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, ser. SCA '02, San Antonio, Texas: Association for Computing Machinery, 2002, pp. 49–54.

[22] M. Müller and M. Gross, "Interactive virtual materials", in *Proceedings of Graphics Interface 2004*, ser. GI '04, London, Ontario, Canada: Canadian Human-Computer Communications Society, 2004, pp. 239–246.

[23] M. Pharr, W. Jakob, and G. Humphreys, *Physically based rendering: From theory to implementation*. Morgan Kaufmann, 2016.

[24] C. Rankin and F. Brogan, "An element independent corotational procedure for the treatment of large rotations", 1986.

[25] J. R. Shewchuk *et al.*, *An introduction to the conjugate gradient method without the agonizing pain*, 1994.

[26] H. Si and A. TetGen, "A quality tetrahedral mesh generator and three-dimensional delaunay triangulator", *Weierstrass Institute for Applied Analysis and Stochastic, Berlin, Germany*, vol. 81, 2006.

[27] E. Sifakis and J. Barbic, "Fem simulation of 3d deformable solids: A practitioner's guide to theory, discretization and model reduction", in *Acm siggraph 2012 courses*, 2012, pp. 1–50.

[28] B. Smith, F. D. Goes, and T. Kim, "Stable neo-hookean flesh simulation", *ACM Trans. Graph.*, vol. 37, no. 2, Mar. 2018.

[29] B. Smith, F. D. Goes, and T. Kim, "Analytic eigensystems for isotropic distortion energies", *ACM Trans. Graph.*, vol. 38, no. 1, Feb. 2019.

[30] R. Tamstorf, T. Jones, and S. F. McCormick, "Smoothed aggregation multigrid for cloth simulation", *ACM Transactions on Graphics (TOG)*, vol. 34, no. 6, pp. 1–13, 2015.

[31] D. Terzopoulos, J. Platt, A. Barr, and K. Fleischer, "Elastically deformable models", in *Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, 1987, pp. 205–214.

[32] X. Zhang and Y. J. Kim, “Interactive collision detection for deformable models using streaming aabbs”, *IEEE Transactions on Visualization and Computer Graphics*, vol. 13, no. 2, pp. 318–329, 2007.