

Interactive Finite Element Simulation of Deformable Objects

Márton Solti

Supervisor: Attila Kossa

*Young man, in mathematics you don't understand things.
You just get used to them.*

– János Neumann (a.k.a *Good ole' Johnny*)

Glossary of Symbols

General Notation

Symbol	Description
\mathbb{R}	set of real numbers
a	scalar
\mathbf{a}	vector
\mathbf{A}	matrix
\mathbb{A}	higher-order tensor
a_i	i^{th} element of a vector
a_{ij}	element in the i^{th} column and j^{th} row of some matrix
$\frac{\partial f}{\partial x}$	partial derivative of f in terms of x
$\text{vec}(\cdot)$	vectorization operation
$\ \cdot\ _F^2$	squared Frobenius norm
\mathbf{U}	left singular vectors of some matrix
\mathbf{V}	right singular vectors of some matrix
Σ	singular values of some matrix
λ_i	i^{th} eigenvalue of some matrix
\mathbf{q}_i	i^{th} eigenvector of some matrix
\mathbf{Q}_i	i^{th} eigenmatrix of some tensor

Finite Element Method

Symbol	Description
\mathbf{K}	(tangent) stiffness matrix
\mathbf{M}	mass matrix
\mathbf{C}	damping matrix
\mathbf{f}_{ext}	nodal external forces
\mathbf{f}_{int}	nodal internal forces

Symbol	Description
Π	potential energy
$\Psi(\mathbf{F})$	strain energy density function
\mathbf{u}	nodal displacements
$\dot{\mathbf{u}}$	velocity change a.k.a. $\frac{d\mathbf{u}}{dt}$
\mathbf{x}	spatial position $[x, y, z]^T$
$\dot{\mathbf{x}}$	velocity, a.k.a. $\frac{d\mathbf{x}}{dt}$
$\ddot{\mathbf{x}}$	acceleration, a.k.a. $\frac{d^2\mathbf{x}}{dt^2}$
h	timestep size
v	volume of a volumetric element
a	area of a shell element

Continuum Mechanics

Symbol	Description
\mathbf{x}	a value (\mathbf{x}) at the material configuration
$\bar{\mathbf{x}}$	a value (\mathbf{x}) at the spatial configuration
ϕ	motion
\mathbf{F}	deformation gradient
\mathbf{C}	right Cauchy-Green strain tensor
\mathbf{E}	material Green strain tensor
\mathbf{R}	rotational part of \mathbf{F}
\mathbf{S}	stretching part of \mathbf{F}
J	volume change
σ	Cauchy stress tensor
\mathbf{P}	first Piola-Kirchhoff stress tensor
I_C	first Cauchy-Green invariant
II_C	second Cauchy-Green invariant
III_C	third Cauchy-Green invariant
I_1	first Smith et al. invariant
I_2	second Smith et al. invariant
I_3	third Smith et al. invariant
μ	Lamé's first parameter
λ	Lamé's second parameter
E	Young's modulus
ν	Poisson's ratio

Contents

1	Introduction	1
2	Computer Graphics and Animation	2
2.1	Computer Graphics	2
2.1.1	A typical scene	2
2.1.2	Real-Time vs. Offline	3
2.2	Animation	4
2.3	Physically Based Animation	5
2.4	Physically Based Deformables	6
3	Fundamentals of FEM	8
3.1	The Essence of FEM	8
3.1.1	The Lumped-Paramter Mathematical Model	9
3.1.2	Structural Elements	11
3.1.3	The Direct Stiffness Method	13
3.1.4	Example: Mass-Spring Systems	14
3.2	Finite Element Formulation	15
3.2.1	Conservative Systems	15
3.2.2	Balance of Total Potential Energy	15
3.2.3	The Newton-Raphson Iterative Scheme	17
3.3	Dynamics Simulation	20
3.3.1	Equilibrium of a Dynamic Process	20
3.3.2	Time Integration	21
3.4	FEM: Beyond Deformation Simulation	23
3.4.1	Surface Modelling	23
3.4.2	Geometry Optimization	24

4 Fundamentals of Continuum Mechanics	25
4.1 Kinematics	25
4.1.1 Motion	25
4.1.2 Deformation Gradient	27
4.1.3 Strain	28
4.1.4 Polar Decomposition	29
4.1.5 Volume change	30
4.2 The Concept of Stress	31
4.3 Balance Laws	32
4.4 The Hyperelastic Constitutive Model	33
4.4.1 What Does an Energy Function Look Like?	33
5 Jacobians and Hessians	35
5.1 Higher-Order Tensor Manipulations	36
5.1.1 Mental Representation of Higher-Order Tensors	36
5.1.2 Multiplication With Higher-Order Tensors	37
5.1.3 Vectorization	38
5.2 Computing Forces	39
5.2.1 Examples	39
5.2.2 The next step	41
5.3 Computing Force Gradients	41
5.3.1 The Mental Image of The Energy Hessian	42
5.3.2 Gradient Computation	43
5.4 The Cauchy-Green Invariants	44
5.4.1 Generic Invariant Gradients and Hessians	45
5.4.2 Force Jacobian the Cauchy-Green Way	46
5.4.3 Examples	47
5.4.4 As-Rigid-As-Possible: Things Go Terribly Wrong	47
5.5 A New Set of Invariants?	48
5.5.1 Invariants as Rotation Removers	48
5.5.2 Invariants as Geometric Measures	50
5.6 S-based Invariants	52
5.6.1 Does ARAP Work Now?	53
5.7 The Eigenmatrices of the Rotation Gradient	54
5.7.1 Eigenvalues, Eigenvectors, Eigendecomposition	55
5.7.2 What's an Eigenmatrix?	55

5.7.3	Structures Lurk in the Decomposition of an Eigenmatrix	57
5.7.4	Building the Rotation Gradient	57
5.8	New Generic Hessian	58
5.8.1	Does ARAP Work Now?	59
6	Keeping the Hessian Positive Definite	60
6.1	Solving a Linear System Iteratively	61
6.1.1	Meaning of Matrix's Definiteness	61
6.1.2	The Method of Steepest Descent	62
6.1.3	The Method of Conjugate Gradients	63
6.2	Solving the Issue of Definiteness	64
6.3	Methods for Hessian Projection	64
6.4	Analytical Eigensystem of ARAP	65
6.5	Analytical Eigendecompositions of Arbitrary Energies	68
7	Some Production Practicalities	69
7.1	Geometric Calculation of the Deformation Gradient	69
7.1.1	The Mechanical Engineer's Way	69
7.1.2	The Simulator Programmer's Way	70
7.1.3	Calculating the Derivative	72
7.2	Force Computation Trick	73
7.3	Boundary Condition Projection	74
7.4	Interpolating the Results on a Higher Resolution Mesh	76
8	Implementing a Finite Element Simulator	77
8.1	Core Algorithm	77
8.2	A Naive Simulator Architecture	79
8.2.1	Configuring the Simulator	79
8.2.2	The Solver Class	79
8.2.3	Resources	80
8.2.4	Initialization	83
8.2.5	Simulation loop	84
8.2.6	Time Integration	85
8.2.7	Building The Force Jacobian	87
8.2.8	Naive Parallelization of the Global Matrix Assembly	89
8.3	Drawing the Results On The Screen	91
8.3.1	A Basic Real-Time Graphics Application	92

8.3.2	Drawing the Model	94
9	Case Studies	96
9.1	Testing Rig, Example Models	96
9.2	Testing the Core Algorithm	96
9.3	Projected Newton Solver	97
9.4	Dynamics Simulations: What?	97
10	Summary	99
A	An Introductory Example of FEM	100
A.1	System Idealization	100
A.2	Element Equilibrium	100
A.2.1	Linear Elastic Spring as a Structural Element	101
A.3	Element Assemblage, Solution	102
A.4	Other Lumped-Paramter Models	104
B	The Method of Steepest Descent	105
B.1	What is a Linear System, Anyways?	105
B.2	The Method of Steepest Descent	108
C	A Config File	111

Chapter 1

Introduction

Did you know that simulating deformables with the Finite Element Method (FEM) is also pretty popular in the entertainment industry as well? Yes, the same FEM we all know very well, is behind all the cloth and hair simulations, character animations we see in those Hollywood blockbusters and AAA games.

This thesis is about the application of FEM, but in the discipline of computer graphics. In the old days, physics simulation in computer graphics – in movies and games – were about creating some *physically plausible* visual effect with some smart tricks – *hacks*. The field of deformable simulation advanced greatly in the last few decades: they built upon the FEM techniques found in the mechanical engineering literature while bringing some exciting innovations to the field as well.

The thesis' title is *interactive* FEM simulation of deformables, which might suggest that this whole text will be about bit hacks and weird x86 assembly instructions. You usually have to wait, when you run a simulation in ANSYS, right? How do you make this interactive? Of course, the *implementation* is equally important in this case, but we will look at the *math* behind the sheer *craftsmanship* of high performance simulator programming instead.

First we will go through the basic concepts of computer graphics and animation, FEM, and continuum mechanics in Chapter 2-4. Then we will start our journey into the ins and outs of interactive volumetric simulations in the field of computer graphics. Just to offer you a spoiler, we will look deeply into the efficient computation (Chapter 5) as well as the *stability* (Chapter 6) of the tangent stiffness matrix $\mathbf{K} = \frac{\partial^2 \Psi(\mathbf{F})}{\partial \mathbf{x}}$, mostly following the work of Pixar Research. (Smith et al. in [24], [25] and [15]) We were able to implement a super-robust quasistatic simulator (Chapter 8) which recovers to the rest state even if the whole mesh is squished into a plane or a point. (Chapter 9) Along the way we will look at some clever tricks applied in the industry. (Chapter 7)

Chapter 2

Computer Graphics and Animation

2.1 Computer Graphics

Computer Graphics is an umbrella term for any method that is about synthesizing and manipulating visual content. Major subfields include:

- **Geometry** processing, or *computational geometry*, which is about the representation and processing of surfaces. We will look at some examples from this subfield, as FEM is pretty popular there.
- **Rendering**, or *image synthesis* is about telling what color each pixel should be. Rendering usually simulates the physics of light in order to create realistic images.
- **Animation** is about the description of geometry, that *move or deform over time*. It's basically describing a geometry, but in 4D.

2.1.1 A typical scene

Let's describe a typical example of creating a computer generated image of a *3D scene*, pictured on Fig. 2.1. A *scene* contains all the resources, models, which participate in the image synthesis. It's the whole thing on Fig. 2.1.

In order to *draw* an image of this scene with the computer, we need to simulate all the photons that would get into our eye, if we would look at the same scene in real life (IRL). *Image synthesis* – a.k.a. *rendering* – is all about that. It simulates the physics of light: photons are emitted from the *light sources* bounce around the scene. They interact with the surfaces on the way, and eventually hit our digital *eye*, determining what color a pixel should be.

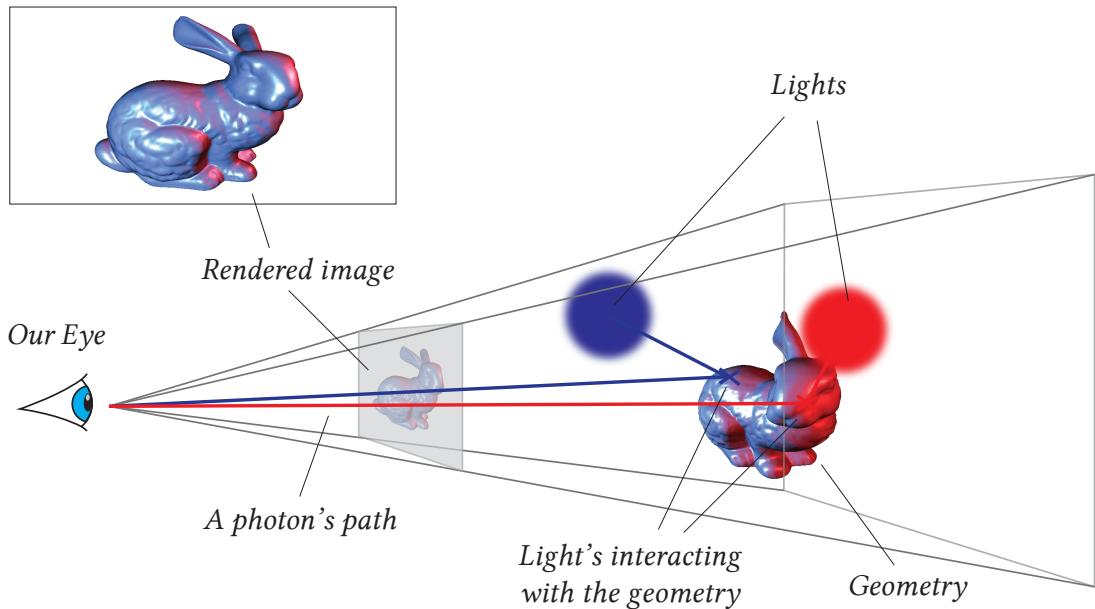


Figure 2.1: A typical scene

The set of surfaces that are present in the scene – i.e. the drawable models’ *geometry* – are usually defined as a *set of triangles*. The surfaces’ material defines how light’s interacting with it, which also determines how it’s going to look like: is it a shiny silver one, or is it matte black? The interaction – and the look, or the *shading* – will be very different. There has been thick books written about how light’s interacting with different surfaces, e.g. [20]. When you hear *shader code*, it’s usually some code running on the GPU, calculating the result of some specific surface interaction.

2.1.2 Real-Time vs. Offline

A thing we have to make clear in the beginning, is the difference between *real-time* and *offline* simulations.

A *real-time* simulation means, that the solution is *running real-time*: each frame of data is calculated *’on the fly’*, right before the image is shown on the screen. This usually means that we have a specific amount of time to calculate *everything* – $1/30 = 0.0333s$ if we display 30 images per second, or $1/60 = 0.0167s$ if we display 60 images a second. Good examples for real-time solutions are games, of course, but there are a vast field of applications outside the entertainment industry, like simulators for training professionals – e.g. forklift operators or fighter jet pilots.

Offline simulations on the other hand, are usually for stuff that is *not* feasible for real-time. In case of rendering, while real-time graphics is for games, offline rendering is for animated movies and CGI effects. They both simulate light, but use fundamentally different methods for that.

2.2 Animation

We talked about *geometry* and *rendering* a lot, but haven't said a word about *animation* since I introduced it. Animation is essentially about describing geometry, but in *time* as well – so instead of $[x, y, z]^T$ we have $[x, y, z, t]^T$

Usually, an artist – an animator – creates the animation with some animation software – like Maya. However, a typical mesh's vertex count is in the magnitude of thousands – millions. An artist moves all these points by hand? It is *certainly* a possibility, but most of the time it's just stupid, and we need a *more optimal* (sic!) way. One key method to mention here is *skeletal animation*.

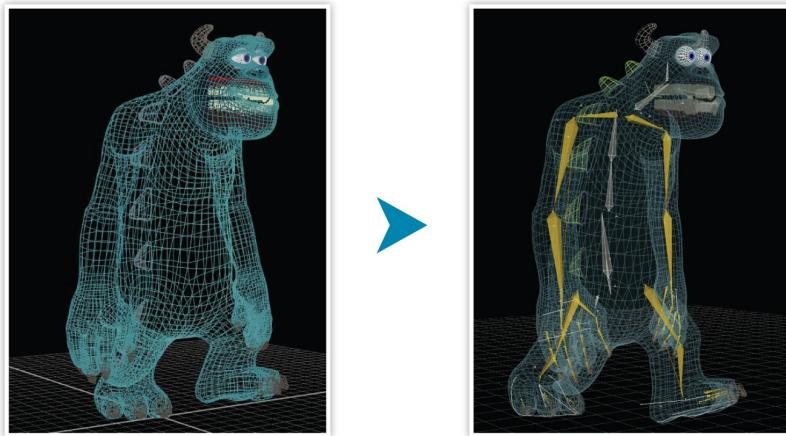


Figure 2.2: Skeletal animation [21]

Skeletal animation (Fig. 2.2) is usually used in character animation, where a character has two representations: a surface representation used to draw the character (called the mesh or skin) and a hierarchical set of interconnected parts (called bones, and collectively forming the skeleton or rig), a virtual armature used to animate the mesh: the position of these bones are set for some specific points in time – called *keyframes* – and the software *interpolates* between the keyframes in some manner.

2.3 Physically Based Animation

What about other kinds of animations? E.g. imagine you want to animate a realistically moving curtain. How do you do that? You can *certainly* do it by hand, but it would probably cost hundreds of hours of work.

Why not just simulate it? Isn't it just some FEM simulation at the end of the day? The answer is yes, such a *cloth simulation* is *nothing but* a FEM simulation. The area of physics simulation based animations is usually called Physically Based Animation.

Physically Based Animation includes every type animation what's done simulating some physical phenomena. Important examples include:

- **Rigid Body Dynamics** is the same rigid body dynamics we talk about in computational mechanics, however, usually in a real-time, interactive framework. Good examples of such frameworks are *PhysX* and *Havok*. It has to be noted that
- **Collision Detection**, the act of detecting the intersection of two or more objects, is an equally important part of rigid body dynamics frameworks. Two main engineering challenges are:
 - First you have to find explicit formulas for figuring out if two pieces of geometries intersect or not. Notable methods include the Gilbert–Johnson–Keerthi distance algorithm. [7]
 - Just elaborate on this a bit: how do you know if any of n objects intersect with any other object in the scene? The naive way would be that you check every other object, if it's intersected by the one in discussion. That's at least n^2 steps – which could be a lot! We need to optimize this, and the '*basic approach*' is to build some spatial acceleration structure, where we could discard almost all the elements from the problem.
- **Simulation of Fluids** – be it liquid or even sound – is usually done the usual way, by approximating the Navier-Stokes equation on a Cartesian grid. However, Eulerian and hybrid methods are definitely on the rise with the (re)introduction of the Material Point Method by the Disney Research Team: in [30] they proposed a new method for simulating snow. (Fig. 2.3a)
- **Deformables** – just like fluids – is done the usual way, using FEM. However there are notable exceptions, pushing a (semi-)particle-based solver, e.g. the aforementioned Material Point Methods [30] or Position Based Dynamics [19].

2.4 Physically Based Deformables

In 1987, Terzopoulos et. al introduced elastically deformable objects to the computer graphics community in a seminal paper. [34] Since then numerous researchers have partaken in the quest for the visually and physically plausible animation of deformable objects.

Your natural, engineering instinct might suggest, that there has to be some trick involved in the models used by deformables in this field. There are a few clever tricks indeed, some of them are presented along the way in this text, but by and large, we are talking about the same, '*basic*' FEM and continuum mechanics we all know and love. Notable examples from the industry:

- **Cloth Simulation** was the original problem which sparked innovation in the late 90s. Everyone in the field knows the original Baraff-Witkin paper. [2] The method presented there was used to simulate Boo's T-Shirt in Monsters Inc. (See Fig. 2.3b)
- **Character Animation** is another field which was pioneered by Pixar using FEM. They were the first to use a sophisticated volume simulation, in order to simulate the fleshy behaviour of an animated character's flesh. (Fig. 2.3c) More specifically, they were not able to model Mr. Incredible's (2004) muscular body with their existing linear interpolation scheme, so they had to design a completely new solution, based on nonlinear FEM and a hyperelastic constitutive model. [11]
- Have you noticed how great **Hair Simulations** became? Disney Research is the big player in the hair game. [14] Aside their supreme physically based hair simulation, they have a very intuitive toolset which allows the artist to art-direct the movement of the hair. (Fig. 2.3d)
- This new paper by Facebook Reality Labs [26] is so great that I have to make a separate point for **Hand Physics**. Hand physics deals with the real-time simulation of the user's hands in virtual reality. Smith et. al achieve an *epic* hand model by constraining a vision-based tracking algorithm with a FEM-based, hyperelastic (Neo-Hookean) deformable model. Their algorithm is able to handle self-interaction like massaging of the hands. (Fig. 2.3e)



Figure 2.3: Examples of Physically Based Animation: a – Material Point Method for snow simulation [30]; b – Implicit backward Euler formulation for cloth simulation [2]; c – Stable Neo-Hookean flesh simulation for character animation [24]; d – Art-directed hair simulation [14]; e – Constraining dense hand surface tracking with elasticity for VR hands [26];

Chapter 3

Fundamentals of FEM

3.1 The Essence of FEM

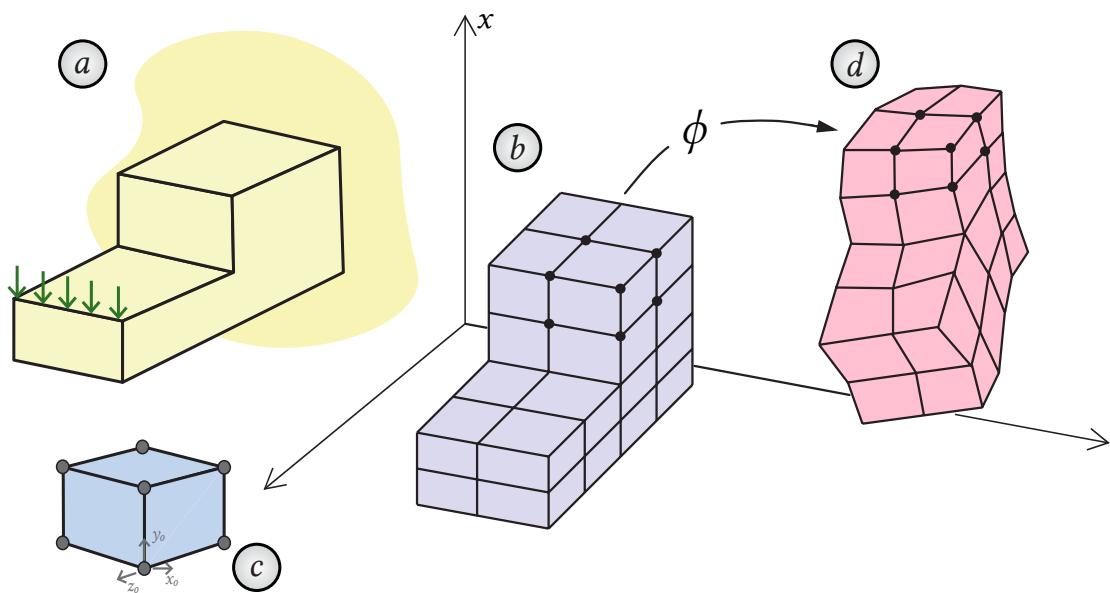


Figure 3.1: a - Calculating the response of a deformable object, one end is built in to the wall, while the other end is loaded with some forces. b - The problem is idealized as an assemblage of brick elements (c). d - the response of the assemblage of elements – the movement of the all the nodes, ϕ – is calculated according to the loads/boundary conditions.

The Finite Element Method (FEM) is the core technology behind many applications for engineering analysis. Despite we are '*just*' going to use it for some Physically Based Animation, the math and physics behind it stays the same.

3.1.1 The Lumped-Paramter Mathematical Model

Just to be sure we are talking about the same FEM, let's walk through the fundamentals of it. I think the best summary is the definition of '*lumped-parameter mathematical models*' by Bathe in [5]:

"The essence of a lumped-parameter mathematical model is that the state of the system can be described directly with adequate precision by the magnitudes of a finite (and usually small) number of state variables. The solution requires the following steps:

1. **System idealization:** the actual system is idealized as an assemblage of elements.
2. **Element equilibrium:** the equilibrium requirements of each element are established in terms of state variables
3. **Element assemblage:** the element interconnection requirements are invoked to establish a set of simultaneous equations for the unknown state variables
4. **Calculation of response:** the simultaneous equations are solved for the state variables, and using the element equilibrium requirements, the response of each element is calculated."

Idealization

According to this definition, we need to *idealize the system* first. This means that we relaxate the problem a bit, we make some assumptions and simplifications in order to make the problem computable. This is most important part.

In order to do so, first we need to figure out what kind of problem we are dealing with. Are we doing a fluid simulation? A simulation of dynamic deformables? Maybe a coupled fluid-deformable sim? All of them are possible.

On Fig. 3.1, you can see a neat example of system idealization: instead of integrating the fundamental equations of elasticity through that L-shaped thing (a), we build an L-shaped thing (b) from small bricks (c), with which we can compute stuff much easier.

State Variables, Element Equilibrium

The problem in discussion determines what kind of elements we are going to use. The structural elements are the atomic building blocks of FEM. We define structural elements, whose behavoir can be fully characterized and understood. We then use these elements to build much more complex structures – just like on Fig. 3.1.

The structural elements' behaviour – or *response* – is characterized by a set of *state variables*. The ideas of the *response* and *state variables* are much easier to digest through an example, so let's look at the response of a structural element: a linear elastic spring. (Fig. 3.2)

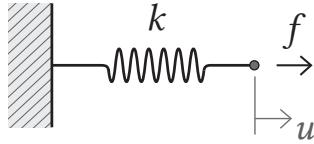


Figure 3.2: A linear elastic spring with single degree of freedom

The *response* of a spring, that has one of its end attached to some fixed object, while the other end is pulled by some force of magnitude f , is calculated according to Hooke's law as

$$ku = f \quad (3.1)$$

This is the element equilibrium! Here the *response* is the displacement of the pulled end, denoted by u . u is the state variable! k on the other hand, is the *spring constant*. k is a positive, real number; it can be thought of the material property of the spring. k and u together completely characterize the response – or *behaviour* – of this ideal spring.

Loosely speaking, f can be thought of as some external 'energy' which 'perturbs' the system from its steady state. As a response, the system 'stores' this energy in the spring itself, by equilibrating f with ku , hence the equation $ku = f$.

Element Assemblage, Calculation of the Response

Now we've got our state variables and structural elements defined, and we also have some notion what the system's response is. All that's left is to elaborate a bit on the element assemblage and the calculation of its response.

In order to build some structures from the structural elements, we need to define how they are connected together. This is done by accumulating each individual elements' '*behavioural data*' in the *global stiffness matrix* \mathbf{K} using the *direct stiffness method*. The process is explained later in Sec. 3.1.3, but what we are doing is essentially expanding the idea of the $ku = f$ for an n dimensional problem, such that

$$\boxed{\mathbf{K}\mathbf{u} = \mathbf{f}_{ext}} \quad (3.2)$$

Here we stacked all the state variables in a vector \mathbf{u} , which denotes each elements' displacement, while the external forces are stacked in the same manner in \mathbf{f}_{ext} ; both

\mathbf{u} and \mathbf{f}_{ext} are $\in \Re^n$. The global stiffness matrix $\mathbf{K} \in \Re^{n \times n}$ is characterizing the system's response as a whole.

Plugging in any force vector \mathbf{f} to 3.2, we have a nice linear system which we can solve for \mathbf{u} . So cool! And actually Eq. 3.2 is the fundamental equation of FEM; basically we always want to solve this very equation. But where does this \mathbf{K} come from?

You probably know the answer already, and have a hard time not to fall asleep reading this chapter. If it's not the case, I recommend you to read Appendix A, where I go through an easy example wrapping all the things up presented in this section.

3.1.2 Structural Elements

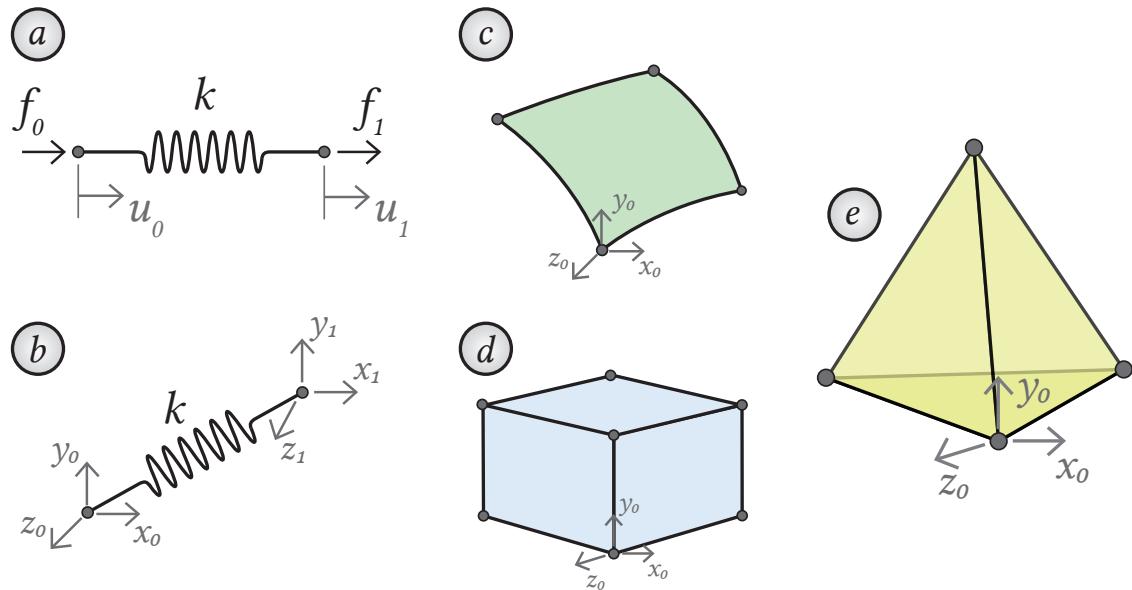


Figure 3.3: Different Structural Elements

The structural elements represent the fundamental building block of FEM. The state variables are usually some spatial points called nodes; an element connects some nodes together. The different elements are connected together with these nodes. In the simulations, we solve the equation $\mathbf{Ku} = \mathbf{f}_{ext}$ for the nodal displacements \mathbf{u} , and interpolate the results to acquire the values inside the element.

We define structural elements, whose behaviour can be fully characterized and understood, and use these elements to build much more complex structures. This is done by accumulating each individual elements' '*behavioural data*' in the global stiffness matrix. This '*behavioural data*' is in the form of *element stiffness matrices*.

1D elements

In case of some special structures, like trusses and beams, *element stiffness matrices* can be acquired in closed form. Let's derive the stiffness matrix of the *1D truss element* as of Fig. 3.3a. This is pretty much $ku = f$ again, but the equilibrium now is a *system of equations*

$$\begin{aligned} k(u_0 - u_1) &= f_0 \\ k(-u_0 + u_1) &= f_1 \end{aligned} \quad (3.3)$$

which can be reformulated as

$$k \begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix} \begin{bmatrix} u_0 \\ u_1 \end{bmatrix} = \begin{bmatrix} f_0 \\ f_1 \end{bmatrix} \quad (3.4)$$

in this equation,

$$k \begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix} = \mathbf{K}^{(e)} \quad (3.5)$$

is the *element stiffness matrix of the 1D truss element*. Actually we would get back our *rank-1* $ku = f$ with $u_0 = 0$ and $u_1 = u$. Nice! And this idea scales pretty well to *3D truss elements*, like the one on Fig. 3.3b.

Two- and Three-Dimensional Elements

The big picture is that you are essentially doing the same thing with continuous structures: you can build 2D structures from 2D elements, like the quad element on Fig. 3.3c; and 3D structures from 3D, volumetric elements, like the hexa '*brick element*' on Fig. 3.3d or the classic tetrahedron a.k.a. tet on Fig. 3.3e. However, the math is very different behind it!

In the analysis of some '*1D*', beam or truss structures, the exact element stiffness matrices can be calculated. The stiffness properties of a beam element are physically the element end forces that correspond to unit element end displacements – in beam theory!

However, if we consider more general, 2D or 3D continuous structures, we need to use a very different approach to approximate the actual displacements. The reason is that we do *not* know the exact displacement functions opposed to the case of truss and beam elements. The result is going to be a set of *partial differential equations*, whose equilibrium are not satisfied in general, but this error is reduced as the finite element idealization of the structure or the continuum is refined.

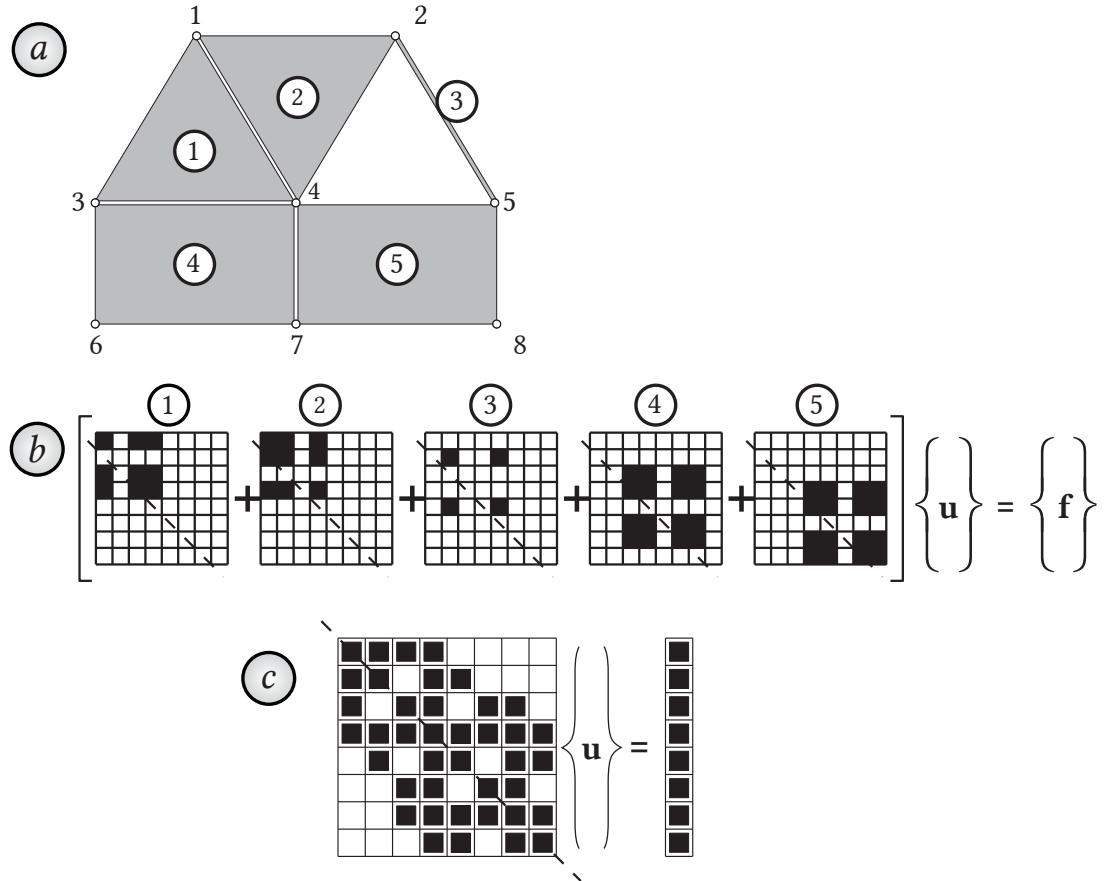


Figure 3.4: The general pattern: a – The system is an assemblage of structural elements. b – Element stiffness matrices are formulated. c – The element stiffness matrices are accumulated to the global stiffness matrix by adding the element matrix to the corresponding degrees of freedom, according to the element interconnection. [36]

There are many ways to derive these differential equations in discussion, which are more specifically the general formulation of the finite element method. In this text, we are going to approach it as a *minimization of the total potential of the system*; presented later on in Sec. 3.2.

3.1.3 The Direct Stiffness Method

Quick recap: the basic process is that the complete structure is idealized as an assemblage of individual structural elements. We have now some understanding what the elemental stiffness matrices are.

The next step is to assemble the global stiffness matrix \mathbf{K} , characterizing the system as a whole. The *direct stiffness* method states that the global stiffness matrix \mathbf{K} can be calculated by summing up the element stiffness matrices:

$$\mathbf{K} = \sum_i^n \mathbf{K}^{(i)} \quad (3.6)$$

where n is the number of element, and $\mathbf{K}^{(i)}$ is the i^{th} element's stiffness matrix, but *expanded in the dimensions of the global degrees of freedom* by filling in zeros at positions related to nodes not adjacent to the element.

In order to consolidate this idea together with the whole lumped-parameter mathematical process, let's consider a new example, shown on Fig. 3.4.

3.1.4 Example: Mass-Spring Systems

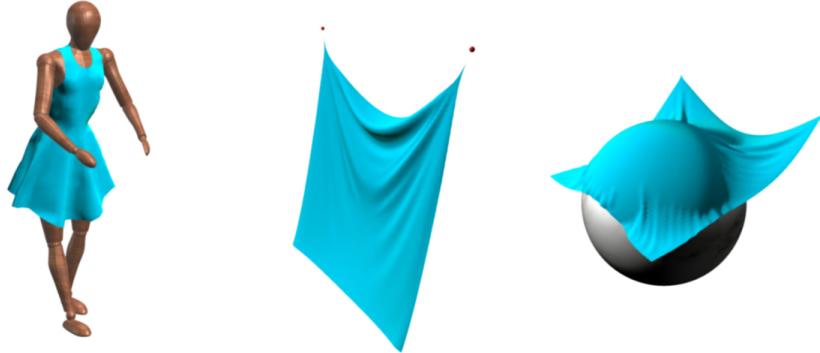


Figure 3.5: Mass-Spring Simulation of cloth. [16]

Believe it or not, most deformable simulations of cloth and volumetric objects were based on the simple spring model I presented in this section. Instead of integrating the proper finite element formulation, they just built some approximate structures from masses and springs, called – not so surprisingly – Mass-Spring Systems.

Despite its simplicity, you can make pretty nice sims with such a simple formulation as well. Just take a look at Fig. 3.5: Mass-Spring simulation of cloths could work really well. There is a catch of course: this is *not* how cloth's physics work at all. You have to tweak some '*arbitrary*' parameters to get the '*visual effect*' you want to achieve. That is why I would not call this a *physically based* method.

3.2 Finite Element Formulation

For the '*proper*' finite element formulation we need to pretty much do the same thing what we did in the analysis of truss structures: first we need to idealize the structure as an assemblage of elements. The only difference is that the elements are going to abstract some 2D or 3D continuum – we already talked about this in Sec. 3.1.2. In the end, we are going to end up with the same element stiffness matrices, which can be accumulated in the same manner – with the direct stiffness method – in the global stiffness matrix. However, it will take some work to make the calculation of the element stiffness matrices equally easy and pretty. Don't be afraid, we will end up with the same, comfortable $\mathbf{Ku} = \mathbf{f}$. We'll start our journey with the law of conservation of energy.

3.2.1 Conservative Systems

The law of conservation of energy states that the total energy of an isolated system remains constant. You can loosely state this as

$$\sum \Pi = \text{const} \quad (3.7)$$

where Π is some conservative – or *potential* – measure. E.g. chemical energy is converted to kinetic energy when a stick of dynamite explodes.

From basic calculus, we also know, that if a function is constant, its derivative must be 0, that is

$$\frac{\partial \Pi}{\partial \mathbf{x}} = 0 \quad (3.8)$$

This is a very important property!

Also a very important property of a conservative force field is that it's independent from the path taken. Consider now a more pacifist example. We drop a bowling ball – or a feather – located at $\mathbf{x} = [x, y, z]^T$ down in a huge vacuum chamber. The only force we reasonably expect to act on the body is the gravitational field, which is a conservative measure. The body has potential energy $\Pi(\mathbf{x}) = mgx$ where $\mathbf{g} = [0, 0, g]^T$. Now, following Eq. 3.8 we can then obtain the gravitational force as the negative gradient of the potential energy with respect to the object position \mathbf{x} :

$$\mathbf{f} = -\frac{\partial \Pi}{\partial \mathbf{x}} = [0, 0, -mg]^T \quad (3.9)$$

3.2.2 Balance of Total Potential Energy

Let's go back to deformables, and try to apply this idea to them. First, we need to postulate that we only consider conservative energies. Consider it done!

If the internal potential energy Π_{int} in a deformable object is aggregated in the system by some external source Π_{ext} , we can state that

$$\Pi_{int} + \Pi_{ext} = \text{const} \quad (3.10)$$

So far so good! Now going back to the *lumped-parameter mathematical model* (Sec. 3.1.1), we need to find some *state variables* in order to calculate the *response* of the system.

In case of '*highly deformable*' deformables, this is the territory of *continuum mechanics*. Continuum mechanics is a separate discipline on its own, dealing with a mechanical behaviour of materials modeled as a continuous mass. Chapter 4 is all about the fundations continuum mechanics, so let's just cut to the chase now.

Our state variable is going to be the *deformation gradient* F . It is calculated for each continuum element, and all you need to know now that it quantifies how much deformation – rotation and scaling – that element has sustained. We are going to exclusively write all calculations in terms of F , so it's hard to overstate its importance.

We got our state variables; what to do with them? This is a really hard question to answer. Intuitively, the question is more like: if we squash the element *some amount, how much energy does it generate?* If I squash it just a little bit, the system should probably '*contain less energy*' compared to the case, when I completely crush it to the point it becomes a pancake.

Actually this is two separate question:

1. How squashed am I? Or: how badly deformed am I compared to my original shape?

We already answered that: the deformation gradient F measures this stuff.

2. How much energy does this cost? Once I know exactly how deformed I am, what should I do with this knowledge?

In order to answer this question, we need to exploit the fact that we are talking about a conservative system. In this case the energy is fully determined by the deformation gradient, and it has a special name: strain energy. An alternative definition could state that the consequence of elastic deformation is the accumulation of strain energy.

The measure we are going to use is the strain energy density function $\Psi(F)$. It assign a deformation score for a corresponding F , which is a number, so if you want to sound like a true intellectual, you can call it a *scalar valued tensor function*. Ψ measures the strain energy per unit undeformed volume on an infinitesimal domain dV . Integrating through the entire body Ω_0 yields the total potential energy of the system:

$$\Pi_{int} = \int_{\Omega_0} \Psi(F) dV \quad (3.11)$$

I don't know if it's just me, but I tend to get pretty anxious when I see an integral. We are lucky because there's a pretty intuitive way to get rid of it, presented in Sec. 7.1. We will use a special case of a tetrahedron with piecewise linear bases, with a single quadrature point in the center of it. In this case Eq. 3.11 becomes

$$\Pi_{int} = v\Psi(\mathbf{F}) \quad (3.12)$$

where v is the volume of the tetrahedron. This is so easy that literally every computer graphics article that uses some kind $\Psi(\mathbf{F})$ -based continuum mechanics measure uses the formulation I present in Sec. 7.1.

Can we finally get the formula for \mathbf{K} ? Bear with me for a couple more loops, and focus on a new physical concept instead: the *internal elastic force* incurred by a given deformation is calculated in the same manner as the bowling ball – vacuum chamber example in Eq. 3.9:

$$\mathbf{f}_{int} = -\frac{\partial\Pi_{int}}{\partial\mathbf{x}} = -v\frac{\partial\Psi}{\partial\mathbf{x}} \quad (3.13)$$

This is the force that pushes back on you when you squash a tetrahedron! This is a set of nonlinear equations, and we are going to '*solve*' them the way such a set of equations should be '*solved*': with the Newton-Raphson Iterative Scheme.

3.2.3 The Newton-Raphson Iterative Scheme

The Newton-Raphson method is an iterative scheme, which produces successively better *approximation* of some a real-valued function's roots. This method is not exclusive for the finite element method; it's a general method to approximate *any* real valued function.

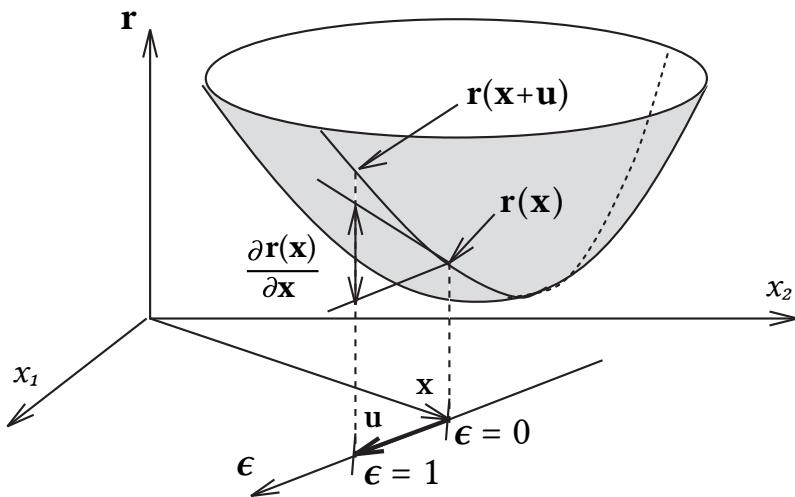


Figure 3.6: The Newton-Raphson scheme.

3. Fundamentals of FEM

Consider the solution of a set of nonlinear algebraic equations:

$$\mathbf{r}(\mathbf{x}) = \mathbf{0} \quad (3.14)$$

where, for example, for a simple case with two equations and two unknowns

$$\mathbf{r}(\mathbf{x}) = \begin{bmatrix} r_1(x_1, x_2) \\ r_2(x_1, x_2) \end{bmatrix} \quad (3.15)$$

In the Newton–Raphson iterative process, we have some solution estimate \mathbf{x}_k at iteration k . The value of the next iteration $\mathbf{x}_{k+1} = \mathbf{x}_k + \mathbf{u}$ is obtained in terms of an increment \mathbf{u} by establishing the linear approximation

$$\mathbf{r}(\mathbf{x}_{k+1}) \approx \mathbf{r}(\mathbf{x}_k) + \frac{\partial \mathbf{r}(\mathbf{x}_k)}{\partial \mathbf{x}} = 0 \quad (3.16)$$

The directional derivative $\frac{\partial \mathbf{r}(\mathbf{x}_k)}{\partial \mathbf{x}}$ is evaluated with the help of the chain rule as

$$\begin{aligned} \frac{\partial \mathbf{r}(\mathbf{x}_k)}{\partial \mathbf{x}} &= \frac{d}{d\epsilon} \Big|_{\epsilon=0} \mathbf{r}(\mathbf{x}_k + \epsilon \mathbf{u}) \\ &= \frac{d}{d\epsilon} \Big|_{\epsilon=0} \begin{bmatrix} r_1(x_1 + \epsilon u_1, x_2 + \epsilon u_2) \\ r_2(x_1 + \epsilon u_1, x_2 + \epsilon u_2) \end{bmatrix} \\ &= \hat{\mathbf{K}} \mathbf{u} \end{aligned} \quad (3.17)$$

You might have noticed, that for a given \mathbf{x} and \mathbf{u} , $\mathbf{r}(\mathbf{x}_{k+1})$ is a function of the parameter ϵ and we just derived the first-order Taylor series expansion about $\epsilon = 0$. Cool!

However, there's a much more important thing about Eq. 3.17. Matrix $\hat{\mathbf{K}}$ is the *tangent matrix* of the system:

$$\hat{\mathbf{K}} = \frac{\partial \mathbf{r}}{\partial \mathbf{x}} \quad (3.18)$$

If we substitute Eq. 3.17 for the directional derivative into Equation Eq. 3.16, we obtain a linear set of equations for \mathbf{u} to be solved at each Newton–Raphson iteration as

$$\hat{\mathbf{K}} \mathbf{u} = -\mathbf{r} \quad \mathbf{x}_{k+1} = \mathbf{x}_k + \mathbf{u} \quad (3.19)$$

See Fig. 3.6 for a nifty visualization!

If we plug in the principle of total potential energy into 3.19, we will finally, *FINALLY* get our hands on this precious \mathbf{K} . However $\mathbf{r} \neq \Pi_{int} + \Pi_{ext}$, but

$$\mathbf{r} = \frac{\partial \Pi_{int}}{\partial \mathbf{x}} + \frac{\partial \Pi_{ext}}{\partial \mathbf{x}} = \mathbf{f}_{int} - \mathbf{f}_{ext} \quad (3.20)$$

It's because the derivative is the one that should be zero – following Eq. 3.8.

We haven't really talked about Π_{ext} much, what's the deal with that? This is because if you use exclusively nodal forces – forces explicitly only acting on the nodes – in your simulator, Π_{ext} burns away during differentiation, so you don't need to worry about it, just add it to the final $\mathbf{K}\mathbf{u} = \mathbf{f}$ system and you're done. You can definitely go fancy and define some more exotic external energies, like external pressure components as in [6] Example 9.6; however, they might contribute to your tangent matrix as they might *not* burn away with differentiation.

Consequently, plugging in $\mathbf{r} = \mathbf{f}_{int} + \mathbf{f}_{ext}$ into Eq. 3.19 we arrive at our final set of equations: when only nodal forces considered, the *tangent stiffness matrix* \mathbf{K} is defined as

$$\boxed{\mathbf{K} = -v \frac{\partial \mathbf{f}}{\partial \mathbf{x}}} \quad (3.21)$$

where I dropped the *int* subscript for brevity. Many texts do that. There is an alternative form,

$$\boxed{\mathbf{K} = -v \frac{\partial^2 \Psi}{\partial \mathbf{x}^2}} \quad (3.22)$$

which is at least as important as the $\frac{\partial \mathbf{f}}{\partial \mathbf{x}}$ form. It expresses that while the force is the energy *Jacobian*, the tangent stiffness matrix is the energy *Hessian*. In this text I use all three terms.

The Newton-Raphson iteration step is then defined as

$$\boxed{\begin{aligned} \mathbf{K}\mathbf{u} &= -\mathbf{f}_{int} + \mathbf{f}_{ext} \\ \mathbf{x}_{k+1} &= \mathbf{x}_k + \mathbf{u} \end{aligned}} \quad (3.23)$$

where again,

$$\boxed{\mathbf{f}_{int} = -v \frac{\partial \Psi}{\partial \mathbf{x}}} \quad (3.24)$$

so you can pretty much write this slightly more obtuse form as well:

$$\boxed{-v \frac{\partial^2 \Psi}{\partial \mathbf{x}^2} \mathbf{u} = v \frac{\partial \Psi}{\partial \mathbf{x}} + \mathbf{f}_{ext}} \quad (3.25)$$

We just defined the *quasistatic formulation* of the nonlinear finite element procedure. I will tell you the next section, why it's called the quasistatic formulation, but let me calm you down a bit if you've got a bit afraid of these mysterious Hessians and Jacobians.

There are surely a lot of here to digest, but believe me, everything will become so clean and pretty, you will just die to program your first simulator after you finish reading my thesis. The thing is that you *really* just have to take the derivative; that's it! There is of course a lot of evil professor trick lurking around here, but all of them are going to be addressed in my favourite chapter, Chapter 5.

3.3 Dynamics Simulation

Believe it or not, we pretty much defined the key equation ($\mathbf{Ku} = -\mathbf{f}_{int} + \mathbf{f}_{ext}$) for a nonlinear finite element solver. For the sake of your own good, do not look up how much does such a software cost on the market. You might think that we are done, we just acquired the nice jello effect, show me the C++, please.

Well we definitely have the hardest part defined. However, a key piece is missing: *inertia!* We can integrate through time, but this equilibrium only remains valid if the loads are applied *slowly*. If in actuality the loads are *not* applied slowly, but they're applied rapidly, *inertia forces* need to be considered; i.e., a truly *dynamic problem* needs to be solved.

3.3.1 Equilibrium of a Dynamic Process

According to '*Newton's Second Law*', we can calculate the inertial force of a particle from $\mathbf{f} = m\mathbf{a}$, where \mathbf{a} is the acceleration vector of the particle. We usually introduce this idea to the finite element method by constructing a mass matrix \mathbf{M} for each element. Then, the inertial forces are calculated as $\mathbf{M}\ddot{\mathbf{u}}$ where $\ddot{\mathbf{u}} = \mathbf{a}$ lists the nodal point accelerations. The element mass matrices are accumulated in the same manner as the element stiffness matrices in the global mass matrix; and the equilibrium equations become

$$\mathbf{M}\ddot{\mathbf{u}} + \mathbf{Ku} = \mathbf{f}_{ext} \quad (3.26)$$

The element mass matrix is usually computed by lumping the element's physical mass m to the diagonal, such that

$$\mathbf{M} = m \cdot \mathbf{I}_{n \times n} \quad (3.27)$$

where n is the number of the element's degrees of freedom.

In a real, dynamic system, it is observed that some energy is dissipated during the motion. This behavior is usually taken account of by introducing velocity-dependent damping forces. In this way the equilibrium equations are further expanded to

$$\mathbf{M}\ddot{\mathbf{u}} + \mathbf{C}\dot{\mathbf{u}} + \mathbf{Ku} = \mathbf{f}_{ext} \quad (3.28)$$

where \mathbf{C} is the damping matrix of the structure.

In practice it is difficult, if not impossible, to determine for general finite element assemblages the element damping parameters. For this reason, the matrix \mathbf{C} is in general not assembled from element damping matrices but is constructed using the mass matrix and stiffness matrix of the complete element assemblage together, such that

$$\mathbf{M}\ddot{\mathbf{u}} + (\alpha\mathbf{M} + \beta\mathbf{K})\dot{\mathbf{u}} + \mathbf{K}\mathbf{u} = \mathbf{f}_{\text{ext}} \quad (3.29)$$

This kind of damping is called *Rayleigh damping* in the literature; and α and β are the damping parameters.

3.3.2 Time Integration

Mathematically, Eq. 3.29 represents a system of linear differential equations of second order and, in principle, the solution to the equations can be obtained by standard procedures for the solution of differential equations with constant coefficients. However, this is just stupid, and not even feasible when the order of the matrices is large.

We have to take specific advantage of the special characteristics of the coefficient matrices \mathbf{K} , \mathbf{C} , and \mathbf{M} ! Such methods are called *time stepping* or *time integration* schemes, and there are two, distinct groups of them: *explicit* and *implicit* methods.

Explicit Time Integration

Explicit methods calculate the state of a system at a later time from the state of the system at the current time. For example, take a look at the explicit *forward Euler* method:

$$\begin{pmatrix} \mathbf{u} \\ \dot{\mathbf{u}} \end{pmatrix} = h \begin{pmatrix} \dot{\mathbf{x}}_0 \\ \mathbf{M}^{-1}\mathbf{f}_0 \end{pmatrix} \quad (3.30)$$

Here $\dot{\mathbf{x}}_0$ is the velocity vector of the current timestep, $\mathbf{f}_0 = \mathbf{f}(\mathbf{x}_0, \dot{\mathbf{x}}_0)$, and h is the stepsize.

As you can see, the stiffness matrix, \mathbf{K} is *not* featured in this equation. This makes the implementation of such methods much-much easier than implicit methods, which do require the calculation of \mathbf{K} .

However, there is a big catch! Explicit methods require very small timesteps to stay stable – or in other words not to explode. Sometimes timesteps can get so small the simulation doesn't even move anywhere. Such big timesteps are just not feasible in an interactive framework, so most of the time implicit methods are used, which allow us to use much bigger timesteps.

Implicit Integration

Implicit methods find a solution by solving an equation involving both the current state of the system and the later one as well. They were popularized by Baraff and Witkin by their 98 paper on cloth simulation. [2]

They used the implicit *backward* Euler method. Generally, in implicit time integration, \mathbf{u} and $\dot{\mathbf{u}}$ are approximated as

$$\begin{pmatrix} \mathbf{u} \\ \dot{\mathbf{u}} \end{pmatrix} = h \begin{pmatrix} \dot{\mathbf{x}}_0 + \dot{\mathbf{u}} \\ \mathbf{M}^{-1}\mathbf{f}(\mathbf{x}_0 + \mathbf{u}, \dot{\mathbf{x}}_0 + \dot{\mathbf{u}}) \end{pmatrix} \quad (3.31)$$

The forward method requires only an evaluation of the function \mathbf{f} , while the backward method requires that we solve for values of \mathbf{u} and $\dot{\mathbf{u}}$ that satisfy equation Eq. 3.31. Let's do a first order Taylor series expansion on \mathbf{f} , which yields

$$\mathbf{f}(\mathbf{x}_0 + \mathbf{u}, \dot{\mathbf{x}}_0 + \dot{\mathbf{u}}) = \mathbf{f}_0 + \frac{\partial \mathbf{f}}{\partial \mathbf{x}} \mathbf{u} + \frac{\partial \mathbf{f}}{\partial \dot{\mathbf{u}}} \dot{\mathbf{u}} \quad (3.32)$$

Plugging this approximation back to Eq. 3.31 results

$$\begin{pmatrix} \mathbf{u} \\ \dot{\mathbf{u}} \end{pmatrix} = h \begin{pmatrix} \dot{\mathbf{x}}_0 + \dot{\mathbf{u}} \\ \mathbf{M}^{-1}(\mathbf{f}_0 + \frac{\partial \mathbf{f}}{\partial \mathbf{x}} \mathbf{u} + \frac{\partial \mathbf{f}}{\partial \dot{\mathbf{u}}} \dot{\mathbf{u}}) \end{pmatrix} \quad (3.33)$$

Since $\mathbf{u} = h(\dot{\mathbf{x}} + \dot{\mathbf{u}})$ we have

$$\dot{\mathbf{u}} = h\mathbf{M}^{-1} \left[\mathbf{f}_0 + \frac{\partial \mathbf{f}}{\partial \mathbf{x}} h(\dot{\mathbf{x}} + \dot{\mathbf{u}}) + \frac{\partial \mathbf{f}}{\partial \dot{\mathbf{x}}} \dot{\mathbf{u}} \right] \quad (3.34)$$

which can be conveniently regrouped to its final form:

$$\left[\mathbf{M} - h \frac{\partial \mathbf{f}}{\partial \dot{\mathbf{x}}} - h^2 \frac{\partial \mathbf{f}}{\partial \mathbf{x}} \right] \dot{\mathbf{u}} = h\mathbf{f} + h^2 \frac{\partial \mathbf{f}}{\partial \mathbf{x}} \dot{\mathbf{x}}$$

(3.35)

Or, you can use the version with \mathbf{C} and \mathbf{K} , instead of $\frac{\partial \mathbf{f}}{\partial \dot{\mathbf{x}}}$ and $\frac{\partial \mathbf{f}}{\partial \mathbf{x}}$ respectively as

$$[\mathbf{M} - h\mathbf{C} - h^2\mathbf{K}] \dot{\mathbf{u}} = h\mathbf{f} + h^2\mathbf{K}\dot{\mathbf{x}} \quad (3.36)$$

alough I think it's much easier to pick up girls in a bar with a partial derivative.

3.4 FEM: Beyond Deformation Simulation

Some interesting stuff for the end: FEM is very popular in geometry processing. Let's look at some examples!

3.4.1 Surface Modelling

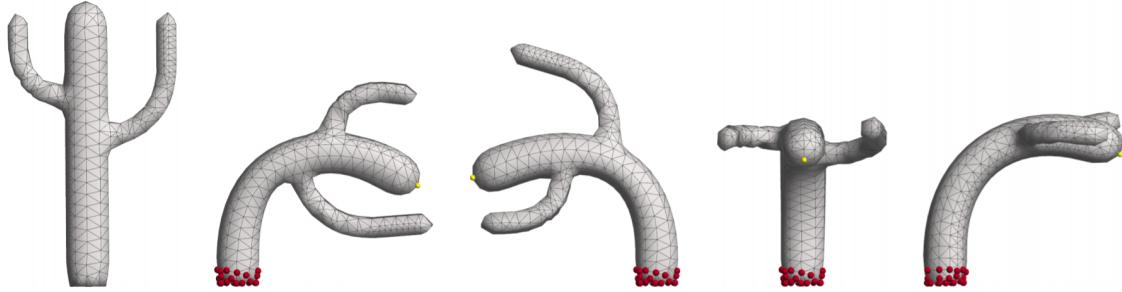


Figure 3.7: As-Rigid-As-Possible surface modelling. [27]

The first example is a pretty straightforward one. Imagine the following scenario: you are a 3D designer, and have some arbitrary mesh like leftmost one on Fig. 3.7. If you move e.g. the yellow point around, while the red ones should remain fixed, you probably expect *intuitively*, that the mesh's other vertices are move around the manner it's shown on Fig. 3.7.

How do you formulate this? Of course, with FEM! This problem is a regular FEM problem, but instead of telling the system what are the loads, we tell it where should a set of vertices should be – there is a neat trick for formulating this as well, presented in Sec. 7.3. The As-Rigid-As-Possible (ARAP) surface modelling method [27] presents nothing but a nonlinear FEM based solver, however, they defined their own new strain energy function:

$$\Psi_{\text{ARAP}} = \frac{\mu}{2} \|\mathbf{F} - \mathbf{R}\|_F^2 \quad (3.37)$$

Here $\|\cdot\|_F^2$ is the Frobenius norm (Sec. 4.4.1) turning a matrix (tensor) into a scalar. \mathbf{F} is the deformation gradient of course, while \mathbf{R} is the rotational part of \mathbf{F} – arising from the polar decomposition of \mathbf{F} . (Sec. 4.1.4)

This energy function can actually be used in deformation simulation as well, and has some nice properties, which we are going to discuss in Chapter 5 and 6.

3.4.2 Geometry Optimization

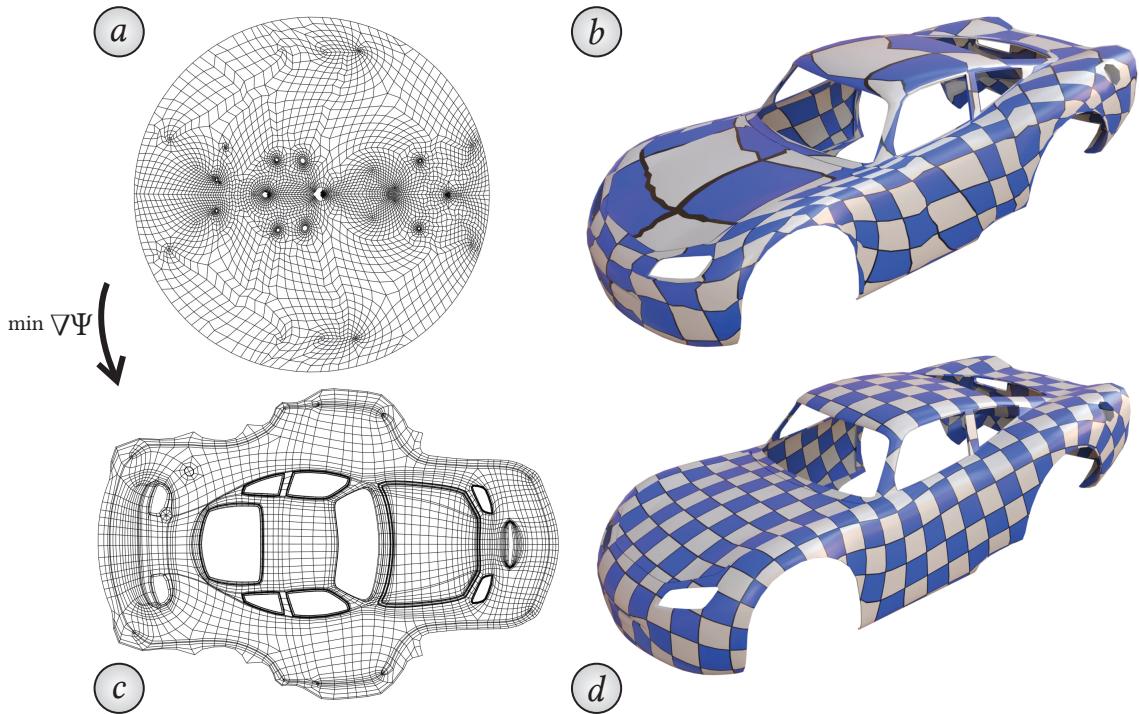


Figure 3.8: Optimizing surface parametrization: Initial surface parametrization (a) results poor utilization of the per-pixel color data (b) a.k.a. texture of the surface. Optimizing for consistent '*deformation energy*' Ψ achieves the desired paraetization automatically (c & d). [25]

This a bit more meta: meshes for FEM simulations are made with... FEM! Crazy right? Turns out that the finite element formulation is a pretty nice measure for the '*niceness*' of meshes: again, intuitively you expect a volumetric mesh's elements to be evenly distributed, each element having kind of the same, uniform edge length and volume.

The FEM-based approach for this problem does nothing but applies FEM: they start with a basic (random) Delaunay-triangulation, then consistently minimize a global energy function over the domain by relocating the vertices at each step. In the end they end up with a mesh where the energy is evenly distributed.

Aside its use for FEM meshes [1], they use it for surface parameterization as well (Fig. 3.8). This is a pretty big thing, because the reparametrization of surfaces is still done by hand, consuming hundreds of hours of expensive man-power.

Chapter 4

Fundamentals of Continuum Mechanics

Continuum mechanics is a branch of mechanics that deals with the mechanical behavior of materials modeled as a continuous mass rather than as discrete particles.

The initial objective of continuum mechanics is to provide a concise, coherent mathematical description of the deformation and motion that an elastic body has sustained. Then we can use these measures to quantify stuff about the deformation. For example, *Piola-Kirchhoff stress* \mathbf{P} or the *strain energy* Ψ are both computed from kinematic measures; both are essential in the finite element formulation.

If you really want to dig deep into continuum mechanics, I highly recommend the two classics on the subjects: the books by Bonet – Wood [6] and Holzapfel [9].

4.1 Kinematics

Let's begin our journey into continuum mechanics by exploring the set of kinematic measures we are going to use.

4.1.1 Motion

Fig. 4.1 shows the general motion of the deformable body. The body is an assemblage of particles labeled by the coordinates \mathbf{x} with respect to the Cartesian basis \mathbf{e}_i , at their initial positions at time $t = 0$. Generally, the current positions of these particles are located, at time $= t$, by the coordinates $\bar{\mathbf{x}}$ with respect to an alternative Cartesian basis $\bar{\mathbf{e}}_i$. In the remainder of this document, \mathbf{e}_i and $\bar{\mathbf{e}}_i$ will be taken to be coincident. Alternatively these *initial* and *current* configurations are referred as *material* or *Lagrangian* and *spatial* or *Eulerian* descriptions in the literature.

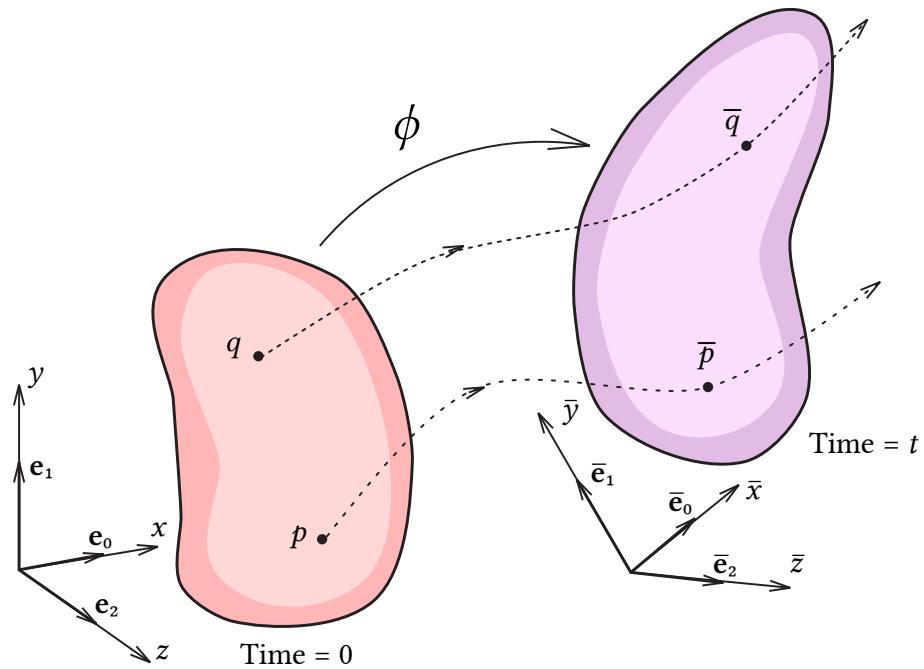


Figure 4.1: General motion of the body

With the particle positions precisely defined, we can describe the motion mathematically as a mapping ϕ between initial and current particle positions as

$$\bar{\mathbf{x}} = \phi(\mathbf{x}, t). \quad (4.1)$$

For a fixed value of t the above equations represent a mapping between the undeformed and deformed bodies. Additionally, for a fixed particle \mathbf{x} , Eq. 4.1 describes the motion or trajectory of this particle as a function of time.

It is important to note here, that I just departed from the regular notation in continuum mechanics: the initial (*material*) configuration is usually denoted by uppercase letters – e.g. \mathbf{X} and \mathbf{E}_I – while lowercase letters mark the current (*spatial*) configuration: e.g. \mathbf{x} and \mathbf{e}_i respectively. I ditched this notation and used the *overline* operator – $\bar{\mathbf{x}}$ and $\bar{\mathbf{e}}$ – to denote the current (*spatial*) configuration as seen above. It has a practical reason: in the subsequent chapters, significant amount of linear algebra and tensor trickery will be present, so it's easier to follow the train of thought with lowercase letters denoting vectors (\mathbf{a}) and uppercase ones matrices (\mathbf{A}).

4.1.2 Deformation Gradient

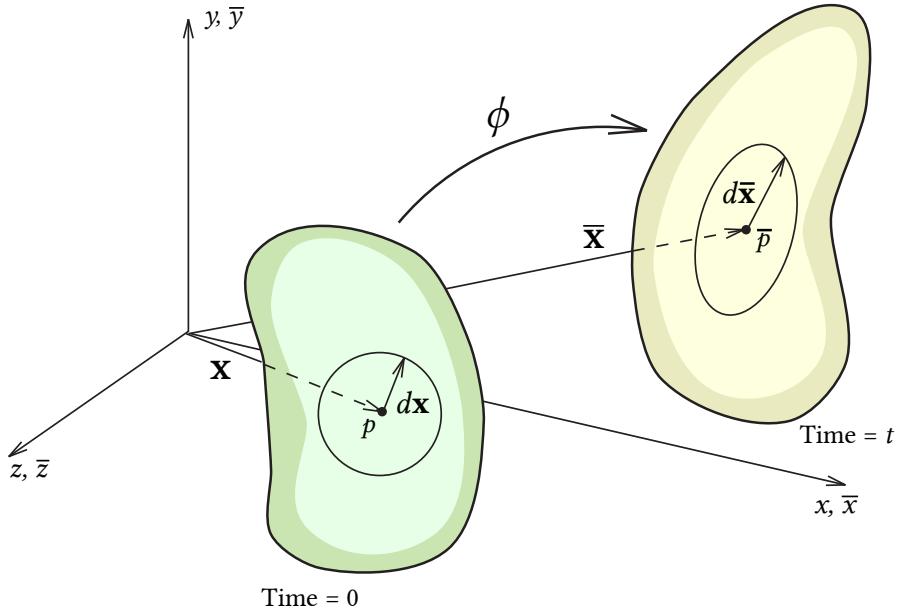


Figure 4.2: The Deformation Gradient

While ϕ characterizes the general motion of the body, there is a measure, which precisely quantifies the deformation – that is, all the *rotation* and *scaling* – at each quadrature point: it's the deformation gradient, denoted by F , which I briefly introduced it in Sec. 3.2.2. It is among the important quantities of deformable simulation, we calculate *everything* from F : it is involved in all equations *relating quantities before deformation to corresponding quantities after (or during) deformation*. The deformation gradient tensor enables the relative *spatial* position of a particle after deformation to be described in terms of its relative *material* position before deformation.

Let's derive it! In order to do so, consider the infinitesimal material line elements dx and $d\bar{x}$ originating from material particles p and \bar{p} respectively. After deformation, the material particles p and dx in the initial configuration can be described in the current configuration as the mapping

$$\bar{x} + d\bar{x} = \phi(x + dx, t) \quad (4.2)$$

This can be further expanded to

$$\bar{x} + d\bar{x} = \phi(x, t) + \frac{\partial \phi(x, t)}{\partial x} dx \quad (4.3)$$

4. Fundamentals of Continuum Mechanics

As $\bar{\mathbf{x}} = \phi(\mathbf{x}, t)$, from Eq. 4.3 we have

$$d\bar{\mathbf{x}} = \frac{\partial\phi(\mathbf{x}, t)}{\partial\mathbf{x}} d\mathbf{x} \quad (4.4)$$

The thing on the right hand side is the deformation gradient:

$$\boxed{F = \frac{\partial\phi}{\partial\mathbf{x}}} \quad (4.5)$$

It's pretty much the derivative of the motion:

$$F = \begin{bmatrix} \partial\bar{x}/\partial x & \partial\bar{x}/\partial y & \partial\bar{x}/\partial z \\ \partial\bar{y}/\partial x & \partial\bar{y}/\partial y & \partial\bar{y}/\partial z \\ \partial\bar{z}/\partial x & \partial\bar{z}/\partial y & \partial\bar{z}/\partial z \end{bmatrix} \quad (4.6)$$

Alternatively you can write

$$d\bar{\mathbf{x}} = F d\mathbf{x}, \quad (4.7)$$

which I think shows pretty well how F transforms vectors in the initial configuration into vectors in the current configuration.

Very important property coming up: changing the configuration, i.e. translating our material particle by an arbitrary vector, t *does nothing to a deformation gradient*:

$$\bar{\mathbf{x}} = \phi(\mathbf{x}) = \mathbf{x} + \mathbf{t} \quad \rightarrow \quad F = \frac{\partial\phi(\mathbf{x})}{\partial\mathbf{x}} = I \quad (4.8)$$

\mathbf{t} just burned away! It is said that F is an *objective measure*. No matter how much *rigid-body* translation you apply to an element, it doesn't do anything with F . See [9] Chapter 5 for more information about *objectivity*.

Hopefully I was able to grasp the essence of the deformation gradient. Right now, it is not important *exactly* how you would compute the quantity – it has its own section. (Sec. 7.1) So for the time being, just try to *feel* what F means.

4.1.3 Strain

To simplify analyses, the *concept* of strain tensors are introduced. It is nothing but another geometric representation to measure deformation. The right Cauchy-Green tensor C is defined as

$$C = F^T F \quad (4.9)$$

while the material Green Strain E is calculated from

$$E = \frac{1}{2}(C - I) = \frac{1}{2}(F^T F - I) \quad (4.10)$$

4.1.4 Polar Decomposition

The deformation gradient \mathbf{F} transforms any vector in the material coordinate system to one in the spatial coordinate system. We also stated that rigid-body translations are *not* included in \mathbf{F} , so it only encodes *all* the *rotation* and *stretching* the material particle has sustained. Now the thing is that you can actually factor out the separate rotational \mathbf{R} and stretching component \mathbf{S} from \mathbf{F} , while

$$\mathbf{F} = \mathbf{RS} \quad (4.11)$$

Note that in the continuum mechanics literature, \mathbf{S} is usually denoted by \mathbf{U} . I used \mathbf{S} instead (following [15]), because \mathbf{U} is reserved for the result of the singular value decomposition.

Remember the As-Rigid-As-Possible (ARAP) energy from Sec. 3.4.2? It was

$$\Psi_{\text{ARAP}} = \frac{\mu}{2} \|\mathbf{F} - \mathbf{R}\|_F^2 \quad (4.12)$$

where \mathbf{R} is of course the rotational part of \mathbf{F} . We are going to use this energy a lot, so we need an efficient approach to factorize \mathbf{R} from \mathbf{F} .

Compute Polar Decomposition from SVD

One way to factorize \mathbf{F} into \mathbf{R} and \mathbf{S} is by taking the *singular value decomposition (SVD)* of it. SVD is a factorization of some matrix \mathbf{A} to its constituent parts, such that

$$\mathbf{A} = \mathbf{U}\Sigma\mathbf{V}^T \quad (4.13)$$

\mathbf{U} and \mathbf{V} are the left- and right singular vectors, representing rotations, while Σ is a scaling by the singular values. See Fig. 4.3 for reference.

Cool thing about this SVD is that if you apply on \mathbf{F} ,

$$\mathbf{F} = \hat{\mathbf{U}}\hat{\Sigma}\hat{\mathbf{V}}^T, \quad (4.14)$$

is that you can pretty easily get \mathbf{S} and \mathbf{R} as

$$\hat{\mathbf{R}} = \hat{\mathbf{U}}\hat{\mathbf{V}}^T \quad (4.15)$$

$$\hat{\mathbf{S}} = \hat{\mathbf{V}}\hat{\Sigma}\hat{\mathbf{V}}^T \quad (4.16)$$

Okay but we need \mathbf{S} and \mathbf{R} , not $\hat{\mathbf{S}}$ and $\hat{\mathbf{R}}$, what's up with all that hatting?

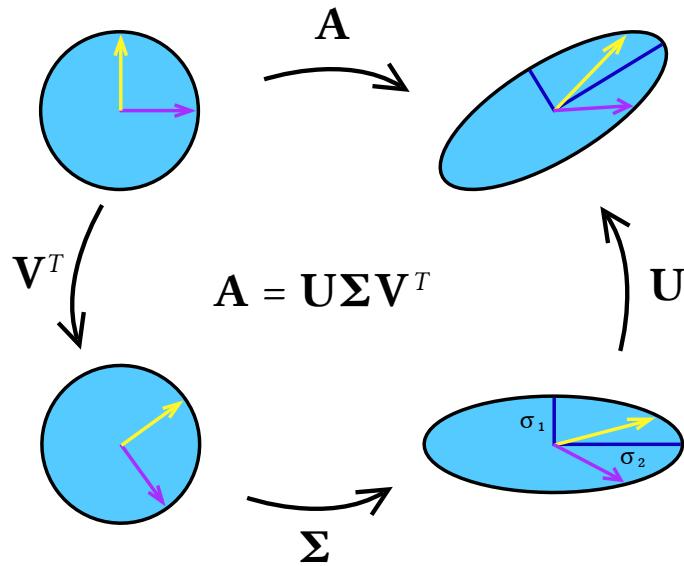


Figure 4.3: Singular value decomposition of some matrix \mathbf{A} .

There's a small caveat as – following Teran et al. [12] – we need a *rotationally variant* SVD. The basic SVD factorization only guarantees that \mathbf{R} is unitary matrix – that is $\mathbf{R}^T\mathbf{R} = \mathbf{I}$. This means that there could be a reflection lurking somewhere in \mathbf{R} , whereas we want \mathbf{R} to be a pure rotation. In our case, if a reflection has to lurk somewhere, we would prefer that it do so in \mathbf{S} .

There's a pretty neat way to pull this off by Sorkine-Hornung and Rabinovich. [28] It is presented in [15], Appendix F. Of course, you will end up with the unhatted values:

$$\mathbf{R} = \mathbf{U}\mathbf{V}^T \quad (4.17)$$

$$\mathbf{S} = \mathbf{V}\Sigma\mathbf{V}^T \quad (4.18)$$

4.1.5 Volume change

Another very important kinematic measure is the volume change of an element, denoted by J , which can be easily computed from

$$J = \det \mathbf{F} \quad (4.19)$$

4.2 The Concept of Stress

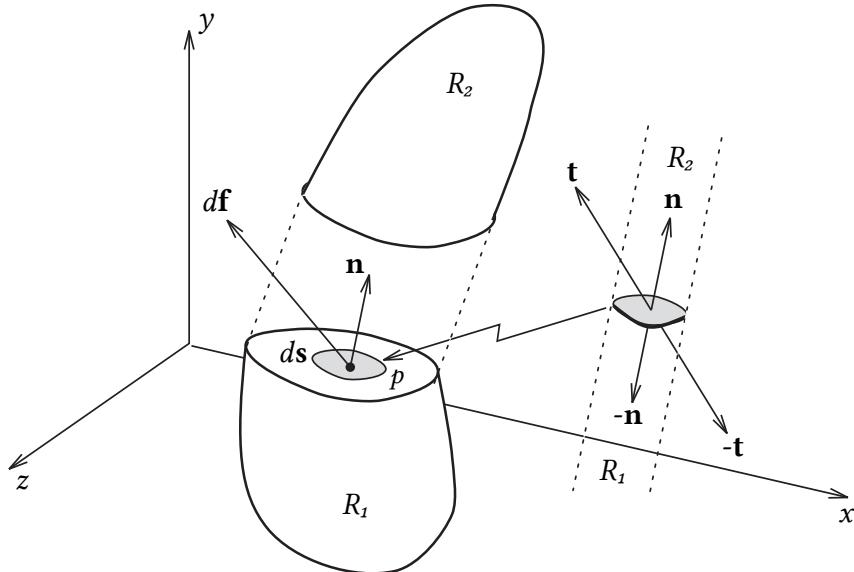


Figure 4.4: The concept of stress in the *material* coordinate system.

In the previous section some kinematical aspects of the motion and deformation of the continuum body were discussed. Motion and deformation give rise to interactions between the neighboring particles in the interior part of the body. One consequence of these interactions is stress, which has physical dimension *force per unit area*.

(*Yet again*) consider general a deformable body under loading as in Fig. 4.4. Let the body now be cut by a plane with surface normal \mathbf{n} at any given point $\mathbf{x} \in \Omega$, resulting two regions, R_1 and R_2 . In order to develop the *concept of stress*, we should study the action of forces, applied by one region (R_1) to the remaining part of the body (R_2).

For this purpose, let's examine some infinitesimal area ds on the cutting-plane. Since forces are transmitted across the internal plane surface, we can define a *resultant force* acting on the surface element as df . According to Fig. 4.4 for every surface element

$$df = \bar{\mathbf{t}} d\bar{s} = \bar{\mathbf{t}} ds \quad (4.20)$$

where

$$\bar{\mathbf{t}} = \bar{\mathbf{t}}(\bar{\mathbf{x}}, t, \bar{\mathbf{n}}) \quad (4.21)$$

represent the *Cauchy traction vector* – force measured per unit surface area defined in the *current configuration* – exerted on $d\bar{s}$ with outward normal $\bar{\mathbf{n}}$. On the other hand,

$$\mathbf{t} = \mathbf{t}(\mathbf{x}, t, \mathbf{n}) \quad (4.22)$$

represent the *first Piola-Kirchhoff traction vector* – force measured per unit surface area defined in the *reference configuration*. Again, both traction vectors measure how much one side is pushing on the other side in case of some specific loading condition.

Cauchy's stress theorem states, that there exist unique second-order tensor fields σ and \mathbf{P} so that

$$\begin{aligned}\bar{\mathbf{t}}(\bar{\mathbf{x}}, t, \bar{\mathbf{n}}) &= \sigma(\bar{\mathbf{x}}, t)\bar{\mathbf{n}} \\ \mathbf{t}(\mathbf{x}, t, \mathbf{n}) &= \mathbf{P}(\mathbf{x}, t)\mathbf{n}\end{aligned}\quad (4.23)$$

where σ denotes the *spatial tensor field* called *Cauchy stress tensor*, while \mathbf{P} characterizes a tensor field called *first Piola-Kirchhoff stress tensor* in the *reference configuration*.

Relation 4.23 is one of the most important axiom of continuum mechanics. It basically states that if a traction vector such as \mathbf{t} or $\bar{\mathbf{t}}$ depend on the outward unit normals \mathbf{n} or $\bar{\mathbf{n}}$, then they must be *linear in \mathbf{t} or $\bar{\mathbf{t}}$* , respectively.

Furthermore, an immediate consequence of Eq. 4.23 is that

$$\mathbf{t}(\mathbf{x}, t, \mathbf{n}) = -\mathbf{t}(\mathbf{x}, t, -\mathbf{n}) \quad (4.24)$$

for all unit vectors \mathbf{n} . This is known as *Newton's third law of action and reaction* and it applies in the reference configuration as well (here omitted).

4.3 Balance Laws

We've got a pretty concise descriptions of motion, and introduced the concept of stress, which are two important building blocks, when we try to model some '*physically based*' deformable phenomena. However, there are two key pieces missing. Actually we already talked about one of them: in Sec. 3.2.2, we briefly introduced the concept of total potential energy, which allowed us to completely characterize response of the physical phenomena of a strain energy-based deformable model, i.e.

$$\Pi_{int} + \Pi_{ext} = \text{const} \quad (4.25)$$

$$\Pi_{int} = \int_{\Omega_0} \Psi(\mathbf{F}) dV \quad (4.26)$$

For our purposes, we don't need to worry about much where does this idea is derived from. But if you are interested, I highly recommend you going through Holzapfel's derivation in [9], Chapter 4. It can be derived precisely from the conservation of mass, and the balance of linear and angular momentum (a.k.a. Newton's first and second law).

4.4 The Hyperelastic Constitutive Model

It's only a single Lego-block that is missing now: what is that energy function Ψ ? The thing we haven't talked about yet is the material the body in discussion made of. All the things we described here, the kinematics, stresses, balance principle, holds for any continuum body. The energy function Ψ is what distinguishes, if we talk about steel or a piece of well chewed gum.

In this text, we are going to use a special energy function, exploiting that we are considering a conservative system: we are going to use the hyperelastic material – or more 'scientifically' *constitutive* – model.

The hyperelastic constitutive model is a phenomenological approach modelling perfectly elastic material. In this context perfectly elastic means that no matter how badly I abuse a piece of rubber, after the loads are removed, the body recovers to its original state. This means that we can characterize hyperelastic models exclusively with the deformation gradient, so the energy function, Ψ , becomes the strain energy function, $\Psi(\mathbf{F})$.

$\Psi(\mathbf{F})$ returns the deformation score for a specific \mathbf{F} , which is a scalar, making $\Psi(\mathbf{F})$ a scalar valued tensor function. Then as we have already seen it in Eq. 3.23, we need to take the Jacobian and Hessian of it in order to timestep our Newton-solver. It surely doesn't sound as fun as it's going to be. Chapter 5 will be all about that.

If you want to be a constitutive god, I have to point you again to Holzapfel's masterpiece [9], but this time to Chapter 6.

There are two very important properties of the strain energy function in case of hyperelastic material models, which we are going to exploit in the following chapters.

First, strain energy vanishes in the reference configuration, or in other words, if there is no motion: $\mathbf{F} = \mathbf{I}$. That is

$$\Psi(\mathbf{I}) = 0 \tag{4.27}$$

Second, if $\mathbf{F} = \mathbf{R}$, that is, the motion is some *pure rigid-body rotation*

$$\Psi(\mathbf{R}) = 0 \tag{4.28}$$

We will elaborate on this a lot in Sec. 5.5.

4.4.1 What Does an Energy Function Look Like?

$\Psi(\mathbf{F})$ is a scalar valued tensor function. You plug in a tensor, \mathbf{F} , and you get a scalar back. But how do you boil down a matrix to a number exactly?

One way to do that is to define energy functions as the squared Frobenius norm. You might not have heard about it, so let me just define it for you.

The Squared Frobenius Norm

Consider a matrix

$$\mathbf{A} = \begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{bmatrix} \quad (4.29)$$

then the squared Frobenius norm of matrix \mathbf{A} , denoted by $\|\mathbf{A}\|_F^2$ is

$$\|\mathbf{A}\|_F^2 = \sum_{i=0}^2 \sum_{j=0}^2 a_{ij}^2$$

(4.30)

So it's just summing up all the squared entries of matrix \mathbf{A} .

Why would you do that? It feels a bit... arbitrary, isn't it? It surely felt for me, when I first came across these norms. Doesn't anybody care which row and column these entries came from? Surely their ordering in the matrix should count for something?

Well the point is that first: there isn't much else you can do when you want to distill the '*essence*' of some vector or a matrix into a single number. You can take the square root of this, or if it's a matrix, maybe you can calculate the trace, but none of these do something fundamentally differently than the squared Frobenius norm.

Second of all – and more importantly – it gets the job done. And I'm – hopefully going to be soon – an engineer so it's all I care about at this point. For me, it actually got the job done so well, that I started using norms a lot. E.g. if you program any linear algebra, it's a pretty neat way to '*debug*' huge matrices and vectors.

An Energy Function

We've got a nice way to boil down a matrix to a scalar. Let's just use it and have a look at an energy function. You get the stretching part of the Saint-Venant Kirchhoff energy if you take the squared Frobenius norm of the Green strain from Eq. 4.10:

$$\Psi_{\text{StVK,stretch}} = \|\mathbf{E}\|_F^2 = \frac{1}{4} \|\mathbf{F}^T \mathbf{F} - \mathbf{I}\|_F^2 \quad (4.31)$$

In the next chapter we are going to use this form a lot. Buckle up, because we are about to take a deep dive in the inner workings of energy functions, and its Jacobians and Hessians.

Chapter 5

Jacobians and Hessians

Either you go down the quasistatic way

$$\mathbf{K}\mathbf{u} = \mathbf{f}_{int} - \mathbf{f}_{ext} \quad (5.1)$$

$$\mathbf{f}_{int} = -v \frac{\partial \Psi}{\partial \mathbf{x}} \quad \mathbf{K} = -v \frac{\partial \mathbf{f}}{\partial \mathbf{x}} = -v \frac{\partial^2 \Psi}{\partial \mathbf{x}^2} \quad (5.2)$$

or the backward Euler way

$$\left[\mathbf{M} - h \frac{\partial \mathbf{f}}{\partial \dot{\mathbf{x}}} - h^2 \frac{\partial \mathbf{f}}{\partial \mathbf{x}} \right] \dot{\mathbf{u}} = h \mathbf{f} + h^2 \frac{\partial \mathbf{f}}{\partial \mathbf{x}} \quad (5.3)$$

you need to compute both the *Jacobian* $\frac{\partial \Psi}{\partial \mathbf{x}}$ and the *Hessian* $\frac{\partial^2 \Psi}{\partial \mathbf{x}^2}$ of the strain energy function $\Psi(\mathbf{F})$. But how would you do that? $\Psi(\mathbf{F})$ is a scalar valued tensor function; how do you take its derivative in terms of \mathbf{x} , which is a vector? And what is a tensor anyways? Don't worry, we are going to go through the $\frac{\partial \Psi}{\partial \mathbf{x}}$ and $\frac{\partial^2 \Psi}{\partial \mathbf{x}^2}$ computation extensively, following Kim in [15].

Let's take one step at a time and try to compute \mathbf{f}_{int} ! First we need to stack the vertices of our element of choice into a big vector – e.g. for a 2D triangle this vector is

$$\mathbf{x} = \begin{bmatrix} \mathbf{x}_0 \\ \mathbf{x}_1 \\ \mathbf{x}_2 \end{bmatrix} = \begin{bmatrix} x_0 \\ y_0 \\ x_1 \\ y_1 \\ x_2 \\ y_2 \end{bmatrix}, \quad (5.4)$$

then take the gradient of it:

$$\mathbf{f} = -a \frac{\partial \Psi}{\partial \mathbf{x}} \quad (5.5)$$

Here a is the area of the original triangle at rest. As you can see in 2D we multiply with a instead of v .

To be able to deal with this problem, we need to apply the good ole' chain rule that separates $\frac{\partial \Psi}{\partial \mathbf{x}}$ into two components:

$$\frac{\partial \Psi}{\partial \mathbf{x}} = \frac{\partial \Psi}{\partial \mathbf{F}} : \frac{\partial \mathbf{F}}{\partial \mathbf{x}} \quad (5.6)$$

Not so much friendlier looking, huh? And there are several hidden gotchas as well, what we need to address here before we progress to calculate the force jacobians $\frac{\partial^2 \Psi}{\partial \mathbf{x}^2}$.

5.1 Higher-Order Tensor Manipulations

The first term in Eq. 5.6 is relatively easy. $\Psi \in \Re$ and $\mathbf{F} \in \Re^{3 \times 3}$, so the derivative is just a matrix:

$$\frac{\partial \Psi}{\partial \mathbf{F}} = \begin{bmatrix} \frac{\partial \Psi}{\partial f_{00}} & \frac{\partial \Psi}{\partial f_{10}} & \frac{\partial \Psi}{\partial f_{20}} \\ \frac{\partial \Psi}{\partial f_{01}} & \frac{\partial \Psi}{\partial f_{11}} & \frac{\partial \Psi}{\partial f_{21}} \\ \frac{\partial \Psi}{\partial f_{02}} & \frac{\partial \Psi}{\partial f_{12}} & \frac{\partial \Psi}{\partial f_{22}} \end{bmatrix} \quad (5.7)$$

Here f_{ij} are the scalar entries of \mathbf{F} . The result is clearly a 3×3 matrix, also known as a *2nd-order tensor*.

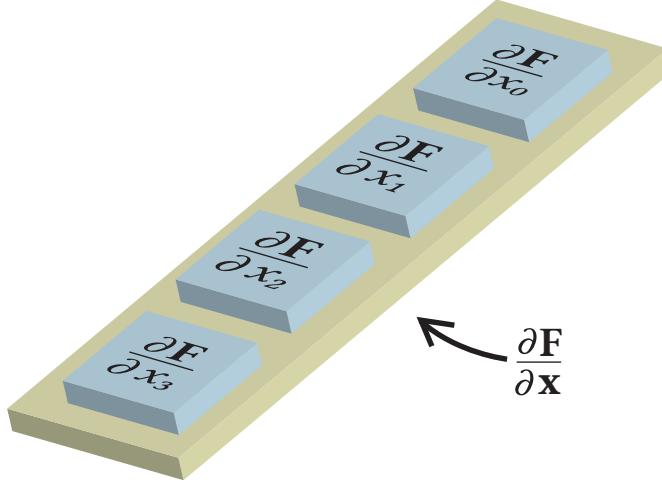
5.1.1 Mental Representation of Higher-Order Tensors

The confusion begins when we try to compute $\frac{\partial \mathbf{F}}{\partial \mathbf{x}}$. Since $\mathbf{F} \in \Re^{3 \times 3}$ is a matrix and $\mathbf{x} \in \Re^{12}$ (or \Re^6 in 2D). The first entry of $\frac{\partial \mathbf{F}}{\partial \mathbf{x}}$ would look like then

$$\frac{\partial \mathbf{F}}{\partial x_0} = \begin{bmatrix} \frac{\partial f_{00}}{\partial x_0} & \frac{\partial f_{10}}{\partial x_0} & \frac{\partial f_{20}}{\partial x_0} \\ \frac{\partial f_{01}}{\partial x_0} & \frac{\partial f_{11}}{\partial x_0} & \frac{\partial f_{21}}{\partial x_0} \\ \frac{\partial f_{02}}{\partial x_0} & \frac{\partial f_{12}}{\partial x_0} & \frac{\partial f_{22}}{\partial x_0} \end{bmatrix} \quad (5.8)$$

Already, the result is a matrix. Taking the derivates with respect to the next entries of \mathbf{x} , we will have a pile of matrices, 12 of them actually. There are many methods for dealing with such structures; e.g. you might come across the so called *Einstein* or *indical notation* during your continuum mechanics studies.

The dimension of the derivative $\frac{\partial \mathbf{F}}{\partial \mathbf{x}} \in \Re^{3 \times 3 \times 12}$, where the third dimension is the big giveaway that we're looking at a 3rd-order tensor. Many textbooks prefer to think about these as a cube of matrices, but for our purposes a *vector of matrices* is the most straightforward mental image – see Fig. 5.1 for visualisation.

Figure 5.1: Thinking about 3rd-order tensors as a vector of matrices.

That is, some 3rd-order tensor \mathbb{A} looks like

$$\mathbb{A} = \begin{bmatrix} a & c \\ b & d \\ e & g \\ f & h \\ i & k \\ j & l \end{bmatrix} = \begin{bmatrix} [\mathbf{A}] \\ [\mathbf{B}] \\ [\mathbf{C}] \end{bmatrix} \quad (5.9)$$

5.1.2 Multiplication With Higher-Order Tensors

After dealing with the mind image of higher-order tensors, all that's left to discuss from $\frac{\partial\Psi}{\partial\mathbf{F}} : \frac{\partial\mathbf{F}}{\partial\mathbf{x}}$ is the funky ':' sign. It denotes the *double-contraction* operation. *Contracting two matrices together* would be like

$$\mathbf{A} : \mathbf{B} = \begin{bmatrix} a_0 & a_2 \\ a_1 & a_3 \end{bmatrix} \begin{bmatrix} b_0 & b_2 \\ b_1 & b_3 \end{bmatrix} = a_0b_0 + a_1b_1 + a_2b_2 + a_3b_3 \quad (5.10)$$

Pretty easy! Just sum up the product of each entry (like $\sum a_i b_i$). However, $\frac{\partial\mathbf{F}}{\partial\mathbf{x}}$ is not a matrix, but a 3rd-order tensor. A 3rd vs. 2nd double-contraction is defined as

$$\mathbb{A} : \mathbf{B} = \begin{bmatrix} a_0 & a_2 \\ a_1 & a_3 \\ a_4 & a_6 \\ a_5 & a_7 \\ a_8 & a_{10} \\ a_9 & a_{11} \end{bmatrix} \begin{bmatrix} b_0 & b_2 \\ b_1 & b_3 \end{bmatrix} = \begin{bmatrix} a_0b_0 + a_1b_1 + a_2b_2 + a_3b_3 \\ a_4b_0 + a_5b_1 + a_6b_2 + a_7b_3 \\ a_8b_0 + a_9b_1 + a_{10}b_2 + a_{11}b_3 \end{bmatrix} \quad (5.11)$$

It's not that hard either! It is basically the same operation as with matrices, but we contract the 2nd-order one against each '*submatrix*' of the higher-order tensor. Stacking the results on top of each other yields a good ole' vector.

5.1.3 Vectorization

So far so good, but working with higher-order tensors in high-performance code becomes pretty cumbersome. However, there's a solution, which will make our life much simpler: the act of *vectorization* or simply *flattening* turns any tensor, be it 3rd-order, 4th-order, or 100th-order, back into the familiar 2nd-order matrix form. The vectorized representation was popularized by Pixar in [24].

We introduce the vectorization operator $\text{vec}(\cdot)$ to convert any matrix to a vector, and any higher-order tensor to a matrix. First up, this is how it converts a matrix to a vector:

$$\text{vec}(\mathbf{A}) = \text{vec} \left(\begin{bmatrix} a_0 & a_2 \\ a_1 & a_3 \end{bmatrix} \right) = \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix} \quad (5.12)$$

The important pattern to notice is that we *stacked the columns on top of each other*. The flattening convention of a 3rd-order tensor looks like

$$\text{vec}(\mathbb{A}) = \text{vec} \begin{bmatrix} [\mathbf{A}] \\ [\mathbf{B}] \\ [\mathbf{C}] \end{bmatrix} = \begin{bmatrix} \text{vec}(\mathbf{A}) & \text{vec}(\mathbf{B}) & \text{vec}(\mathbf{C}) \end{bmatrix} \quad (5.13)$$

which expands to

$$\text{vec} \begin{bmatrix} \begin{bmatrix} a_0 & a_2 \\ a_1 & a_3 \end{bmatrix} \\ \begin{bmatrix} a_4 & a_6 \\ a_5 & a_7 \end{bmatrix} \\ \begin{bmatrix} a_8 & a_{10} \\ a_9 & a_{11} \end{bmatrix} \end{bmatrix} = \begin{bmatrix} \text{vec} \begin{bmatrix} a_0 & a_2 \\ a_1 & a_3 \end{bmatrix} & \text{vec} \begin{bmatrix} a_4 & a_6 \\ a_5 & a_7 \end{bmatrix} & \text{vec} \begin{bmatrix} a_8 & a_{10} \\ a_9 & a_{11} \end{bmatrix} \end{bmatrix} = \begin{bmatrix} a_0 & a_4 & a_8 \\ a_1 & a_5 & a_9 \\ a_2 & a_6 & a_{10} \\ a_3 & a_7 & a_{11} \end{bmatrix} \quad (5.14)$$

The big advantage of using vectorized form of tensors is that we can replace the mighty double contraction operation with a conventional matrix multiply, as follows:

$$\mathbb{A} : \mathbf{B} = (\text{vec } \mathbb{A})^T \text{vec } \mathbf{B} \quad (5.15)$$

5.2 Computing Forces

With all the tensor stuff in place, we can finally compute some forces. Once again it's calculated from $\mathbf{f} = -v \frac{\partial \Psi}{\partial \mathbf{x}}$, and with the chain rule it becomes

$$\frac{\partial \Psi}{\partial \mathbf{x}} = \frac{\partial \Psi}{\partial \mathbf{F}} : \frac{\partial \mathbf{F}}{\partial \mathbf{x}} \quad (5.16)$$

The trick is that the 3rd-order tensor, $\frac{\partial \mathbf{F}}{\partial \mathbf{x}}$ has a static, simple, and energy-independent structure that stays the same through the simulation. It has to be derived and coded only once for each element type. For the time being, let's just assume we have this $\frac{\partial \mathbf{F}}{\partial \mathbf{x}}$ – as it's actually featured in Sec. 7.1.2.

The other term, $\frac{\partial \Psi}{\partial \mathbf{F}}$ is what we care about! Do you remember the *first Piola-Kirchhoff stress tensor*? I already introduced it in Sec. 4.2. It is denoted by \mathbf{P} , and believe it or not,

$$\frac{\partial \Psi(\mathbf{F})}{\partial \mathbf{F}} = \mathbf{P}(\mathbf{F}) \quad (5.17)$$

Isn't it wonderful? And the best thing is that if you want to use a new energy, all you need to do is derive a new \mathbf{P} a.k.a. $\frac{\partial \Psi}{\partial \mathbf{F}}$.

If you got both terms, you can go down the double contraction route as of Eq. 5.16, or use our fancy new vectorization method to compute things purely matrix way as well. The latter approach yields

$$\frac{\partial \Psi}{\partial \mathbf{x}} = \text{vec} \left(\frac{\partial \mathbf{F}}{\partial \mathbf{x}} \right)^T \text{vec} \left(\frac{\partial \Psi}{\partial \mathbf{F}} \right) \quad (5.18)$$

5.2.1 Examples

With all this knowledge, computing forces becomes pretty easy! To get the hang of Frobenius norms, let's take a look at some examples on calculating \mathbf{P} .

St. Venant-Kirchhoff

The St. Venant-Kirchhoff energy is defined as

$$\Psi_{\text{StVK}} = \mu \|\mathbf{E}\|_F^2 + \frac{\lambda}{2} (\text{tr} \mathbf{E})^2 \quad (5.19)$$

Here, the first member, $\|\mathbf{E}\|_F^2$ is the stretching term, while $(\text{tr} \mathbf{E})^2$ is volume preservation one. With the constants μ and λ you can tell the model how much relative stretching resistance vs. volume preservation you want.

5. Jacobians and Hessians

First lets deal with $\|\mathbf{E}\|_F^2$. Recall from Eq. 4.10 that Green Strain $\mathbf{E} = \frac{1}{2}(\mathbf{F}^T \mathbf{F} - \mathbf{I})$. Plug this in the energy function and expand the Frobenius norm using the identities $\|\mathbf{A} + \mathbf{B}\|_F^2 = \|\mathbf{A}\|_F^2 + \|\mathbf{B}\|_F^2 - 2\text{tr}\mathbf{A}^T \mathbf{B}$ and $\text{tr}(\mathbf{F}^T \mathbf{F}) = \|\mathbf{F}\|_F^2$ such that

$$\Psi_{\text{StVK,stretch}} = \frac{1}{4} \|\mathbf{F}^T \mathbf{F} - \mathbf{I}\|_F^2 = \frac{1}{4} \|\mathbf{F}^T \mathbf{F}\|_F^2 + \text{tr}\mathbf{I} - 2\|\mathbf{F}\|_F^2 \quad (5.20)$$

$\text{tr}\mathbf{I}$ burns away during differentiation, so the \mathbf{P} of the stretching term becomes

$$\begin{aligned} \mathbf{P}_{\text{StVK,stretch}} &= \frac{1}{4} (4\mathbf{F}\mathbf{F}^T - 4\mathbf{F}) \\ &= \mathbf{F}(\mathbf{F}^T - \mathbf{I}) \\ &= \mathbf{F}\mathbf{E} \end{aligned} \quad (5.21)$$

It's so easy I could even recall this during an exam! The complete StVK is not that hard either:

$$\mathbf{P}_{\text{StVK}} = \mu\mathbf{F}\mathbf{E} + \lambda(\text{tr}\mathbf{E})\mathbf{F} \quad (5.22)$$

As-Rigid-As-Possible

As-Rigid-As-Possible (ARAP) energy, presented in Sec. 3.4.2 can be expanded to

$$\begin{aligned} \Psi_{\text{ARAP}} &= \frac{\mu}{2} \|\mathbf{F} - \mathbf{R}\|_F^2 \\ &= \frac{\mu}{2} (\|\mathbf{F}\|_F^2 + \|\mathbf{R}\|_F^2 - 2\text{tr}(\mathbf{F}^T \mathbf{F})) \\ &= \frac{\mu}{2} (\|\mathbf{F}\|_F^2 + \|\mathbf{R}\|_F^2 - 2\text{tr}(\mathbf{S})) \end{aligned} \quad (5.23)$$

where $\mathbf{F} = \mathbf{R}\mathbf{S} \Rightarrow \mathbf{S} = \mathbf{F}^T \mathbf{R}$ (since \mathbf{S} is symmetric). \mathbf{P}_{ARAP} is then:

$$\mathbf{P}_{\text{ARAP}} = \mu(\mathbf{F} - \mathbf{R}) \quad (5.24)$$

A pretty easy \mathbf{P} again!

Bonet-Wood Neo-Hookean

There are many energies that call themselves 'Neo-Hookean'. One of the more popular ones is the one from the Bonet-Wood book [6]:

$$\Psi_{\text{BW08}} = \frac{\mu}{2} (\|\mathbf{F}\|_F^2 - 3) - \mu \log(J) + \frac{\lambda}{2} (\log(J))^2 \quad (5.25)$$

This is a bit more involved, so I'll jump to the result. Recall that the volume change of \mathbf{F} is $J = \det \mathbf{F}$, so \mathbf{P} of Ψ_{BW08} is

$$\mathbf{P}_{\text{BW08}} = \mu \left(\mathbf{F} - \frac{1}{J} \frac{\partial J}{\partial \mathbf{F}} \right) + \lambda \frac{\log J}{J} \frac{\partial J}{\partial \mathbf{F}} \quad (5.26)$$

A new term appears here: $\frac{\partial J}{\partial \mathbf{F}}$ is the gradient of J . In 3D, it can be written in terms of the *columns* of \mathbf{F} :

$$\mathbf{F} = \left[\begin{array}{c|c|c} & & \\ \mathbf{f}_0 & \mathbf{f}_1 & \mathbf{f}_2 \\ & & \end{array} \right] \quad (5.27)$$

With the identity

$$J = \mathbf{f}_0 \cdot (\mathbf{f}_1 \times \mathbf{f}_2) \quad (5.28)$$

we can define the following convenient shorthand for $\frac{\partial J}{\partial \mathbf{F}}$:

$$\frac{\partial J}{\partial \mathbf{F}} = \left[\begin{array}{c|c|c} & & \\ \mathbf{f}_1 \times \mathbf{f}_2 & \mathbf{f}_2 \times \mathbf{f}_0 & \mathbf{f}_0 \times \mathbf{f}_1 \\ & & \end{array} \right] \quad (5.29)$$

5.2.2 The next step

Computing forces is enough to get a simulator off the ground that does explicit time integration, but that just doesn't cut it in real-time. We want large timesteps, and as we discussed earlier, this requires some implicit timestepping scheme.

Such schemes, e.g. *backward Euler method* (Sec. 3.3.2) require the computation of the *force gradients*, i.e. $\frac{\partial \mathbf{f}}{\partial \mathbf{x}}$. All the tensor stuff we just did to get the Ps, just becomes so-so-so much more involved, if you try to get the gradient of it. It'll take some work to make force gradient computation equivalently easy and pretty.

5.3 Computing Force Gradients

Recall that we computed forces from the strain energy function as

$$\mathbf{f} = -v \frac{\partial \Psi}{\partial \mathbf{x}} \quad (5.30)$$

It's pretty straightforward from this that we can calculate the *force gradients* as

$$\frac{\partial \mathbf{f}}{\partial \mathbf{x}} = -v \frac{\partial^2 \Psi}{\partial \mathbf{x}^2} \quad (5.31)$$

In case of the force computation, the chain rule helped us to simplify the problem to

$$\frac{\partial \Psi}{\partial \mathbf{x}} = \frac{\partial \Psi}{\partial \mathbf{F}} : \frac{\partial \mathbf{F}}{\partial \mathbf{x}} \quad (5.32)$$

This baked all the difficulty into the energy-agnostic tensor $\frac{\partial \mathbf{F}}{\partial \mathbf{x}}$ that we only have to derive once. Then for each energy we only have to deal with the relatively easy $\frac{\partial \Psi}{\partial \mathbf{F}} \in \Re^{3 \times 3}$ (a.k.a. P).

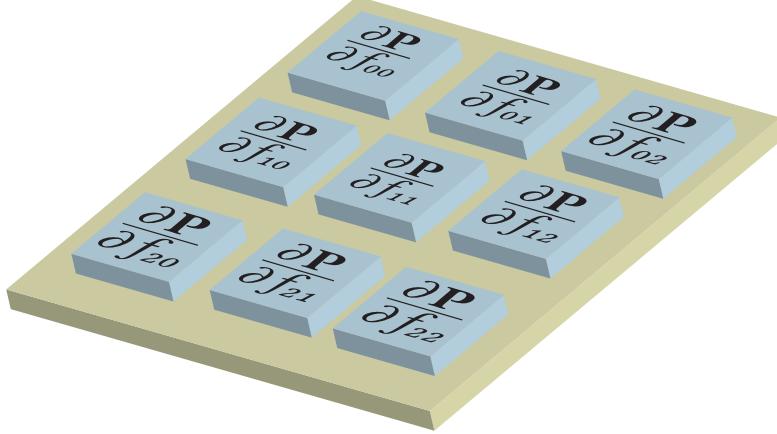


Figure 5.2: Thinking about 4th-order tensors as a matrix of matrices.

Can we do something similar for the force gradient? Sure we can! Using the chain rule yet again, we arrive at

$$\frac{\partial^2 \Psi}{\partial \mathbf{x}^2} = \frac{\partial \mathbf{F}^T}{\partial \mathbf{x}} \frac{\partial^2 \Psi}{\partial \mathbf{F}^2} \frac{\partial \mathbf{F}}{\partial \mathbf{x}} \quad (5.33)$$

That is, we take the *Hessian* of the strain energy function and left- and right-multiply it with the static 3rd-order tensor $\frac{\partial \mathbf{F}}{\partial \mathbf{x}}$.

5.3.1 The Mental Image of The Energy Hessian

The *Jacobian* of the strain energy function, $\frac{\partial \Psi}{\partial \mathbf{F}}$ is already the matrix $\mathbf{P}(\mathbf{F}) \in \Re^{3 \times 3}$. Taking the derivate of Ψ yet again yields a 4th-order tensor. It could be written like $\frac{\partial^2 \Psi}{\partial \mathbf{F}^2}$ as before, but $\frac{\partial \mathbf{P}}{\partial \mathbf{F}}$ means the same. We can write more explicitly

$$\frac{\partial^2 \Psi}{\partial \mathbf{F}^2} = \frac{\partial \mathbf{P}(\mathbf{F})}{\partial \mathbf{F}} = \left[\begin{array}{c|c|c} \left[\begin{array}{c} \frac{\partial \mathbf{P}}{\partial f_{00}} \\ \hline \frac{\partial \mathbf{P}}{\partial f_{10}} \\ \hline \frac{\partial \mathbf{P}}{\partial f_{20}} \end{array} \right] & \left[\begin{array}{c} \frac{\partial \mathbf{P}}{\partial f_{01}} \\ \hline \frac{\partial \mathbf{P}}{\partial f_{11}} \\ \hline \frac{\partial \mathbf{P}}{\partial f_{21}} \end{array} \right] & \left[\begin{array}{c} \frac{\partial \mathbf{P}}{\partial f_{02}} \\ \hline \frac{\partial \mathbf{P}}{\partial f_{12}} \\ \hline \frac{\partial \mathbf{P}}{\partial f_{22}} \end{array} \right] \\ \hline \end{array} \right] \quad (5.34)$$

Just like in the 3rd-order case, the bracketed $\frac{\partial \mathbf{P}}{\partial f_{ij}}$ are matrices, and they are entries in a higher-order tensor. As the 3rd-order tensor was a *vector of matrices*, the 4th-order tensor is a *matrix of matrices*. See Fig. 5.2 for visualisation.

The double-contraction operation looks quite similar to the 3rd-order case, except instead of producing a vector, the results are arranged into a matrix:

$$\begin{aligned} \mathbb{A} : \mathbf{B} &= \left[\begin{array}{cc} a_0 & a_2 \\ a_1 & a_3 \\ a_4 & a_6 \\ a_5 & a_7 \end{array} \right] \left[\begin{array}{cc} a_8 & a_{10} \\ a_9 & a_{11} \\ a_{12} & a_{14} \\ a_{13} & a_{15} \end{array} \right] : \left[\begin{array}{cc} b_0 & b_2 \\ b_1 & b_3 \end{array} \right] \\ &= \left[\begin{array}{cc} (a_0b_0 + a_1b_1 + a_2b_2 + a_3b_3) & (a_8b_0 + a_9b_1 + a_{10}b_2 + a_{11}b_3) \\ (a_4b_0 + a_5b_1 + a_6b_2 + a_7b_3) & (a_{12}b_0 + a_{13}b_1 + a_{14}b_2 + a_{15}b_3) \end{array} \right] \end{aligned} \quad (5.35)$$

Flattening the tensor

$$\mathbb{C} = \left[\begin{array}{cc} 1 & 3 \\ 2 & 4 \\ 5 & 7 \\ 6 & 8 \end{array} \right] \left[\begin{array}{cc} 9 & 11 \\ 10 & 12 \\ 13 & 15 \\ 14 & 16 \end{array} \right] = \left[\begin{array}{cc} [\mathbf{C}_{00}] & [\mathbf{C}_{01}] \\ [\mathbf{C}_{10}] & [\mathbf{C}_{11}] \end{array} \right] \quad (5.36)$$

requires first flattening in *column-wise order*, and then each matrix in turn:

$$\text{vec}(\mathbb{C}) = \left[\begin{array}{c} \text{vec}(\mathbf{C}_{00}) \\ \text{vec}(\mathbf{C}_{10}) \\ \text{vec}(\mathbf{C}_{01}) \\ \text{vec}(\mathbf{C}_{11}) \end{array} \right] = \left[\begin{array}{cccc} 1 & 5 & 9 & 13 \\ 2 & 6 & 10 & 14 \\ 3 & 7 & 11 & 15 \\ 4 & 8 & 12 & 16 \end{array} \right] \quad (5.37)$$

Taking the force gradient will require to multiply a 4th-order tensor with a 3rd-order one. Rather than defining the operation explicitly in 'tensor form', we are going to use the flattened matrix representation for that. Not only will this be easier to think about, but we will see some interesting structures appear.

5.3.2 Gradient Computation

Without further ado, here is the vectorized version of the force Jacobian which we are going to use through this text:

$$\frac{\partial^2 \Psi}{\partial \mathbf{x}^2} = \text{vec} \left(\frac{\partial \mathbf{F}}{\partial \mathbf{x}} \right)^T \text{vec} \left(\frac{\partial^2 \Psi}{\partial \mathbf{F}^2} \right) \text{vec} \left(\frac{\partial \mathbf{F}}{\partial \mathbf{x}} \right) \quad (5.38)$$

As I have already mentioned many times, we got the formula for the $\text{vec} \left(\frac{\partial \mathbf{F}}{\partial \mathbf{x}} \right)$, so '*all we need*' is to derive $\text{vec} \left(\frac{\partial^2 \Psi}{\partial \mathbf{F}^2} \right)$.

It certainly is all we need! However, if you try to derive it for all the energy functions out there, it's going to cost you a lot of tedious tensor manipulations. It's possible, but it's just too painful.

We are lucky, because there's a better way to deal with this. We can use an alternative representation of the energy function called *Cauchy-Green invariants*. It's going to provide us a much simpler scheme to derive the force Jacobians.

5.4 The Cauchy-Green Invariants

In our simulator, we restricted ourselves not only for conservative potential energies, but we also going to use exclusively *isotropic materials*. Isotropy is defined by requiring the constitutive behavior to be identical in any material direction. A more '*mathematical*' definition could state that the energy function's behavior is *invariant under rotation*. This implies that the relationship between Ψ and \mathbf{F} must be independent of the material axes chosen and, consequently, Ψ can be fully described by the Cauchy-Green invariants I_C , II_C , and III_C . These invariants arise from the characteristic polynomial of $\mathbf{C} = \mathbf{F}^T \mathbf{F}$, as follows:

$$\begin{aligned} I_C &= \text{tr}\mathbf{C} \\ II_C &= \text{tr}\mathbf{C}^2 \\ III_C &= \det\mathbf{C} = J^2 \end{aligned} \tag{5.39}$$

But, we are going to use them in the following form:

$$\begin{aligned} I_C &= \|\mathbf{F}\|_F^2 \\ II_C &= \|\mathbf{F}^T \mathbf{F}\|_F^2 \\ III_C &= \det(\mathbf{F}^T \mathbf{F}) \end{aligned} \tag{5.40}$$

Why is this specific representation beneficial for us? First, to make the invariants usable, we need to *rewrite* each energy function using them. For e.g. the St. Venant Kirchhoff this means

$$\begin{aligned} \Psi_{\text{StVK,stretch}} &= \|\mathbf{E}\|_F^2 \\ &= \frac{1}{4} \|\mathbf{F}^T \mathbf{F} - \mathbf{I}\|_F^2 \\ &= \frac{1}{4} \left(\|\mathbf{F}^T\|_F^2 - 2\text{tr}(\mathbf{F}^T \mathbf{F}) + \|\mathbf{I}\|_F^2 \right) \\ &= \frac{1}{4} II_C - \frac{1}{2} I_C + 3 \end{aligned} \tag{5.41}$$

5. Jacobians and Hessians

We can do this for *almost* any energy! This is good news, because we only got 3 invariants! If we derive all the invariants' gradients \mathbb{G}_x , and Hessians \mathbb{H}_x , we can arrive at a *generic representation* of the force Jacobian by using the chain rule on $\frac{\partial^2 \Psi}{\partial \mathbf{F}^2}$

$$\frac{\partial^2 \Psi}{\partial \mathbf{F}^2} = \frac{\partial^2 \Psi}{\partial I_C^2} \mathbb{G}_I + \frac{\partial \Psi}{\partial I_C} \mathbb{H}_I + \frac{\partial^2 \Psi}{\partial II_C^2} \mathbb{G}_{II} + \frac{\partial \Psi}{\partial II_C} \mathbb{H}_{II} + \frac{\partial^2 \Psi}{\partial III_C^2} \mathbb{G}_{III} + \frac{\partial \Psi}{\partial III_C} \mathbb{H}_{III} \quad (5.42)$$

Here, for each invariant $x \in \{I_C, II_C, III_C\}$, the gradient is $\mathbb{G}_x = \frac{\partial \Psi}{\partial \mathbf{F}}$, while the Hessian is $\mathbb{H}_x = \frac{\partial^2 \Psi}{\partial \mathbf{F}^2}$. We can define the energy Hessian in a bit more compact, but definitely more dull form as

$$\frac{\partial^2 \Psi}{\partial \mathbf{F}^2} = \sum_{x \in \{I_C, II_C, III_C\}} \frac{\partial^2 \Psi}{\partial x^2} \mathbb{G}_x + \frac{\partial \Psi}{\partial x} \mathbb{H}_x \quad (5.43)$$

This is a *complete rogue's gallery*. The \mathbb{G}_x and \mathbb{H}_x terms fully characterize the energy – there is no other term left! If we can work them out, all that's left for us is to derive the scalar derivative terms – all the $\frac{\partial \Psi}{\partial x}$ s and $\frac{\partial^2 \Psi}{\partial x^2}$ s – in case we want to get the Hessian of a new energy.

5.4.1 Generic Invariant Gradients and Hessians

Deriving the \mathbb{G}_x and \mathbb{H}_x terms definitely requires some effort. But, if you do it once, you can use it for any energy. For the sake of completeness, I will list here all the final results of the derivations, although I will omit the actual derivations themselves. The reader can refer to [15] for the complete calculations.

As we are going to use the *vectorized* version of the energy Hessian following Eq. 5.38, we have to first rewrite the generic Hessian as

$$\text{vec} \left(\frac{\partial^2 \Psi}{\partial \mathbf{F}^2} \right) = \frac{\partial^2 \Psi}{\partial I_C^2} \mathbf{g}_I \mathbf{g}_I^T + \frac{\partial \Psi}{\partial I_C} \mathbf{H}_I^T + \frac{\partial^2 \Psi}{\partial II_C^2} \mathbf{g}_{II} \mathbf{g}_{II}^T + \frac{\partial \Psi}{\partial II_C} \mathbf{H}_{II}^T + \frac{\partial^2 \Psi}{\partial III_C^2} \mathbf{g}_{III} \mathbf{g}_{III}^T + \frac{\partial \Psi}{\partial III_C} \mathbf{H}_{III}^T \quad (5.44)$$

Here $\text{vec}(\mathbb{G}_x) = \mathbf{g}_x \mathbf{g}_x^T$ and $\text{vec}(\mathbb{H}_x) = \mathbf{H}_x$. Then, all the \mathbf{g}_x and \mathbf{H}_x terms can be defined as follows:

First Invariant

$$\mathbf{g}_I = \text{vec} \left(\frac{\partial I_C}{\partial \mathbf{F}} \right) = 2 \text{vec}(\mathbf{F}) \quad (5.45)$$

$$\mathbf{H}_I = \text{vec} \left(\frac{\partial^2 I_C}{\partial \mathbf{F}^2} \right) = 2 \mathbf{I}_{9 \times 9} \quad (5.46)$$

Second Invariant

$$\mathbf{g}_{II} = 4 \text{vec}(\mathbf{F}\mathbf{E}) \quad (5.47)$$

$$\mathbf{H}_{II} = 4 \left(\mathbf{I}_{3 \times 3} \otimes \mathbf{F}\mathbf{F}^T + \mathbf{F}^T \mathbf{F} \otimes \mathbf{I}_{3 \times 3} + \mathbf{D} \right) \quad (5.48)$$

where \otimes denotes the Kronecker product, while D is

$$D = \begin{bmatrix} \mathbf{f}_0\mathbf{f}_0^T & \mathbf{f}_1\mathbf{f}_0^T & \mathbf{f}_2\mathbf{f}_0^T \\ \mathbf{f}_0\mathbf{f}_1^T & \mathbf{f}_1\mathbf{f}_1^T & \mathbf{f}_2\mathbf{f}_1^T \\ \mathbf{f}_0\mathbf{f}_2^T & \mathbf{f}_1\mathbf{f}_2^T & \mathbf{f}_2\mathbf{f}_2^T \end{bmatrix} \quad (5.49)$$

The \mathbf{f}_i s here are the columns of F , as of Eq. 5.27.

Third invariant We have already derived a fundamental building block for the third invariant. $\frac{\partial J}{\partial F}$ appeared once in the Bonet-Wood Neo-Hookean Piola-Kirchhoff stress tensor (Eq. 5.26), and we are going to define

$$\mathbf{g}_J = \text{vec}\left(\frac{\partial J}{\partial F}\right) = \text{vec}\left(\left[\begin{array}{c|c|c} \mathbf{f}_1 \times \mathbf{f}_2 & \mathbf{f}_2 \times \mathbf{f}_0 & \mathbf{f}_0 \times \mathbf{f}_1 \end{array}\right]\right) \quad (5.50)$$

We are also going to need

$$\mathbf{H}_J = \begin{bmatrix} \mathbf{0} & -\hat{\mathbf{f}}_2 & -\hat{\mathbf{f}}_1 \\ \hat{\mathbf{f}}_2 & \mathbf{0} & -\hat{\mathbf{f}}_0 \\ -\hat{\mathbf{f}}_1 & \hat{\mathbf{f}}_0 & \mathbf{0} \end{bmatrix} \quad (5.51)$$

using the hat operator \hat{x} , which turns a vector into a cross-product matrix:

$$\hat{\mathbf{x}} = \begin{bmatrix} 0 & -x_2 & x_1 \\ x_2 & 0 & -x_0 \\ -x_1 & x_0 & 0 \end{bmatrix} \quad (5.52)$$

With these in hand, the gradient and Hessian of the third invariant is

$$\mathbf{g}_{III} = 2 \det J \cdot \mathbf{g}_J \quad (5.53)$$

$$\mathbf{H}_{III} = 2\mathbf{g}_J\mathbf{g}_J^T + 2 \det J \cdot \mathbf{H}_J \quad (5.54)$$

5.4.2 Force Jacobian the Cauchy-Green Way

There are now three steps to derive the Hessian of an energy:

1. Re-write your energy in terms of I_C , II_C , and III_C .
2. Derive the scalar derivatives. Scalar! Derivates! Just mash them to Wolfram Alpha!
3. Plug the results into Eq. 5.44. You're all done.

5.4.3 Examples

St. Venant-Kirchhoff Stretching

Let's try it out on the stretching term from St. Venant-Kirchhoff:

$$\Psi_{\text{StVK,stretch}} = \|\mathbf{E}\|_F^2 \quad (5.55)$$

Step 1: Rewrite using invariants. We already did this in Eq. 5.41; the result is

$$\Psi_{\text{StVK,stretch}} = \frac{1}{4}II_C - \frac{1}{2}I_C + 3 \quad (5.56)$$

Step 2: Take the invariant derivatives. Here they are:

$$\frac{\partial \Psi}{\partial I_C} = -\frac{1}{2} \quad \frac{\partial^2 \Psi}{\partial I_C^2} = 0 \quad (5.57)$$

$$\frac{\partial \Psi}{\partial II_C} = \frac{1}{4} \quad \frac{\partial^2 \Psi}{\partial II_C^2} = 0 \quad (5.58)$$

$$\frac{\partial \Psi}{\partial III_C} = 0 \quad \frac{\partial^2 \Psi}{\partial III_C^2} = 0 \quad (5.59)$$

Okay this is easy! Almost all of them are zero.

Step 3: Plug the results into Eq. 5.44.

$$\begin{aligned} \text{vec} \left(\frac{\partial^2 \Psi_{\text{StVK,stretch}}}{\partial \mathbf{F}^2} \right) &= \frac{1}{4} \mathbf{H}_{II} - \frac{1}{2} \mathbf{H}_I \\ &= \frac{1}{4} \mathbf{H}_{II} - \mathbf{I}_{9 \times 9} \end{aligned} \quad (5.60)$$

That went pretty well! What about ARAP?

5.4.4 As-Rigid-As-Possible: Things Go Terribly Wrong

From the title you might guess that something odd is going to happen with ARAP. Let's find it out! ARAP can be expanded as

$$\begin{aligned} \Psi_{\text{ARAP}} &= \frac{\mu}{2} \|\mathbf{F} - \mathbf{R}\|_F^2 \\ &= \|\mathbf{F}\|_F^2 - 2\text{tr}(\mathbf{F}^T \mathbf{R}) + \|\mathbf{R}\|_F^2 \\ &= I_C - 2\text{tr}(\mathbf{F}^T \mathbf{R}) + 3 \end{aligned} \quad (5.61)$$

Hang on a minute! What is $\text{tr}(\mathbf{F}^T \mathbf{R})$? It does not correspond to any of the Cauchy-Green invariants!

To get to the punchline, the problem is going to boil down to the fact, that what we trying to take the symbolic derivative of something (\mathbf{R}) that is purely numerical. But let's just not rush that far ahead; let us first elaborate a bit on this $\text{tr}(\mathbf{F}^T \mathbf{R})$ term.

5.5 A New Set of Invariants?

This non-cooperative term, $\text{tr}(\mathbf{F}^T \mathbf{R})$, looks almost like

$$I_C = \text{tr}(\mathbf{F}^T \mathbf{F}) \quad (5.62)$$

which is already pretty suspicious! Furthermore, we can use the polar decomposition, $\mathbf{F} = \mathbf{RS}$, to reorder this weird term to

$$\text{tr}(\mathbf{F}^T \mathbf{R}) = \text{tr}(\mathbf{S}^T) = \text{tr}(\mathbf{S}) \quad (5.63)$$

What if we just... added this to our gallery of invariants? Is it a sensible question to ask? In the followings, we are going to look at invariants from two, different perspective, in order to understand them intuitively: we will examine their *rotation removing property*, aswell as their straightforward *geometric interpretation*.

5.5.1 Invariants as Rotation Removers

There are two, very important properties of the energy functions: they should be *invariant* to any rigid body *translations* and *rotations*. If an element has been merely translated or rotated, its deformation score should show up as zero. We've seen in Eq. 4.8, if $\mathbf{F} = \mathbf{I}$, \mathbf{F} corresponds to a pure translation. In case of the rotation invariance property, \mathbf{F} needs to be a pure rotation matrix, e.g.

$$\mathbf{F}(\theta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos \theta & -\sin \theta \\ 0 & \sin \theta & \cos \theta \end{bmatrix} \quad (5.64)$$

We have to design our energy functions carefully in order to get $\Psi(\mathbf{F}) = 0$ in both cases. We have actually already seen two, fundamentally different ways to deal with this.

The St. Venant-Kirchhoff Way

The stretching part of the St. Venant-Kirchhoff energy was

$$\Psi_{\text{StVK,stretch}} = \|\mathbf{E}\|_F^2 = \frac{1}{2} \|\mathbf{F}^T \mathbf{F} - \mathbf{I}\|_F^2 \quad (5.65)$$

If $\mathbf{F} = \mathbf{I}$, $\Psi_{\text{StVK,stretch}} = 0$ so it's definitely translationally invariant, but what about the rotational invariance? We can exploat a neat linear algebra identity here, that is, if \mathbf{F} is a pure rotation, then $\mathbf{F}^T \mathbf{F} = \mathbf{I}$. Plugging this back to Eq. 5.65, we have $\mathbf{I} - \mathbf{I} = \mathbf{0}$. Cool!

With the help of the polar decomposition $\mathbf{F} = \mathbf{RS}$ (Sec. 4.1.4) we can explicitly see the moment the rotation gets burned off:

$$\begin{aligned}\mathbf{F}^T \mathbf{F} &= (\mathbf{RS})^T \mathbf{RS} = \mathbf{S}^T \mathbf{R}^T \mathbf{RS} \\ &= \mathbf{S}^T \mathbf{I} \mathbf{S} = \mathbf{S}^T \mathbf{S} \\ &= \mathbf{S}^2\end{aligned}\tag{5.66}$$

We just found a new form of I_C , which is

$$I_C = \text{tr}(\mathbf{F}^T \mathbf{F}) = \text{tr}(\mathbf{S}^2)\tag{5.67}$$

and the StVK energy can be rewritten as

$$\Psi_{\text{StVK,stretch}} = \frac{1}{2} \|\mathbf{S}^2 - \mathbf{I}\|_F^2\tag{5.68}$$

The As-Rigid-As-Possible Way

In Section 3.4.2, we introduced a fairly unorthodox energy function: the As-Rigid-As-Possible or ARAP energy comes from geometry processing, and it has a distinctly different way of dealing with rotation removal.

The rotational part of \mathbf{F} , \mathbf{R} , is factorized from the polar decomposition, then it's simply subtracted from \mathbf{F} before its Frobenius norm is taken:

$$\Psi_{\text{ARAP}} = \frac{\mu}{2} \|\mathbf{F} - \mathbf{R}\|_F^2\tag{5.69}$$

If $\mathbf{F} = \mathbf{I}$, or \mathbf{F} is a pure rotation, we get the 0 valued energy. Score! This surely feels like a hack, but it works pretty well!

Similarly to the StVK energy, we can do some tensor algebra trickery to get it to some more interesting form. We can use the fact that the Frobenius norm does not change under rotation to burn off \mathbf{R} in this case as well:

$$\begin{aligned}\Psi_{\text{ARAP}} &= \|\mathbf{F} - \mathbf{R}\|_F^2 \\ &= \|\mathbf{R}^T(\mathbf{F} - \mathbf{R})\|_F^2 \\ &= \|\mathbf{R}^T(\mathbf{RS} - \mathbf{R})\|_F^2 \\ &= \|\mathbf{S} - \mathbf{I}\|_F^2\end{aligned}\tag{5.70}$$

This almost looks like $\frac{1}{2} \|\mathbf{S}^2 - \mathbf{I}\|_F^2$! How much does that square matters tough?

Comparing the Two Ways

If we want to *really* know what's going in a matrix, it's a pretty good idea to utilize SVD. (See Section 4.1.4.) The SVD of $\mathbf{F} = \mathbf{U}\Sigma\mathbf{V}^T$, and the entries of the diagonal matrix, Σ measure *exactly* how much stretching is going on along each of the 3D directions.

We can write both energies in terms of Σ , but yet again comes an obscene amount of algebra, which I'm going to omit here. You can look it up in [15] at page 53. In summary, we have

$$\Psi_{\text{StVK,stretch}} = \frac{1}{2} \|\Sigma^2 - \mathbf{I}\|_F^2 \quad (5.71)$$

$$\Psi_{\text{ARAP}} = \|\Sigma - \mathbf{I}\|_F^2 \quad (5.72)$$

Ignoring that $\frac{1}{2}$ in front of $\Psi_{\text{StVK,stretch}}$ – as we are looking for *qualitatively* different behaviours – the only real difference is that StVK squares all the singular values, while ARAP doesn't. Interesting!

Summing this all up, we have a new alias for I_C by exploiting $\mathbf{S} = \mathbf{V}^T\Sigma\mathbf{V}$:

$$I_C = \text{tr}(\mathbf{F}^T\mathbf{F}) = \text{tr}(\mathbf{S}^2) = \text{tr}(\Sigma^2) \quad (5.73)$$

while that ill-behaving term of ARAP becomes

$$\text{tr}(\mathbf{R}^T\mathbf{F}) = \text{tr}(\mathbf{S}) = \text{tr}(\Sigma) \quad (5.74)$$

When you looking at solely the singular values Σ , $\text{tr}(\mathbf{R}^T\mathbf{F})$ definitely look something similar to I_C . Again, how important is that squaring tough? To answer that, we'll need to look at the invariants from a different perspective.

5.5.2 Invariants as Geometric Measures

Recall that Σ from the SVD of \mathbf{F} , $\mathbf{F} = \mathbf{U}\Sigma\mathbf{V}^T$, represent the singular values of \mathbf{F} , which measure how much stretching is going on *exactly*.

Going back to the $\text{tr}(\Sigma^2)$ form of I_C , it works out to

$$I_C = \sigma_x^2 + \sigma_y^2 + \sigma_z^2 \quad (5.75)$$

where $\sigma_{x,y,z}$ are the singular values. This has a pretty straightforward geometric representation! As we discussed in Sec. 4.1.2, the matrix \mathbf{F} characterizes the rotation and *scaling* of an infinitesimal volumetric piece of a solid. On the other hand, the singular values are the amount of stretching and squashing *along each axis*; see Fig. 5.3!

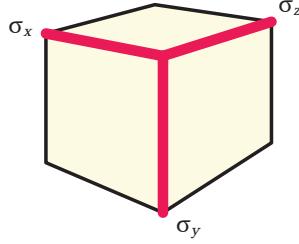


Figure 5.3: I_C measures the overall edge lengths of the entire cube. It i.e. sums up the squared lengths of each side of the cube.

So the invariant I_C is the *sum of the squared lengths of each side of the cube*. This is amazing! I actually haven't found any other material that described the Cauchy-Green invariants in such manner. We can now go ahead and define some similar term for both II_C and III_C . Both terms require some tweaking to get the final form presented in the followings. I omitted here the derivations, but you can look it up in the usual place, [15], page 55-56.

$II_C = \|\mathbf{F}^T \mathbf{F}\|_F^2$ works out to

$$II_C = \sigma_x^4 + \sigma_y^4 + \sigma_z^4 \quad (5.76)$$

which is lengths of each side of the cube, raised to the fourth power. This isn't differs much from I_C . To make it profitable, we need to fold it to an alternative form of

$$II_C^* = \frac{1}{2}(I_C - II_C) \quad (5.77)$$

II_C^* appears in e.g. [6], Example 6.5. Chugging through we will get

$$II_C^* = (\sigma_x \sigma_y)^2 + (\sigma_x \sigma_z)^2 + (\sigma_y \sigma_z)^2 \quad (5.78)$$

Whereas I_C was the summed squared lengths of the cube, II_C^* is the sum of the squared areas of the cube faces.

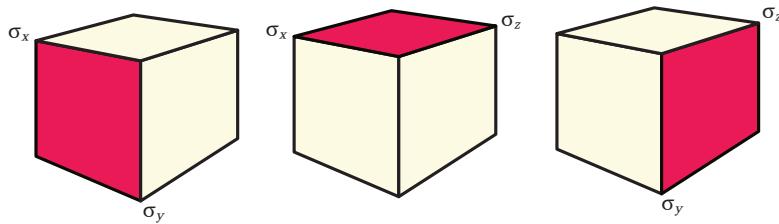


Figure 5.4: II_C measures the overall face areas of the cube. It i.e. sums up the squared areas of each side of the cube.

Finally, let's look at $III_C = \det(\mathbf{F}^T \mathbf{F})$. By definition, the determinant of a matrix is the product of its singular values, so this one is relatively easy:

$$III_C = (\sigma_x \sigma_y \sigma_z)^2 \quad (5.79)$$

that is, III_C measures the square of the cube's volume.

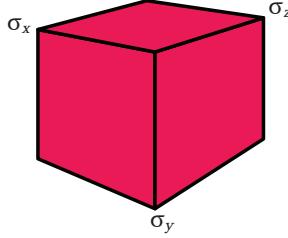


Figure 5.5: III_C measures squared volume of the cube.

Taken all together, all three invariants have a very intuitive geometric measure. It feels like a *fairly complete* way to describe how a small cube of material can deform, so it makes sense that the Cauchy-Green invariants is so popular among computational mechanists.

However, ARAP breaks this pattern; and it's a shame, as the use of rotation gradient \mathbf{R} turned out to be pretty handy, when one designs new energies, e.g. McAdams et al. use it in their co-rotational formulation as well. [18]

With all this knowledge in hand, let's look at our problem – describing $\text{tr}(\mathbf{R}^T \mathbf{F})$ – again.

5.6 S-based Invariants

I_C boiled down to

$$I_C = \text{tr}(\Sigma^2) \quad (5.80)$$

while the non-cooperative term in ARAP can be expressed as

$$\text{tr}(\mathbf{R}^T \mathbf{F}) = \text{tr}(\mathbf{S}) = \text{tr}(\Sigma) \quad (5.81)$$

The singular value version of I_C was

$$I_C = \sigma_x^2 + \sigma_y^2 + \sigma_z^2 \quad (5.82)$$

while the non-cooperative term takes the form.

$$\text{tr}(\mathbf{S}) = \sigma_x + \sigma_y + \sigma_z \quad (5.83)$$

This is the *unsquared version* of I_C ! Finally, here becomes clear that **you can't use a sum of squared values to express a sum of unsquared values**. This is why ARAP can't be written in terms of the Cauchy-Green invariants. This strongly suggest that $\text{tr}(\mathbf{S})$ should be its own invariant:

$$I_1 = \text{tr}(\mathbf{S}) \quad (5.84)$$

We essentially substituted $\mathbf{F}^T \mathbf{F}$ with \mathbf{S} . Can we do this to the other invariants as well? Doing so would reveal the new \mathbf{S} -based second and third invariant.

$$II_C = \text{tr}(\mathbf{F}^T \mathbf{F} \mathbf{F}^T \mathbf{F}) \rightarrow I_2 = \text{tr}(\mathbf{S}^2) \quad (5.85)$$

$$III_C = \det(\mathbf{F}^T \mathbf{F}) \rightarrow I_3 = \det(\mathbf{S}) \quad (5.86)$$

Is it a good set of invariants? Kim ([15], page 58) used these new invariants to express the original Cauchy-Green invariants. Doing so proved that the new invariants can describe a *super-set* of the phenomena that the Cauchy-Green ones could.

5.6.1 Does ARAP Work Now?

After all this work, let's try to write the ARAP energy in terms of our *new* invariants:

$$I_1 = \text{tr} \mathbf{S} \quad I_2 = \text{tr}(\mathbf{S}^2) \quad I_3 = \det \mathbf{S} \quad (5.87)$$

Can we do it? For the following computations, I will drop the $\frac{\mu}{2}$ term to make the derivations cleaner.

$$\begin{aligned} \Psi_{\text{ARAP}} &= \|\mathbf{F} - \mathbf{R}\|_F^2 \\ &= \|\mathbf{F}\|_F^2 - 2\text{tr}(\mathbf{F}^T \mathbf{R}) + \|\mathbf{R}\|_F^2 \\ &= I_2 - 2I_1 + 3 \end{aligned} \quad (5.88)$$

So far so good! Let's chug ahead and calculate the PK1:

$$\begin{aligned} \frac{\partial \Psi_{\text{ARAP}}}{\partial \mathbf{F}} &= \frac{\partial}{\partial \mathbf{F}} (I_2 - 2I_1 + 3) \\ &= \frac{\partial I_2}{\partial \mathbf{F}} - 2 \frac{\partial I_1}{\partial \mathbf{F}} \end{aligned} \quad (5.89)$$

Since $I_2 = I_C$

$$\frac{\partial I_2}{\partial \mathbf{F}} = \frac{\partial I_C}{\partial \mathbf{F}} = 2\mathbf{F} \quad (5.90)$$

while

$$\frac{\partial I_1}{\partial \mathbf{F}} = \frac{\partial \text{tr}(\mathbf{S})}{\partial \mathbf{F}} = \frac{\partial \text{tr}(\mathbf{R}\mathbf{F}^T)}{\partial \mathbf{F}} = \mathbf{R} \quad (5.91)$$

so

$$\mathbf{P}_{\text{ARAP}} = 2(\mathbf{F} - \mathbf{R}) \quad (5.92)$$

This exactly matches Eq. 5.24 – if you multiply it with $\frac{\mu}{2}$. Cool! What about the Hessian? It will expand to

$$\begin{aligned} \frac{\partial^2 \Psi_{\text{ARAP}}}{\partial \mathbf{F}^2} &= 2 \frac{\partial}{\partial \mathbf{F}} (\mathbf{F} - \mathbf{R}) \\ &= 2 \left(\frac{\partial \mathbf{F}}{\partial \mathbf{F}} - \frac{\partial \mathbf{R}}{\partial \mathbf{F}} \right) \end{aligned} \quad (5.93)$$

Oh noo! An unfamiliar term again! What is this $\frac{\partial \mathbf{R}}{\partial \mathbf{F}}$? It's called the *rotation gradient*, but we didn't see anything like this before. And our brand new I_1 invariant doesn't help us at all. What now?

5.7 The Eigenmatrices of the Rotation Gradient

We have this fancy, new, more expressive set of invariants, but we stuck again with the rotation gradient. The situation is even worse, as $\frac{\partial \mathbf{R}}{\partial \mathbf{F}} = \frac{\partial^2 I_1}{\partial \mathbf{F}^2}$, so if we can't figure out a good way to deal with this problem, then everything was for naught.

As I hinted already in Sec. 5.4.4, the core issue is, that *there isn't any method to obtain a tidy symbolic derivative of some numerical quantity* like \mathbf{R} . However, \mathbf{R} *does indeed* will have a simple, clean structure; we just need to look at it from the right perspective.

The trick is going to be that the *eigendecomposition* of $\frac{\partial \mathbf{R}}{\partial \mathbf{F}}$ will be a simple structure. Smith et al. in [25] were able to come up with the *analytic, closed-form* representation of the eigenvectors and eigenvalues – which are usually packed into \mathbf{Q} and Λ . The rotation gradient then can be simply calculated as $\frac{\partial \mathbf{R}}{\partial \mathbf{F}} = \mathbf{Q} \Lambda \mathbf{Q}^T$.

I think it's not the place where I need to stress how big of an achievement that is. And the researcher who found it, Theodore Kim, allegedly was just '*messing around in Matlab*' trying to '*plug in some easy-looking integers*' into the equations, until he '*found something that looked like a pattern*'. This process might sounds frighteningly informal to you, but as a – *hopefully soon to-be* – engineer, I found it facinating – as it just works and it's also pretty efficient in code, and I think that's all that matters.

But first, to be sure we are all on the same page about *eigenvalues*, *eigenvectors*, *eigendecomposition*, and the slightly more exotic *eigenmatrices*, let's recap these linear algebra terms real quick.

5.7.1 Eigenvalues, Eigenvectors, Eigendecomposition

Everybody knows the basic eigenvalue problem:

$$\mathbf{A}\mathbf{q}_0 = \lambda_0\mathbf{q}_0 \quad (5.94)$$

The eigenvalue λ_0 and the eigenvector \mathbf{q}_0 form an eigenpair of \mathbf{A} . The vector \mathbf{q}_0 is special because even after you push it through a multiply with \mathbf{A} , it remains exactly the same. Except, it was scaled by λ_0 .

A geometrical definition could state, that an eigenvector points in a direction in which it is stretched by the matrix – or more like the *transformation* – and the eigenvalue is the factor by which it is stretched.

This means that the eigenvectors \mathbf{q}_i and the corresponding eigenvalues λ_i of a matrix – *a.k.a. transform* – \mathbf{A} completely characterize the matrix itself. Thus, you can recover matrix \mathbf{A} from

$$\mathbf{A} = \mathbf{Q}\Lambda\mathbf{Q}^T \quad (5.95)$$

where \mathbf{Q} 's i^{th} column corresponds to an i^{th} eigenvector, \mathbf{q}_i . Λ is a diagonal matrix, whose diagonal elements are the corresponding eigenvalues, such that $\Lambda_{ii} = \lambda_i$.

5.7.2 What's an Eigenmatrix?

While everybody heard about the eigenvalues and eigenvectors, you might not have come across an eigenmatrix. For some 4th-order tensor

$$\mathbb{A} = \left[\begin{array}{cc} \left[\begin{array}{cc} a_0 & a_2 \\ a_1 & a_3 \\ a_4 & a_6 \\ a_5 & a_7 \end{array} \right] & \left[\begin{array}{cc} a_8 & a_{10} \\ a_9 & a_{11} \\ a_{12} & a_{14} \\ a_{13} & a_{15} \end{array} \right] \end{array} \right] \quad (5.96)$$

we define a similar eigenpair, $\{\lambda_0, \mathbf{Q}_0\}$, like in the matrix-world, but here we talk about an eigenmatrix, instead of an eigenvector. The equivalent eigenproblem then defined as

$$\mathbb{A} : \mathbf{Q}_0 = \lambda_0\mathbf{Q}_0 \quad (5.97)$$

where we used our favourite double-contraction operation instead of a plain matrix-vector multiply.

This can be expanded to

$$\begin{aligned}
 \mathbb{A} : \mathbf{Q}_0 &= \left[\begin{array}{cc} a_0 & a_2 \\ a_1 & a_3 \\ a_4 & a_6 \\ a_5 & a_7 \end{array} \right] \left[\begin{array}{cc} a_8 & a_{10} \\ a_9 & a_{11} \\ a_{12} & a_{14} \\ a_{13} & a_{15} \end{array} \right] : \left[\begin{array}{cc} q_0 & q_2 \\ q_1 & q_3 \end{array} \right] \\
 &= \left[\begin{array}{cc} (a_0q_0 + a_1q_1 + a_2q_2 + a_3q_3) & (a_8q_0 + a_9q_1 + a_{10}q_2 + a_{11}q_3) \\ (a_4q_0 + a_5q_1 + a_6q_2 + a_7q_3) & (a_{12}q_0 + a_{13}q_1 + a_{14}q_2 + a_{15}q_3) \end{array} \right] \\
 &= \lambda_0 \left[\begin{array}{cc} q_0 & q_2 \\ q_1 & q_3 \end{array} \right] = \lambda_0 \mathbf{Q}_0
 \end{aligned} \tag{5.98}$$

Again, this is essentially the same thing, as with the matrix eigenvalue-eigenvector problem: contracting \mathbb{A} with \mathbf{Q}_0 , \mathbb{A} stays the same. Except it was scaled by λ_0 .

Remember that vectorization process from Sec. 5.3.1, which turned this mighty 4th-order tensor into a matrix? It was $\text{vec}(\mathbb{A}) = \mathbf{A}$. Now the thing is that the flattened eigenmatrix \mathbf{Q}_0 of a tensor \mathbb{A} – which is a vector, of course – is going to be the eigenvector of the flattened tensor – which is a matrix. Are you still with me? It's much more understandable written like this:

$$\text{vec}(\mathbb{A})^T \text{vec}(\mathbf{Q}_0) = \mathbf{A}^T \mathbf{q}_0 \tag{5.99}$$

This is my favourite property of the vectorization process – it reveals that $\mathbf{A}\mathbf{q}_0 = \lambda_0\mathbf{q}_0$ and $\mathbb{A} : \mathbf{Q}_0 = \lambda_0\mathbf{Q}_0$ are essentially the same. Spelling out the whole thing verbosely yields:

$$\begin{aligned}
 \text{vec}(\mathbb{A})^T \text{vec}(\mathbf{Q}_0) &= \mathbf{A}^T \mathbf{q}_0 = \left[\begin{array}{cccc} a_0 & a_4 & a_8 & a_{12} \\ a_1 & a_5 & a_9 & a_{13} \\ a_2 & a_6 & a_{10} & a_{14} \\ a_3 & a_7 & a_{11} & a_{15} \end{array} \right]^T \left[\begin{array}{c} q_0 \\ q_1 \\ q_2 \\ q_3 \end{array} \right] \\
 &= \left[\begin{array}{c} a_0q_0 + a_1q_1 + a_2q_2 + a_3q_3 \\ a_4q_0 + a_5q_1 + a_6q_2 + a_7q_3 \\ a_8q_0 + a_9q_1 + a_{10}q_2 + a_{11}q_3 \\ a_{12}q_0 + a_{13}q_1 + a_{14}q_2 + a_{15}q_3 \end{array} \right] \\
 &= \lambda_0 \left[\begin{array}{c} q_0 \\ q_1 \\ q_2 \\ q_3 \end{array} \right] = \lambda_0 \mathbf{q}_0 = \lambda_0 \text{vec}(\mathbf{Q}_0)
 \end{aligned} \tag{5.100}$$

This property is very handy, because we don't need to come up with any new way to find the eigenmatrix of some tensor \mathbb{A} . You can use your favourite eigenvalue routine – be it Matlab, numpy, Eigen, or even LAPACK – and fold the $\{\lambda_i, \mathbf{q}_i\}$ pairs back into an eigenmatrix pair $\{\lambda_i, \mathbf{Q}_i\}$. But why do we even care about eigenmatrices, if they contain the exact same entries as the eigenvectors; just repackaged into a matrix form?

5.7.3 Structures Lurk in the Decomposition of an Eigenmatrix

The reason is that eigenmatrices are – matrices! This means that we can apply a variety of matrix based tools that are just not available for vectors. Recall again that we have this SVD thing (Sec. 4.1.4) which proved to be so useful already so many times. It was for \mathbf{F} :

$$\mathbf{F} = \mathbf{U}\Sigma\mathbf{V}^T \quad (5.101)$$

We could rotate any matrix – e.g. \mathbf{Q}_0 – to the same space as \mathbf{F} using $\mathbf{U}\mathbf{Q}_0\mathbf{V}^T$. This doesn't give any interpretable result, just a sea of meaningless numbers... But how about – following again Smith et al. [25] – rotating \mathbf{Q}_0 *out* of the space of \mathbf{F} using

$$\mathbf{U}^T\mathbf{Q}_0\mathbf{V} \quad (5.102)$$

This was *the trick*, which revealed that eigenmatrices of $\frac{\partial \mathbf{R}}{\partial \mathbf{F}}$ indeed have a simple clean structure. The whole thought process is documented in Section 5.4 in [15], so I will just jump to the final result now.

5.7.4 Building the Rotation Gradient

As the 3D rotation gradient is rank-3, we have the following three eigenpairs:

$$\lambda_0 = \frac{2}{\sigma_x + \sigma_y} \quad \mathbf{Q}_0 = \frac{1}{\sqrt{2}} \mathbf{U} \begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \mathbf{V}^T \quad (5.103)$$

$$\lambda_1 = \frac{2}{\sigma_y + \sigma_z} \quad \mathbf{Q}_1 = \frac{1}{\sqrt{2}} \mathbf{U} \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & -1 & 0 \end{bmatrix} \mathbf{V}^T \quad (5.104)$$

$$\lambda_2 = \frac{2}{\sigma_x + \sigma_z} \quad \mathbf{Q}_2 = \frac{1}{\sqrt{2}} \mathbf{U} \begin{bmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \\ -1 & 0 & 0 \end{bmatrix} \mathbf{V}^T \quad (5.105)$$

where again, σ_i s, \mathbf{U} , and \mathbf{V} comes from the SVD of \mathbf{F} . The matrices sandwiched between \mathbf{U} and \mathbf{V} are the twist matrices, corresponding to some infinitesimal rotation.

The $\frac{1}{\sqrt{2}}$ makes sense too. Eigenvectors have unit magnitude, and since \mathbf{U} and \mathbf{V} are already unitary matrices, the only things left to normalize are the 1 and -1 in the twist matrix. Since $\sqrt{1^2 + (-1)^2} = \sqrt{2}$, the normalization factor is $\frac{1}{\sqrt{2}}$. Neat, huh?

With the explicit λ_i and \mathbf{Q}_i formulas, we arrive at a very simple form of the vectorized rotation gradient:

$$\text{vec}\left(\frac{\partial \mathbf{R}}{\partial \mathbf{F}}\right) = \sum_{i=0}^2 \lambda_i \text{vec}(\mathbf{Q}_i) \text{vec}(\mathbf{Q}_i)^T \quad (5.106)$$

5.8 New Generic Hessian

As $\frac{\partial \mathbf{R}}{\partial \mathbf{F}} = \frac{\partial^2 I_1}{\partial \mathbf{F}^2}$, we now have everything in hand to make a generic Hessian representation, just like in Sec. 5.4.1; but with the Smith et al. [25] invariants!

The Smith et al. invariants were

$$I_1 = \text{tr}(\mathbf{S}) \quad I_2 = \text{tr}(\mathbf{F}^T \mathbf{F}) \quad I_3 = \det \mathbf{F} \quad (5.107)$$

and we now have the gradients (\mathbf{g}_i) and Hessians (\mathbf{H}_i) of each:

$$\begin{aligned} \mathbf{g}_1 &= \text{vec}(\mathbf{R}) & \mathbf{H}_1 &= \sum_{i=0}^2 \lambda_i \text{vec}(\mathbf{Q}_i) \text{vec}(\mathbf{Q}_i)^T \\ \mathbf{g}_2 &= \text{vec}(2\mathbf{F}) & \mathbf{H}_2 &= 2\mathbf{I}_{9 \times 9} \\ \mathbf{g}_J &= \text{vec}\left(\left[\begin{array}{c|c|c} \mathbf{f}_1 \times \mathbf{f}_2 & \mathbf{f}_2 \times \mathbf{f}_0 & \mathbf{f}_0 \times \mathbf{f}_1 \end{array}\right]\right) & \mathbf{H}_J &= \begin{bmatrix} \mathbf{0} & -\hat{\mathbf{f}}_2 & -\hat{\mathbf{f}}_1 \\ \hat{\mathbf{f}}_2 & \mathbf{0} & -\hat{\mathbf{f}}_0 \\ -\hat{\mathbf{f}}_1 & \hat{\mathbf{f}}_0 & \mathbf{0} \end{bmatrix} \end{aligned} \quad (5.108)$$

and the new generic Hessian is

$$\begin{aligned} \text{vec}\left(\frac{\partial^2 \Psi}{\partial \mathbf{F}^2}\right) &= \frac{\partial^2 \Psi}{\partial I_1^2} \mathbf{g}_1 \mathbf{g}_1^T + \frac{\partial \Psi}{\partial I_1} \mathbf{H}_1^T + \frac{\partial^2 \Psi}{\partial I_2^2} \mathbf{g}_2 \mathbf{g}_2^T + \frac{\partial \Psi}{\partial I_2} \mathbf{H}_2^T + \frac{\partial^2 \Psi}{\partial I_3^2} \mathbf{g}_3 \mathbf{g}_3^T + \frac{\partial \Psi}{\partial I_3} \mathbf{H}_3^T \\ &= \sum_{i=1}^3 \frac{\partial^2 \Psi}{\partial I_i^2} \mathbf{g}_i \mathbf{g}_i^T + \frac{\partial \Psi}{\partial I_i} \mathbf{H}_i \end{aligned} \quad (5.109)$$

Finally, we can follow the very same three-step process as before:

1. Rewrite energies Ψ using I_1 , I_2 , and I_3 .
2. Derive the scalar derivatives $\frac{\partial \Psi}{\partial I_1}$, $\frac{\partial^2 \Psi}{\partial I_1^2}$, $\frac{\partial \Psi}{\partial I_2}$, $\frac{\partial^2 \Psi}{\partial I_2^2}$, $\frac{\partial \Psi}{\partial I_3}$ and $\frac{\partial^2 \Psi}{\partial I_3^2}$.
3. Plug the result into Eq. 5.109.

5.8.1 Does ARAP Work Now?

Spoiler: it does! Again the ARAP energy was

$$\Psi_{\text{ARAP}} = \|\mathbf{F} - \mathbf{R}\|_F^2 \quad (5.110)$$

Step 1: Rewrite using invariants. We did all this mess just to do this:

$$\Psi_{\text{ARAP}} = I_2 - 2I_1 + 3 \quad (5.111)$$

Step 2: Take the invariant derivatives. Here they are:

$$\frac{\partial \Psi}{\partial I_1} = -2 \quad \frac{\partial^2 \Psi}{\partial I_1^2} = 0 \quad (5.112)$$

$$\frac{\partial \Psi}{\partial I_2} = 1 \quad \frac{\partial^2 \Psi}{\partial I_2^2} = 0 \quad (5.113)$$

$$\frac{\partial \Psi}{\partial I_3} = 0 \quad \frac{\partial^2 \Psi}{\partial I_3^2} = 0 \quad (5.114)$$

Step 3: Plug the results into Eq. 5.109.

$$\text{vec} \left(\frac{\partial^2 \Psi_{\text{ARAP}}}{\partial \mathbf{F}^2} \right) = 2\mathbf{I}_{9 \times 9} - 2\mathbf{H}_1 \quad (5.115)$$

Woah! Such a nice equation! And it's not just pretty, it's also very easy to implement, and if you have a nice SVD routine, it's also fast.

Thank YOU for following me on journey into the inner workings of the Smith et al. invariants. In the next chapter we are going to build directly on the analytic eigenvalues of the rotation gradient, to find out another nice property of ARAP.

Chapter 6

Keeping the Hessian Positive Definite

After all this heavy-weight tensor algebra, you probably thought we are done. We finally have the energy Hessian a.k.a. force Jacobian a.k.a. tangent stiffness matrix; so all we need to do now, is to plug this into some timestepping scheme to arrive at some linear system $\mathbf{Ax} = \mathbf{b}$. And we are done! Just solve for \mathbf{x} !

Okay, you are right, we indeed just need to solve this $\mathbf{Ax} = \mathbf{b}$ for \mathbf{x} . But, since we are in a performance critical environment – remember the title of this thesis is *interactive* FE simulations – it *does* matter, how we solve this system of equations.

The go-to method in Finite Element Analysis is to use some direct method, e.g. *LU factorization* of matrix \mathbf{A} and solve the equations by backsubstitution. Such methods are great! They provide the exact solution of *any matrix* in finite number of steps. However, in case of big matrices – and a usual Finite Element matrices consists couple thousand DoFs counts as a big one – they get pretty slow.

We are lucky, again – there is another way! Iterative methods instead of directly solving $\mathbf{Ax} = \mathbf{b}$, starts from some initial value \mathbf{x}_0 , and uses that to generate a sequence of improving approximate solutions. We can exploit that the resultant matrices are sparse – there is usually much more zeros in them than non-zeros – and *positive definite*!

That's nice, because the resulting matrix we got is sparse and positive-definite, right? Well, yes, but no! If you took Finite Element classes, you learnt that \mathbf{K} is positive-definite. However, the use of large timesteps produces substantial divergence from the steady state, leading to a symmetric linear system that is often indefinite. That is a very bad news and the later part of this chapter is going to deal with that problem.

If it's your first time you hear about iterative methods for solving a linear system, I highly recommend you: 1. Shewchuk's infamous guide – *An Introduction to the Conjugate Gradient Method Without the Agonizing Pain* [23] – and 2. Appendix B, where I solve a simple 2×2 problem with the method of steepest descent.

6.1 Solving a Linear System Iteratively

6.1.1 Meaning of Matrix's Definiteness

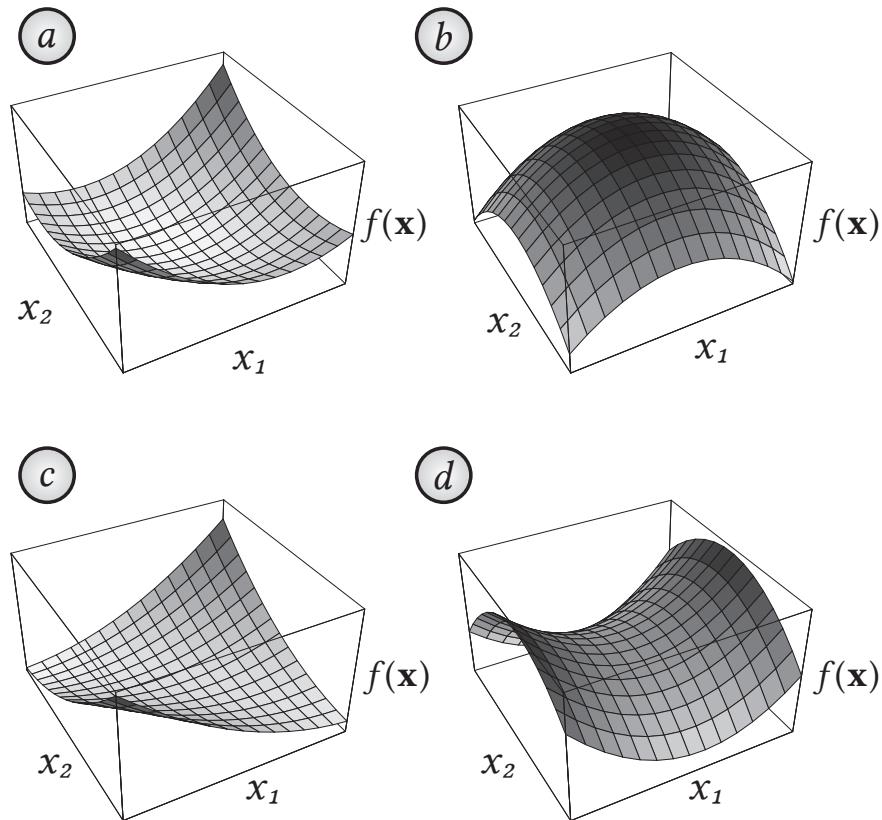


Figure 6.1: a – quadratic form for a positive definite matrix. b – for a negative definite matrix. c – for a singular (and positive-indefinite) matrix. A line that runs through the bottom of the valley is the set of solutions. d – for an indefinite matrix. Because the solution is a saddle point, steepest descent will not work. In three dimensions or higher, a singular matrix can also have a saddle.

Back to this definiteness – indefiniteness mess, what does it even mean if a matrix is definite or indefinite? I *always* prefer some visualization over a formula, so let's look at Fig. 6.1, where a plot of a positive definite, negative definite; singular and an indefinite matrix's quadratic form is shown. If you don't know what a quadratic form of some matrix is, please refer to Appendix B.

6.1.2 The Method of Steepest Descent

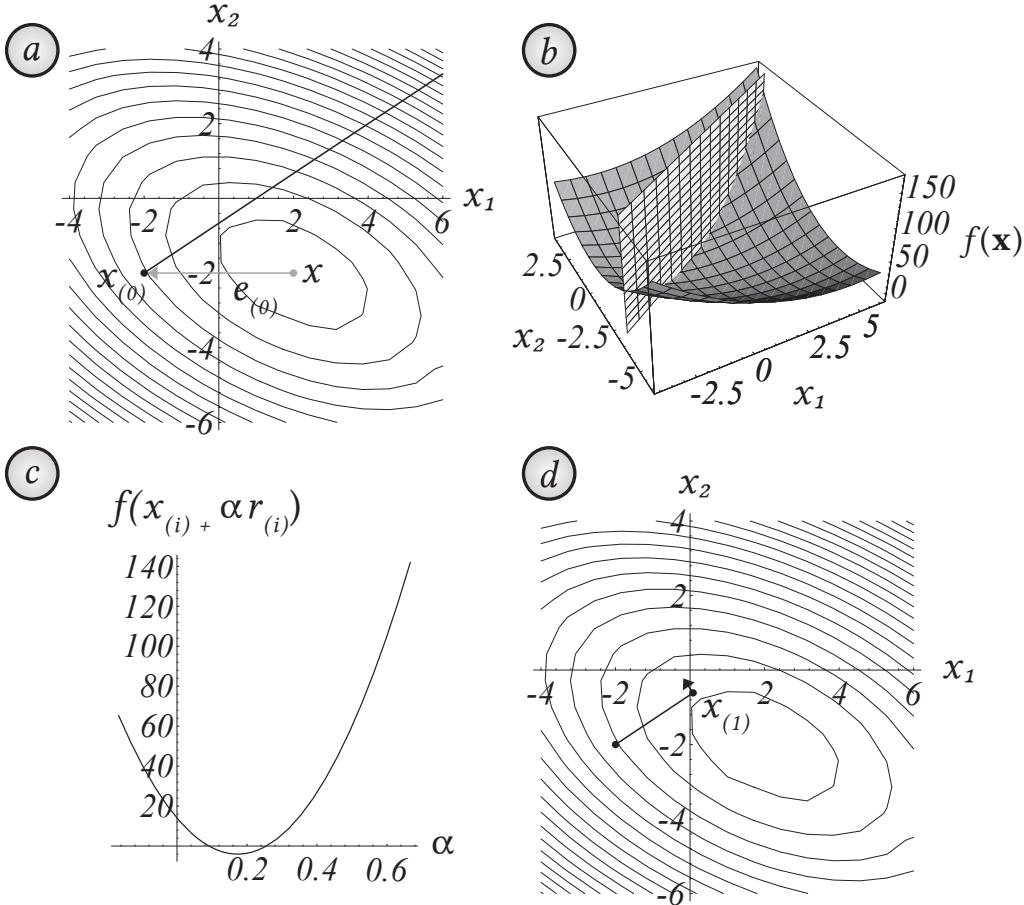


Figure 6.2: The method of Steepest Descent. a – Starting at $[-2, -2]^T$, take a step in the direction of steepest descent of f . b – Find the point on the intersection of these two surfaces that minimizes f . c – This parabola is the intersection of surfaces. The bottommost point is our target. d – The gradient at the bottommost point is orthogonal to the gradient of the previous step.

After looking at Fig. 6.1, you probably get the idea why we want positive definiteness: computing physics is most of the time a *minimization* problem. E.g. we want to find the global minima of the total potential energy function. If you've got something like Fig. 6.1a, you can imagine that you roll down a ball on that nice valley – down is the negative gradient of the surface – being careful that you *always* roll it down, up to the point that there is no more down. You can pull something like this off with a negative definite matrix, but definitely not with an indefinite one. What I just described is the method of steepest descent, which I also present of Fig. 6.2. And in Appendix B.

6.1.3 The Method of Conjugate Gradients

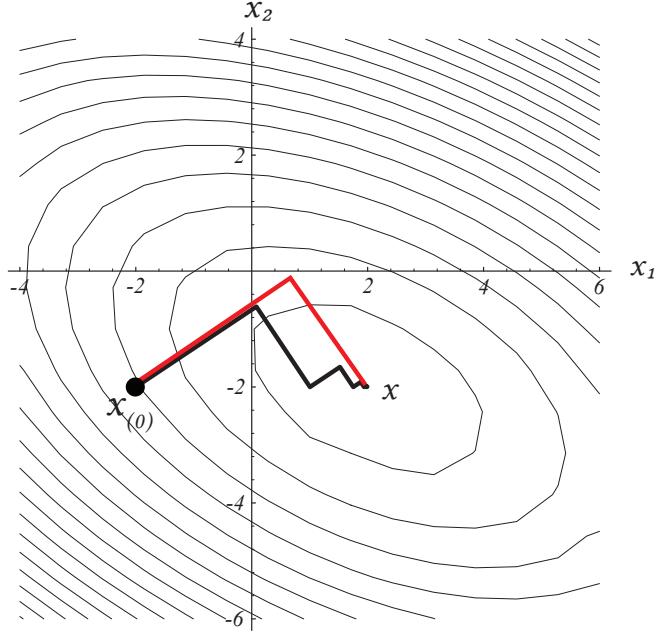


Figure 6.3: Black: The method of Steepest Descent starts at $[-2, -2]^T$ and converges at $[2, -2]^T$. Red: Method of Conjugate Gradient.

The example is run until it converges in Fig. 6.2. Not bad, however, note the zigzag path on Fig. 6.3. It appears because each gradient is *orthogonal* to the previous gradient. Wouldn't it be better if, every time we took a step, we got it right the first time?

To keep things short, most simulators use the method of Conjugate Gradient to solve the positive-definite, linear system. In a nutshell, while the method of steepest descent takes a step in the opposite direction of the gradient (Fig. 6.2, black), conjugate gradient steps in the direction of the conjugate vector (Fig. 6.2, red). I'm not going to derive the conjugate gradient method here, but instead wholeheartedly refer to Shewchuk's explanation. [23]

6.2 Solving the Issue of Definiteness

What to do with this definiteness-indefiniteness problem we talked about? To give you a hint, there is this definition for a positive definite matrix:

$$\mathbf{x}^T \mathbf{A} \mathbf{x} \geq 0 \quad \forall \mathbf{x} \quad (6.1)$$

or if you are a regular human you can just say that *all of its eigenvalues are greater or equal to zero*. This is actually semi-positive definiteness. Another way to look at this is through the product $\mathbf{Ax} = \mathbf{b}$. Positive definiteness implies that the sign of each entry in \mathbf{b} must always match the corresponding entry in \mathbf{x} . The matrix \mathbf{A} is not allowed to flip the sign of any entry in \mathbf{x} , no matter what the entries in \mathbf{x} are.

Here comes *the trick*: what if we *detect that the eigenvalue is less than zero*, we just set it to be zero? Surely we need all the eigenvalues and eigenvectors in order to reassemble the modified matrix, but isn't this eigenvalue sign is all the source of trouble? The answer is: yes, and this will be the solution.

But how to do that? One way is to compute the eigendecomposition of the global Hessian. However, the eigendecomposition of an $\mathfrak{R}^{n \times n}$ matrix takes $O(n^3)$ steps while conjugate gradient runs at $O(n^{\frac{3}{2}})$, so the cure is worse than the disease.

We can instead try per-element projection! If an element is indefinite, i.e. *any of its eigenvalues is less than 0*, then *clamp it to 0*. This would require the eigendecomposition of bunch of 9×9 , not the whole $n \times n$ system. The sum of semi-positive-definite matrices is known to also be semi-positive definite, so the strategy is sound.

6.3 Methods for Hessian Projection

This method of *eigenvalue clamping* is called Hessian projection in the literature, and it had some notable contribution in the last couple of years. The thing is, that since we talk about the Hessian of some specific, closed form energies, we can painstakingly analyze them, in order to make the problem simpler.

The method was first introduced by Teran et al. in [12] and [33]. They successfully reduced the problem to probing a 3×3 and three 2×2 eigenproblem.¹ Their method was very innovative and important, but the use of the Cauchy-Green invariants made it limiting, because it is insufficient to express stretch-based energies such as ARAP or the Co-rotational model.

¹Teran later moved away from the field and started working with the Material Point Method; he was on the team developing the snow simulation for the Disney movie Frozen. [30] Such an amazing movie!

6. Keeping the Hessian Positive Definite

This limitation was addressed in Stomakhin et al. in [29] by replacing the Cauchy-Green invariants with the singular values of the deformation gradient, but the inefficient numerical eigensolves for Hessian blocks remain. Xu et al. [35] improved these computations further by restricting the analysis to energies that satisfy the Valanis-Landel hypothesis. Unfortunately, many energies from geometry processing, such as MIPS [10] or Symmetric ARAP [27] do not fall into this category.

The work of McAdams et al. [18] presented an analytic solution for the indefiniteness of the Co-rotational energy. To do so, they decomposed the 4th order tensor defined by the energy Hessian into symmetric and skew-symmetric parts. This yields the eigenstructure of the Co-rotational model, albeit embedded inside 4th order tensors.

Smith et al. [25] proposed a more general derivation that produces their eigenvalue expressions as a special case, and additionally reveals the structure of the underlying eigenvectors. They successfully acquired closed-form, analytic expression for 6 eigenpairs of the rank-9 Hessian. The first three eigenpairs can be computed from the eigendecomposition of a 3×3 matrix in the most general case, but for many popular distortion energies, we can acquire closed-form expression as well.

6.4 Analytical Eigensystem of ARAP

Okay there are many nice articles about this Hessian projection thing, but what do they *actually* do? Well, we already talked a lot about these eigenthings in Sec. 5.7, when we tried to take the symbolic derivative of the rotation gradient:

$$\frac{\partial \mathbf{R}}{\partial \mathbf{F}} = \sum_{i=0}^2 \lambda_i \text{vec}(\mathbf{Q})_i \text{vec}(\mathbf{Q})_i^T = \mathbf{H}_1 \quad (6.2)$$

while the ARAP energy's Hessian was

$$\text{vec} \left(\frac{\partial^2 \Psi_{\text{ARAP}}}{\partial \mathbf{F}^2} \right) = 2\mathbf{I}_{9 \times 9} - 2\mathbf{H}_1 \quad (6.3)$$

This is *at least* suspicious! Let's just come back to earth and play a bit with that, it might reveal some clean structure for the eigenvalue as well.

6. Keeping the Hessian Positive Definite

We already know the eigensystem of the rotation gradient:

$$\lambda_0 = \frac{2}{\sigma_x + \sigma_y} \quad Q_0 = \frac{1}{\sqrt{2}} U \begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} V^T \quad (6.4)$$

$$\lambda_1 = \frac{2}{\sigma_y + \sigma_z} \quad Q_1 = \frac{1}{\sqrt{2}} U \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & -1 & 0 \end{bmatrix} V^T \quad (6.5)$$

$$\lambda_2 = \frac{2}{\sigma_x + \sigma_z} \quad Q_2 = \frac{1}{\sqrt{2}} U \begin{bmatrix} 0 & 0 & 1 \\ 0 & 0 & 1 \\ -1 & 0 & 0 \end{bmatrix} V^T \quad (6.6)$$

$$\lambda_{3..8} = 0 \quad Q_{3..8} = \text{subspace orthogonal to } Q_{0,1,2} \quad (6.7)$$

We have $-2H_1$, which just scales the eigenvalues of H_1 by -2 . Unfortunately, there is no magic theorem that takes the eigendecomposition of two matrices and tells you the eigendecomposition of their sum...

Unless if one of the matrices has a special structure, like $I_{9 \times 9}$! In that case, you can just add the eigenvalues from the diagonal matrix to the eigenvalues of your second matrix. The analytic eigendecomposition of the ARAP energy is then:

$$\lambda_0 = 2 - \frac{4}{\sigma_x + \sigma_y} \quad Q_0 = \frac{1}{\sqrt{2}} U \begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} V^T \quad (6.8)$$

$$\lambda_1 = 2 - \frac{4}{\sigma_y + \sigma_z} \quad Q_1 = \frac{1}{\sqrt{2}} U \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & -1 & 0 \end{bmatrix} V^T \quad (6.9)$$

$$\lambda_2 = 2 - \frac{4}{\sigma_x + \sigma_z} \quad Q_2 = \frac{1}{\sqrt{2}} U \begin{bmatrix} 0 & 0 & 1 \\ 0 & 0 & 1 \\ -1 & 0 & 0 \end{bmatrix} V^T \quad (6.10)$$

$$\lambda_{3..8} = 2 \quad Q_{3..8} = \text{subspace orthogonal to } Q_{0,1,2} \quad (6.11)$$

Back to the task we want to solve, how do we know if the energy goes indefinite? 2 is definitely a positive number, so we don't need to bother with $\lambda_{3..8}$.

6. Keeping the Hessian Positive Definite

But what about $\lambda_{0,1,2}$? It's so easy now! E.g. for λ_0 it's

$$\begin{aligned}\lambda_0 &= 2 - \frac{4}{\sigma_x + \sigma_y} \leq 0 \\ 2 &\leq \frac{4}{\sigma_x + \sigma_y} \\ \sigma_x + \sigma_y &\leq 2\end{aligned}\tag{6.12}$$

and similarly $\lambda_{1,2}$ works out to

$$\lambda_1 \leq 0 \iff \sigma_y + \sigma_z \leq 2\tag{6.13}$$

$$\lambda_2 \leq 0 \iff \sigma_x + \sigma_z \leq 2\tag{6.14}$$

So basically what we need to do now in our simulator, is that if any of the conditions listed above in Eq. 6.12-6.14 are tripped over, that is, $\lambda_i \leq 0$, than make $\lambda_i = 0$.

With these analytical eigenvalues in hand, this becomes so straightforward, that I can explicitly list here the C++ doing this in the simulator:

```
virtual Mat9 ARAP::GetHessian(const Mat3& F) const override {

    // first calculate  $\Sigma$ ,  $U$ , and  $V$  from the SVD of F and store the results
    JacobiSVD<Mat3> SVD{ F };
    const Vec3 Sigma = SVD.singularValues();
    const Mat3 U = SVD.matrixU();
    const Mat3 V = SVD.matrixV();

    // some precomputation
    double I[3];
    I[0] = Sigma(0) + Sigma(1);
    I[1] = Sigma(1) + Sigma(2);
    I[2] = Sigma(0) + Sigma(2);

    // calculate the eigenvalues  $\lambda_i$ ; clamp to 0.0 if necessary
    double lambda[3];
    for(int i = 0; i < 3; ++i)
        (I[i] >= 2.0) ? lambda[i] = 2.0 / I[i] : eigenvalue[i] = 1.0;

    // create the eigenmatrices  $Q_i = \frac{1}{\sqrt{2}}UT_iV^T$  from the twist matrices  $T_i$ 
    Vec9 Q[3];
    for(int i = 0; i < 3; ++i)
        Q[i] = Flatten(sq2inv * U * T[i] * V.transpose());

    // build the hessian  $\frac{\partial^2 \Psi_{ARAP}}{\partial F^2} = 2I_{9 \times 9} - 2H_1$ 
    Mat9 H = Mat9::Identity();
    for (int i = 0; i < 3; ++i)
        H -= lambda[i] * (Q[i] * Q[i].transpose());

    return 2.0 * H;
}
```

6.5 Analytical Eigendecompositions of Arbitrary Energies

Can we obtain such analytical eigendecomposition for *any* isotropic energy? Sure we can! We already have the eigenvalue of the first Smith et al. invariant, $\frac{\partial I_1}{\partial F^2}$, that's what the previous section was all about. The eigensystem for $\frac{\partial I_2}{\partial F^2}$ is also pretty easy, because it's a diagonal matrix.

The general eigensystem of I_3 is a bit more challenging. You can read the derivation in [15], Section 7.3. Actually I already reviewed the article which is all about that in the last paragraph of Sec. 6.3. It's basically doing the same thing what we did when we computed the rotation gradient.

Weirdly enough, $\lambda_{3\dots 5}$ and $\mathbf{Q}_{3\dots 5}$ is going to exactly equal $\lambda_{0\dots 2}$ and $\mathbf{Q}_{0\dots 2}$ from Eq. 6.8–6.10. $\lambda_{6\dots 8} = -\lambda_{3\dots 5}$ respectively, while the eigenmatrices $\mathbf{Q}_{6\dots 8}$ correspond to the *flip* eigenmode (instead of the *twist* eigenmodes of $\mathbf{Q}_{3\dots 5}$).

The first three eigensystem in the most general case, result in a quadratic system, that is encoded in a 3×3 matrix \mathbf{A} . We can construct this matrix \mathbf{A} by probing $\frac{\partial^2 \Psi}{\partial F^2}$ with scaling modes, e.g., the x -scaling vector is

$$\mathbf{D}_0 = \mathbf{U} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \mathbf{V}^T \quad (6.15)$$

then the entries of \mathbf{A} are obtained by computing the coefficients

$$a_{ij} = \text{vec}(\mathbf{D}_i)^T \text{vec}\left(\frac{\partial^2 \Psi}{\partial F^2}\right) \text{vec}(\mathbf{D}_j) \quad (6.16)$$

This might look a bit clumsy, but many popular distortion energies result closed form expression for $\lambda_{0\dots 2}$, making them an excellent candidate for real-time use. They are listed in [25] Section 5, and [15] Section 7.3.4.

Chapter 7

Some Production Practicalities

Before jumping to the implementation details of the simulator, let's look at some smart tricks applied in the industry.

7.1 Geometric Calculation of the Deformation Gradient

First let's look at a new, clever – and I think a bit more intuitive – way to calculate the deformation gradient F . The deformation gradient is the single most important measure deformable simulation: it is the fundamental building block of all the calculation we were doing. If you haven't done continuum mechanics before, you must be pretty excited to *finally* get your hands on the explicit formula for it.

7.1.1 The Mechanical Engineer's Way

Recall that we defined the motion which the structure is doing back in Sec. 4.1.2 as

$$\phi(\mathbf{x}) = \mathbf{F}\mathbf{x} + \mathbf{t} = \bar{\mathbf{x}} \quad (7.1)$$

where \mathbf{F} can be pretty much defined as

$$\mathbf{F} = \frac{\partial \bar{\mathbf{x}}}{\partial \mathbf{x}} \quad (7.2)$$

We have a discretized structure in the form of a finite element mesh, which is a set of elements. An element is defined by some nodes. However, we are not talking about a set of particles, but a continuum instead, so what's up with that?

One way of thinking about this, is that we want a method to reconstruct a continuous deformation map from a set of discrete samples. It's pretty much a definition for an *interpolation scheme* – and this is the way to think about it in the mechanical engineers' finite element world. This means that each point \mathbf{x} is defined as

$$\mathbf{x} = \sum_{i=1}^n h_i \mathbf{x}_i \quad (7.3)$$

where h_i is a standard interpolation function – usually called *shape function* in this context – for an n noded element.

This is it! All you have to do now is to plug this definition of \mathbf{x} to $\mathbf{F} = \frac{\partial \bar{\mathbf{x}}}{\partial \mathbf{x}}$, apply the chain rule, and you just acquired the deformation gradient for any element.

7.1.2 The Simulator Programmer's Way

Instead of plugging in Eq. 7.3 into the definition of \mathbf{F} , I'm going to present you a nice geometric way to calculate it. It was developed by Teran et al. in [12] and since it's introduction, literally everybody discretizes deformables with this method in computer graphics. It defines 2D and 3D simplex elements (triangle and tetrahedron) with piecewise linear bases, with a single quadrature point in its center. E.g. in case of a 2D triangle, it basically ends up with what we would get, if we would plug in shape functions

$$h_0 = 1 - u - v \quad h_1 = u \quad h_2 = v \quad (7.4)$$

into Eq. 7.3, then $\mathbf{F} = \frac{\partial \bar{\mathbf{x}}}{\partial \mathbf{x}}$, however, we will do it now in a much more intuitive way.

Calculating for a 2D Triangle

Let's say we have a triangle with vertices \mathbf{x}_0 , \mathbf{x}_1 , and \mathbf{x}_2 . After deformation, these same vertices have become $\bar{\mathbf{x}}_0$, $\bar{\mathbf{x}}_1$, and $\bar{\mathbf{x}}_2$. The question now is how do we compute a $\mathbb{R}^{2 \times 2}$ matrix \mathbf{F} that describes the rotation and scaling that occurs to transform the points $\bar{\mathbf{x}}_i$ into \mathbf{x}_i ?

We definitely want to pull off any translations, so let's deal with that first. Both triangles are floating off in space somewhere, so just to establish a common point of reference, let's pull them both back to the origin. If they're centered about the origin, then no relative translation needed to transform one into the other, and any remaining difference between the triangles must be due to rotation and scaling. We'll pull both triangles back to the origin by explicitly pinning $\bar{\mathbf{x}}_0$ and \mathbf{x}_0 to the origin (see Fig. 7.1).

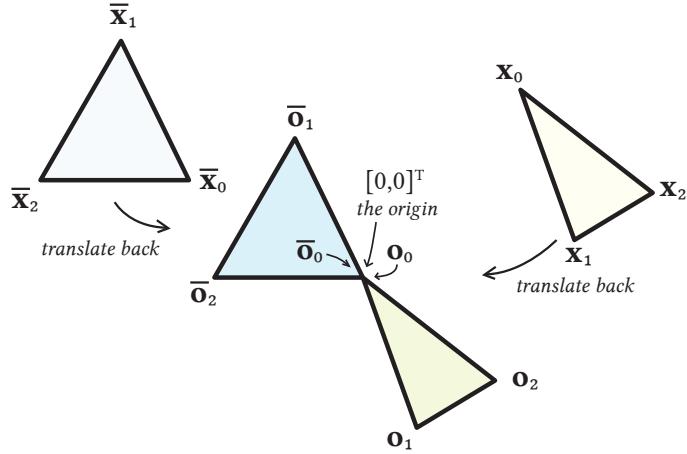


Figure 7.1: Let's eliminate the translation by pulling the rest triangle, back to the origin. We do the same thing to the deformed triangle.

The new vertices of our origin-centered rest-triangle become:

$$\begin{aligned} \mathbf{x}_0 &\rightarrow \mathbf{o}_0 = [0 \ 0]^T \\ \mathbf{x}_1 &\rightarrow \mathbf{o}_1 = \mathbf{x}_1 - \mathbf{x}_0 \\ \mathbf{x}_2 &\rightarrow \mathbf{o}_2 = \mathbf{x}_2 - \mathbf{x}_0 \end{aligned} \tag{7.5}$$

The vertices of the origin-centered deformed triangle are correspondingly:

$$\begin{aligned} \bar{\mathbf{x}}_0 &\rightarrow \bar{\mathbf{o}}_0 = [0 \ 0]^T \\ \bar{\mathbf{x}}_1 &\rightarrow \bar{\mathbf{o}}_1 = \bar{\mathbf{x}}_1 - \bar{\mathbf{x}}_0 \\ \bar{\mathbf{x}}_2 &\rightarrow \bar{\mathbf{o}}_2 = \bar{\mathbf{x}}_2 - \bar{\mathbf{x}}_0 \end{aligned} \tag{7.6}$$

Now we want to know what matrix \mathbf{F} will successfully rotate and scale all of our $\bar{\mathbf{o}}_i$ vertices so that they become the \mathbf{o}_0 vertices. In other words, we want the matrix \mathbf{F} satisfies these three equations:

$$\mathbf{F}\mathbf{o}_0 = \bar{\mathbf{o}}_0 \quad \mathbf{F}\mathbf{o}_1 = \bar{\mathbf{o}}_1 \quad \mathbf{F}\mathbf{o}_2 = \bar{\mathbf{o}}_2 \tag{7.7}$$

The first equation, $\mathbf{F}\mathbf{o}_0 = \bar{\mathbf{o}}_0$ is trivially satisfied by any matrix \mathbf{F} , since both $\bar{\mathbf{o}}_0$ and \mathbf{o}_0 are all zeros. Really we only need to worry about \mathbf{F} covering the other two cases by successfully producing $\bar{\mathbf{o}}_1$ and $\bar{\mathbf{o}}_2$ when provided with \mathbf{o}_1 and \mathbf{o}_2 .

If we write this down explicitly, it's a well-posed linear algebra problem:

$$\begin{aligned} \mathbf{F} \left[\begin{array}{c|c} \mathbf{x}_1 - \mathbf{x}_0 & \mathbf{x}_2 - \mathbf{x}_0 \end{array} \right] &= \left[\begin{array}{c|c} \bar{\mathbf{x}}_1 - \bar{\mathbf{x}}_0 & \bar{\mathbf{x}}_2 - \bar{\mathbf{x}}_0 \end{array} \right] \\ \mathbf{F} \left[\begin{array}{c|c} \mathbf{o}_1 & \mathbf{o}_2 \end{array} \right] &= \left[\begin{array}{c|c} \bar{\mathbf{o}}_1 & \bar{\mathbf{o}}_2 \end{array} \right] \\ \mathbf{FD}_m &= \mathbf{D}_s \end{aligned} \tag{7.8}$$

Computing the final \mathbf{F} then becomes straightforward:

$$\boxed{\mathbf{F} = \mathbf{D}_s \mathbf{D}_m^{-1}} \tag{7.9}$$

Calculating for 3D Tetrahedra

For a 3D tetrahedron, there is a straightforward generalization. In this case, we want a 3×3 version of \mathbf{F} . We translate the rest and deformed versions to the origin again, and after observing that $\mathbf{F}\bar{\mathbf{o}}_0 = \mathbf{o}_0$ is again trivial, we are left with a 3×3 formulation:

$$\begin{aligned} \mathbf{F} \left[\begin{array}{c|c|c} \mathbf{x}_1 - \mathbf{x}_0 & \mathbf{x}_2 - \mathbf{x}_0 & \mathbf{x}_3 - \mathbf{x}_0 \end{array} \right] &= \left[\begin{array}{c|c|c} \bar{\mathbf{x}}_1 - \bar{\mathbf{x}}_0 & \bar{\mathbf{x}}_2 - \bar{\mathbf{x}}_0 & \bar{\mathbf{x}}_3 - \bar{\mathbf{x}}_0 \end{array} \right] \\ \mathbf{F} \left[\begin{array}{c|c|c} \mathbf{o}_1 & \mathbf{o}_2 & \mathbf{o}_3 \end{array} \right] &= \left[\begin{array}{c|c|c} \bar{\mathbf{o}}_1 & \bar{\mathbf{o}}_2 & \bar{\mathbf{o}}_3 \end{array} \right] \\ \mathbf{FD}_m &= \mathbf{D}_s \\ \mathbf{F} &= \mathbf{D}_s \mathbf{D}_m^{-1} \end{aligned} \tag{7.10}$$

Pretty cool, huh? Also, you can calculate the volume of the tetrahedron from

$$v = \frac{1}{6} |\det \mathbf{D}_m| \tag{7.11}$$

7.1.3 Calculating the Derivative

Keeping up the momentum we also wanted the derivative

$$\frac{\partial \mathbf{F}}{\partial \mathbf{x}} = \frac{\partial \mathbf{D}_s}{\partial \mathbf{x}} \mathbf{D}_m^{-1} \tag{7.12}$$

$\frac{\partial \mathbf{D}_s}{\partial \mathbf{x}}$ is a 3rd-order tensor containing twelve matrices; as

$$\mathbf{x} = \begin{bmatrix} \mathbf{x}_0 \\ \mathbf{x}_1 \\ \mathbf{x}_2 \\ \mathbf{x}_3 \end{bmatrix} = \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{11} \end{bmatrix} \quad (7.13)$$

for a tetrahedra, the cheese slices on the cutting board (from Sec. 5.1) becomes

$$\frac{\partial \mathbf{D}_s}{\partial \mathbf{x}} = \begin{bmatrix} \left[\frac{\partial \mathbf{D}_s}{\partial x_0} \right] \\ \left[\frac{\partial \mathbf{D}_s}{\partial x_1} \right] \\ \vdots \\ \left[\frac{\partial \mathbf{D}_s}{\partial x_{11}} \right] \end{bmatrix} \quad (7.14)$$

while \mathbf{D}_m^{-1} is just a regular matrix. We get the result by multiplying through each $\frac{\partial \mathbf{F}}{\partial x_i} = \frac{\partial \mathbf{D}_s}{\partial x_i} \mathbf{D}_m^{-1}$. You can find the whole derivation in [15] Appendix E.

7.2 Force Computation Trick

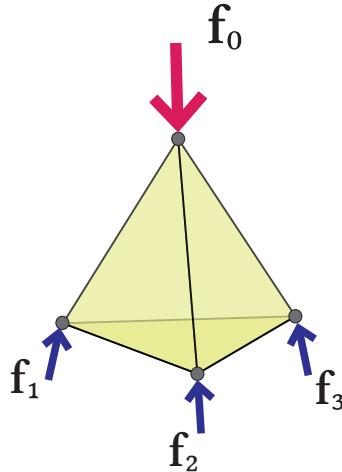


Figure 7.2: Optimizing force computation

This is nothing but a clever hack, and you probably already expect what it's all about from Fig. 7.2: consider the single quadrature point tetrahedra squished between your fingers. The force acting on the lower triangle ($\mathbf{f}_{1..3}$) should sum up to be equal to the negative force acting on the top vertex ($-\mathbf{f}_0$), that is

$$\mathbf{f}_0 = -(\mathbf{f}_1 + \mathbf{f}_2 + \mathbf{f}_3) \quad (7.15)$$

We just reduced the force computation from an \mathbb{R}^{12} to an \mathbb{R}^9 problem. And you can do this with the force Jacobian as well:

$$\frac{\partial \mathbf{f}_i}{\partial \mathbf{x}_0} = - \left(\frac{\partial \mathbf{f}_i}{\partial \mathbf{x}_1} + \frac{\partial \mathbf{f}_i}{\partial \mathbf{x}_2} + \frac{\partial \mathbf{f}_i}{\partial \mathbf{x}_3} \right) \quad (7.16)$$

where we reduced the $\mathbb{R}^{12 \times 12}$ problem to an $\mathbb{R}^{9 \times 12}$ by culling the first three rows of the Jacobian.

7.3 Boundary Condition Projection

At any given step of a finite element simulation, a node is either completely unconstrained (though subject to forces), or the node may be constrained in either one, two or three dimensions.

The easiest way to introduce this concept to our simulator, is to simply delete the rows and columns corresponding to the constrained degrees of freedom from the final matrix. E.g. in case of the backward Euler formulation,

$$\left[\mathbf{M} - h \frac{\partial \mathbf{f}}{\partial \dot{\mathbf{x}}} - h^2 \frac{\partial^2 \mathbf{f}}{\partial \mathbf{x}^2} \right] \ddot{\mathbf{u}} = h \mathbf{r} + h^2 \frac{\partial \mathbf{f}}{\partial \mathbf{x}} \quad (7.17)$$

reduces to

$$\mathbf{A}\mathbf{x} = \mathbf{b} \quad (7.18)$$

and you just delete the corresponding rows and columns of \mathbf{A} .

This is completely fine, but I've found it really challenging to implement it efficiently. I instead implemented the mass modification constraint enforcement mechanics from the original Baraff-Witkin paper [2]. The idea of their method is pretty simple: it comes from an old-school hack applied in rigid-body dynamics simulators. Such dynamic simulators usually store the *inverse mass* of the particles: $\ddot{\mathbf{x}} = \frac{1}{m_i} \mathbf{f}_i$. When inverse mass is used, it becomes trivial to enforce constraints by altering the mass.

Suppose for example that we want to keep particle i 's velocity from changing. If we take $\frac{1}{m_i}$ to be zero, we give the particle an *infinite mass*, making it *ignore all forces exerted on it*. Complete control over a particle's acceleration is thus taken care of by storing a value of zero for the particle's inverse mass. What if we wish to constrain the particle's acceleration in only one or two dimensions? Although we normally think of a particle's mass as a scalar, we need not always do so. Suppose we write

$$\ddot{\mathbf{x}}_i = \begin{bmatrix} 1/m_i & 0 & 0 \\ 0 & 1/m_i & 0 \\ 0 & 0 & 0 \end{bmatrix} \mathbf{f}_i \quad (7.19)$$

Now $\ddot{\mathbf{x}}_i$ must lie in the xy plane. Following this an unconstrained particles inverse mass matrix is $\frac{1}{m_i} \mathbf{I}_{3 \times 3}$.

Of course, we are not restricted to coordinate-aligned constraints. More generally, given a unit vector $\hat{\mathbf{p}}_i \in \Re^3$, a particle is prevented from accelerating along $\hat{\mathbf{p}}_i$ by using an inverse mass matrix $\frac{1}{m_i} (\mathbf{I} - \hat{\mathbf{p}}_i \hat{\mathbf{p}}_i^T)$. This follows from the fact that $(\mathbf{I} - \hat{\mathbf{p}}_i \hat{\mathbf{p}}_i^T) \hat{\mathbf{p}}_i = \mathbf{0}$. Similarly, given two mutually orthogonal unit vectors $\hat{\mathbf{p}}_i$ and $\hat{\mathbf{q}}_i$, we prevent a particle from accelerating in either the $\hat{\mathbf{p}}_i$ or $\hat{\mathbf{q}}_i$ direction by using the inverse mass matrix $\frac{1}{m_i} (\mathbf{I} - \hat{\mathbf{p}}_i \hat{\mathbf{p}}_i^T - \hat{\mathbf{q}}_i \hat{\mathbf{q}}_i^T)$.

This trick really fascinates me! We end up with the filter S_i for each node, where

$$S_i = \begin{cases} \mathbf{I}, & \text{if zero DOFs are constrained} \\ \mathbf{I} - \hat{\mathbf{p}}_i \hat{\mathbf{p}}_i^T, & \text{if one DOF } (\hat{\mathbf{p}}_i) \text{ is constrained} \\ \mathbf{I} - \hat{\mathbf{p}}_i \hat{\mathbf{p}}_i^T - \hat{\mathbf{q}}_i \hat{\mathbf{q}}_i^T, & \text{if two DOFs } (\hat{\mathbf{p}}_i \text{ and } \hat{\mathbf{q}}_i) \text{ are constrained} \\ \mathbf{0}, & \text{if all three DOFs are constrained} \end{cases} \quad (7.20)$$

then we modify $\mathbf{Ax} = \mathbf{b}$, according to [32] as

$$(\mathbf{SAS}^T + \mathbf{I} - \mathbf{S}) \mathbf{y} = \mathbf{Sc} \quad (7.21)$$

$$\mathbf{y} = \mathbf{x} - \mathbf{z} \quad (7.22)$$

$$\mathbf{c} = \mathbf{b} - \mathbf{Az} \quad (7.23)$$

where \mathbf{z} is constrained vertices' velocity change.

This surely needs some explanation! The key equation is the first one. The \mathbf{SAS}^T term is a straightforward projection of the original matrix \mathbf{A} into the subspace spanned by the filters \mathbf{S} . According to Eberle in [15] (page 126) this creates a nearly rank-deficient matrix in the subspace comprised of the DOFs that were removed from the system. The $\mathbf{I} - \mathbf{S}$ term then *serves to improve the conditioning of this subspace*. Anywhere that filter was applied, the $\mathbf{I} - \mathbf{S}$ essentially gives things a boost with a bunch of ones.

This is easiest to see if some particle i was constrained entirely, and $S_i = \mathbf{0}$. Now we are in trouble because we have a 3×3 block of zeros along the diagonal. The *Gershgorin circle theorem* (roughly) states that the eigenvalue corresponding to the row must lie inside some disc. The diagonal entry prescribes the disc's center ($a_{ii} = 0$), and the radius is the absolute sum of the off-diagonal entries in that row ($r = \sum_{j \neq i} \|a_{ij}\|$). The fact that the disc must be centered at zero is bad news, because it means that the eigenvalue is close to zero, and almost certainly ruining the conditioning of the matrix. The $\mathbf{I} - S_i$ fix will paste an identity matrix precisely on top of that zero block, and replace those zeros along the diagonal with ones. The Gershgorin discs are now centered around one, and eigenvalues will be near one, which is a much better state.

7.4 Interpolating the Results on a Higher Resolution Mesh

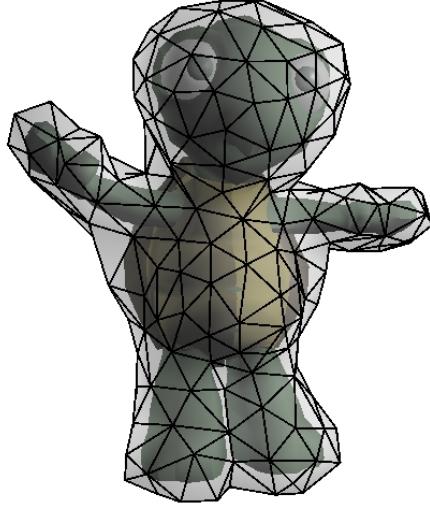


Figure 7.3: A high-res surface mesh embedded inside the volumetric FEM mesh.

In FEM, we simplify the structure to make the problem computable. However, we can easily interpolate the results on a higher resolution mesh, as you can see on Fig. 7.3.

First we need to figure out in which FEM element (e) each vertex of the higher resolution mesh $\hat{\mathbf{x}}_k$ is. Then we will calculate the $\hat{\mathbf{x}}_k$'s position, but in the FEM element's *local coordinate system*, that is, assigning an *interpolation weight* \hat{h}_i – or more specifically a *barycentric weight* – to each FEM element vertex $\mathbf{x}_i^{(e)}$, so in the end we get back the original $\hat{\mathbf{x}}_k$ from

$$\hat{\mathbf{x}}_k = \sum_0^3 \hat{h}_i \mathbf{x}_i^{(e)} \quad (7.24)$$

This is Eq. 7.3 for a 4-noded tetrahedron! You just calculate the weight \hat{h}_i at startup or load them from disk, then the interpolation is nothing but this multiplication with the weight. So-so cheap!

Chapter 8

Implementing a Finite Element Simulator

As an example, what can be done with the nonsense I presented so far, I implemented a *naive* simulator architecture. *Naive* here means, that I focused on readability and the educational value over performance, so it didn't become an interactive solution in the end. The full source code can be found at marcisolti.github.io/ifem. I used the following technologies for the implementation:

- The simulator is developed with **C++ on MSVC**.
- For all math related stuff – '*dense*' and '*sparse*' linear algebra, solution of sparse linear system with conjugate gradient, SVD – I used the **Eigen math library** [8]. This allowed to translate the equations to C++ code almost in 1:1 manner.
- **DirectX 12**¹ for drawing stuff on the screen real-time.
- Some naive parallelization is done with Intel's **Threading Building Blocks** (TBB)

8.1 Core Algorithm

First let me present you the *core algorithm* of our simulator. We will transform this algorithm to C++ code in a pretty straightforward manner. First we need to initialize the simulator where we precompute some stuff, then we will go to a loop where we timestep the chosen integrator forever:

¹Special thanks for László Szécsi for the nice DirectX 12 class (*Grafikus játékok fejlesztése*) at BME-VIK.

8. Implementing a Finite Element Simulator

initialize simulator:

- Parse config file
- Initialize the EnergyFunction class for computing $\frac{\partial \Psi}{\partial F}$ and $\frac{\partial^2 \Psi}{\partial F^2}$.
- Load mesh, build mass matrix M . (See Sec. 3.3.1)
- Parse boundary conditions, build S . (Sec. 7.3)
- Precalculate D_m^{-1} , v (as of Sec. 7.1.2) and $\frac{\partial F}{\partial x}$ (Sec. 7.1.3) for each element.
- Setup the Interpolator class for calculating f_{ext} .
- *(Set mesh initial position)*

loop:

1. $T += dt$,
2. According to config:
 - (a) Get new f_{ext} from the Interpolator
 - (b) *(Update S according to $d\dot{u}$)*
3. Build $K \Rightarrow$ for each element:
 - (a) Calculate $F = D_s D_m^{-1}$ (As of Sec. 7.1.2)
 - (b) Get the Jacobian $\frac{\partial \Psi}{\partial F}$ from the EnergyFunction (Like in Sec 5.2.1)
 - (c) $f_{int} += -v \frac{\partial F^T}{\partial x} \frac{\partial \Psi}{\partial F}$ (See Eq. 5.5 and Eq. 5.18 in the beginning of Chapter 5.)
 - (d) Get the Hessian $\frac{\partial^2 \Psi}{\partial F^2}$ from the EnergyFunction;

either the Cauchy-Green way (Eq. 5.44) or the Smith et al. way (Eq 5.109)

 \rightarrow add 'eigenvalues clamping' if desired, according to Chapter 6.
 - (e) Calculate $\frac{\partial^2 \Psi}{\partial x^2} = -v \frac{\partial F}{\partial x} \frac{\partial^2 \Psi}{\partial F^2} \frac{\partial F}{\partial x}$ (See Eq. 5.38)
 - (f) $K += \frac{\partial^2 \Psi}{\partial x^2}$ according to Fig. 3.4.
4. **do timestep**
See Sec. 8.2.6.

endloop

8.2 A Naive Simulator Architecture

We want to implement the core algorithm from Sec. 8.1 in C++ using the Eigen math library. As it's a *naive implementation*, focusing on readability and the code's educational value, we don't need to worry much about the architecture, so what about this:

```
// instantiate a solver object.
Solver solver;

// initialize the solver object with the config file
solver.StartUp(config)

// loop forever
while(1) {
    // step the simulation
    Vec result = solver.Step();

    // do whatever you want with the result
    // store it in memory, save it to disc, draw to the screen etc...
}
```

Seems pretty okay to me!

8.2.1 Configuring the Simulator

We want to tell the simulator tons of information after the executable is built: what volumetric model to use, what are the material parameters, which are the loaded/constrained vertices; and the list goes on. The most straightforward way would be a config file, more specifically a plain old json file. I used the classic C++ json implementation by Niel Lohmann [17]. An example for the data structure is listed in Appendix C.

8.2.2 The Solver Class

In Sec. 8.2 I showed how the simulator should be used: We instantiate a Solver object – like `Solver solver` – then we call this instance's methods – as `solver.StartUp(config)` or `solver.Step()` – in order to do something with it.

This is because I used the object-oriented programming paradigm, where the data and their methods to manipulate it are grouped together in the form of objects. We define a **class** or a prototype of an object, then we instantiate these prototypes. Intuitively, a **class** is a cookie cutter, while its instance is the cookie what you cut with the cookie cutter from the dough.

8. Implementing a Finite Element Simulator

So our main goal is to define a class which can be used the way we determined in Sec. 8.2. I did this:

```
class Solver
{
    // some resources for the simulator
    // presented in the next section

public:
    // methods for initialization/deinitialization
    Vec StartUp(json* config);
    void Shutdown();

    // stepping the simulator
    Vec Step();

    // helper functions
private:
    void BuildFintAndKeff();
    void ComputeElementJacobianAndHessian(int i);

    void FillFint();
    void FillKeff();

    Mat3 ComputeDm(int i);
    Mat9x12 ComputedFdx(Mat3 DmInv);
};
```

8.2.3 Resources

We have our methods defined, what about the data? Let me walk you through the data fields of the solver class, which lays out all the necessary resources for a FEM simulator. The first thing is an enumeration defined outside the solver class

```
enum Integrator { qStatic, bwEuler, Newmark };
```

which is nothing but a more convinient way to say

```
int integrator = ...; // 0 = qStatic, 1 = bwEuler, 2 = Newmark
```

Our Solver class look like this:

```
class Solver
{
    json* config;

    VolumetricMesh* mesh;
    uint32_t numDOFs, numElements, numVertices;
    std::vector<int> indexArray;

    Integrator integrator;
    EnergyFunction* energyFunction;
```

8. Implementing a Finite Element Simulator

```
// time integration variables
double T, h, magicConstant;
uint32_t numSubsteps;
double alpha, beta;

// boundary conditions
Interpolator interpolator;
std::vector<int> loadedVerts, BCs;
SpMat S;

// matrices and vectors
SpMat Keff, M, spI;
Vec x_0, u, x, v, a, z, fExt;

// precomputed stuff
std::vector<double> tetVols;
std::vector<Mat3> DmInvs;
std::vector<Mat9x12> dFdxs;

// linear solver object
ConjugateGradient cgSolver;

public:
    // methods presented before
};
```

First we have the `json* config`, which stores a pointer – a reference – to the global config object. The app’s whole state is stored in a json object, which allows me to easily serialize/deserialize – load from/save to the disc – the whole app’s state.

We then have a `VolumetricMesh* mesh`, which stores a pointer – not so surprisingly – to the volumetric mesh. I used the Vega FEM library’s [3] volumetric mesh class, which allows me to load meshes from disk in their proprietary .veg format.

There’s not too much about it: it’s nothing but a text file, containing list of vertices and a list of vertex indices, defining volumetric mesh data. In the not so distant future I would like to define my own mesh interface, as I’m planning to support mesh import from ‘proper’ FEM softwares – e.g. ANSYS, or my personal favourite, Hypermesh.

`uint32_t numDOFs, numElements, numVertices;` is storing the number of DOFs, etc., in a 32-bit wide unsigned integer, or in other words, in a regular integer but without the sign. In the `std::vector<int> indexArray` I store the index of each tetrahedron: as tetrahedra has 4 vertices, each 4 `int` makes up a tetrahedron, so the vector’s length is `4 * numElements`.

The integrator is nothing but a glorified `int` as we discussed it. `energyFunction` is a pointer to an `EnergyFunction` object. We will discuss this in the next section.

8. Implementing a Finite Element Simulator

The next set of numbers – **double** is a double precision floating point a.k.a. real number – are some time integration variables. The time integration methods are presented in Sec. 8.2.6.

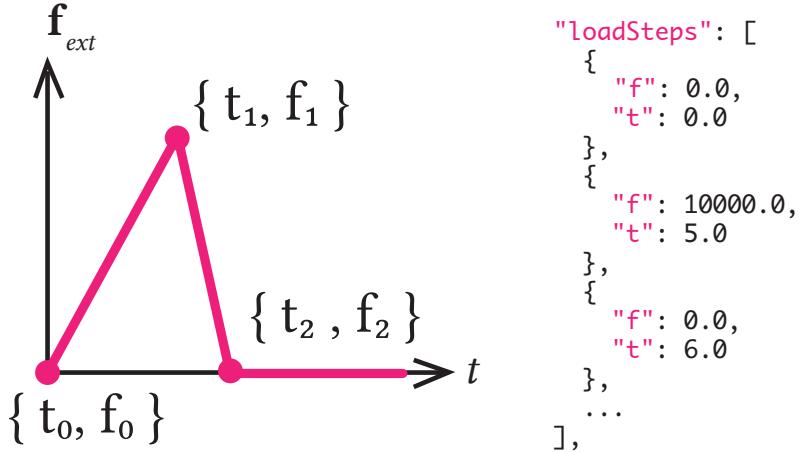


Figure 8.1: Linear interpolation for the load steps

The `interpolator` object stores the load steps and interpolates between them with simple linear interpolation: you get the value for f in case of a specific t from

$$f = f_i + (t - t_i) \frac{f_{i+1} - f_i}{t_{i+1} - t_i} \quad (8.1)$$

You define your load steps in the json as you can see on Fig. 8.1. Then you just call `interpolator.get(t)` and you get your f_{ext} value for a specific timepoint t .

We store a list of integers – a.k.a. `std::vector<int>` – for both the loaded and fixed vertex indices. We also have the sparse matrix object – `SpMat` – for the matrix S , as of Sec 7.3. Next up is all the sparse resources for time integration, like K , M , or f_{int} and f_{ext} , etc., we will use these extensively in Sec. 8.2.6.

It is a significant optimization to precalculate stuff, and we are able to do that with D_m^{-1} , v , and $\frac{\partial F}{\partial x}$ for each element. These are stored in the vectors `DmInvs`, `tetVols` and `dFdxs` respectively.

Last but definitely not least, we have our conjugate gradient solver object `cgSolver`, provided by Eigen to us.

8.2.4 Initialization

Before we can *finally* start using our simulator, we need to do some precalculations and initialization at the application startup. This was pretty much written down already in the core algorithm (Sec. 8.1), the only part that needs some further explanation is the initialization of the `EnergyFunction`.

Yet another principle of object-oriented programming coming up: different `EnergyFunction` classes are *derived* from the `EnergyFunction` base class, thus all descended `EnergyFunction` classes inherits all the data and member function of the base class. That is one feature of the derived classes, another one is polymorphism: we can use different classes with the same interface:

```
// two EnergyFunction pointers
EnergyFunction* arap, neoHookean;

// create the ARAP and NeoHookean instances
// both are derived from the EnergyFunction class
// so you can use EnergyFunction* pointers to reference them
arap = new ARAP{ E1, nu1 };
neoHookean = new NeoHookean{ E2, nu2 };

// GetJacobian(const Mat3& F) is a method of the base class
// So you can call it from any of the derived classes
// This means that you use all the derived energy function classes the same way
Vec9 arapJacobian = arap->GetJacobian(F);
Vec9 neoHookeanJacobian = neoHookean->GetJacobian(F);
```

This will come handy in the simulation loop! But now, at initialization, our duty is to figure out which `EnergyFunction` we want to use. It's done by parsing the config file and calling a good ole' `new` on the corresponding energy function.

```
if (energyName == "ARAP") {
    energyFunction = new ARAP{ E, nu };
} else if(energyName == "SNH")
    energyFunction = new StableNeoHookean{ E, nu };
} else if(energyName == "StVK")
    energyFunction = new StVK{ E, nu };
}
```

These are the `EnergyFunctions` supported by the simulator:

- Projected ARAP as of Sec. 6.4,
- classic Saint Venant-Kirchhoff,
- or Smith et. al Stable Neo-Hookean from [24].

8. Implementing a Finite Element Simulator

Next is to deal with a practical thing: we initialized the `EnergyFunction` with E and ν , which is not μ and λ we were talking about all the time. μ and λ are the *Lamé parameters*, which are perfectly fine, however, Young's modulus E and Poissons's ratio ν are much more common. If you look up a real material's data, it's probably going to be $\{E, \nu\}$ not $\{\mu, \lambda\}$, so I made $\{E, \nu\}$ the default. $\{\mu, \lambda\}$ are calculated at initialization from

$$\mu = \frac{E}{2(1 + \nu)} \quad (8.2)$$

$$\lambda = \frac{Ev}{(1 + \nu)(1 - 2\nu)} \quad (8.3)$$

We also precompute some other stuff necessary for each `EnergyFunction`, e.g. all the fixed T_i twist matrices for ARAP, $\frac{1}{\sqrt{2}}$ as `1.0/std::sqrt(2.0)`, etc.

8.2.5 Simulation loop

After we initialized everything, we'll get into an infinite loop

```
while(1) {
    Vec result = solver.Step();
    // do stuff with the result ...
}
```

stepping the simulation forward forever in time.

Each timestep starts with some setup:

1. Set stuff: `int subStep = 0; T += h;`
2. Get the load value f_{ext} from the interpolator:
`double loadValue = interpolator.get(T)`
3. Set f_{ext} according to the config and the interpolator:
`for (auto index : loadedVerts) fExt(index) = loadValue;`
4. Zero f_{int} and K

8. Implementing a Finite Element Simulator

Then the magic is happening here:

```

for(;;) // loop forever
{
    // Build fint and K
    BuildFintAndKeff();

    // do a timestep according to the chosen integrator
    switch(integrator)
    {
        case qStatic:
        {
            // do timestep the quasistatic way ...
        }
        break;
        case bwEuler:
        {
            // do timestep the backward Euler way ...
        }
        break;
        case Newmark:
        {
            // do timestep the Newmark way ...
        }
        break;
    }

    // exit the loop if you did sufficient amount of substeps
    if(++substep >= numSubsteps)
        break;
}

```

I think this code is pretty self-explanatory: we build \mathbf{f}_{int} and \mathbf{K} , then do a timestep with the chosen integrator scheme. But what schemes are available, exactly?

8.2.6 Time Integration

Back in Sec. 3.3.2 we looked at ways how to timestep a *dynamic process*, and arrived at the closed form solution for $\dot{\mathbf{u}}$. But what about just simply timestepping the plain old $\mathbf{Ku} = \mathbf{f}$ first, does that make sense?

Quasistatic Formulation

Yes it does, and it's called the *quasistatic formulation*. It's basically the regular Newton-Raphson iteration from Sec. 3.2.3:

$$\mathbf{Ku} = -\mathbf{f}_{int} + \mathbf{f}_{ext} \quad (8.4)$$

$$\mathbf{x} += \kappa \mathbf{u} \quad (8.5)$$

8. Implementing a Finite Element Simulator

where κ is specified with line search in [25], but you can find the process in the Bonet-Wood book [6] as well (Section 9.6.2). I didn't want to implement a line search *yet*, so I used a magic constant instead which has to be sufficiently small. This implemented in the simulator in the following way:

```
case qStatic:
{
    // Ku = -fint + fext
    // then do the constraint projection stuff
    SpMat SystemMatrix = S * Keff * S + spI - S;
    Vec SystemVec = S * (-fInt + fExt);

    // factorize and solve the final matrix
    solver.compute(SystemMatrix);
    Vec u = solver.solve(SystemVec);

    // increment as x += κu
    x += magicConstant * u;
}
break;
```

Isn't it nice to see how clearly math translates to C++?

Dynamics Simulations

We derived the **backward Euler** timestepping scheme back in Sec. 3.3.2, which is

$$\left[\mathbf{M} - h \frac{\partial \mathbf{f}}{\partial \dot{\mathbf{x}}} - h^2 \frac{\partial \mathbf{f}}{\partial \mathbf{x}} \right] \dot{\mathbf{u}} = h \mathbf{f} + h^2 \frac{\partial \mathbf{f}}{\partial \mathbf{x}} \dot{\mathbf{x}} \quad (8.6)$$

$$\dot{\mathbf{x}} += \dot{\mathbf{u}} \quad (8.7)$$

$$\mathbf{x} += h \dot{\mathbf{x}} \quad (8.8)$$

It translates to C++ pretty well too:

```
case bwEuler:
{
    // h(fint + fext) + h2 ∂f / ∂ẋ
    Vec RHS = h * ((fInt + fExt) + h * Keff * v);

    // [M - h ∂f / ∂ẋ - h2 ∂f / ∂x]
    SpMat EffectiveMatrix = M - h * (alpha * Keff + beta * M) - h2 * Keff;

    // project constraints
    Vec SystemVec = S * RHS;
    SpMat SystemMatrix = S * EffectiveMatrix * S + spI - S;
```

8. Implementing a Finite Element Simulator

```

solver.compute(SystemMatrix);
Vec dv = solver.solve(SystemVec);

v += dv;
x += h * v;
}
break;
}

```

So easy! I also implemented the **implicit Newmark** scheme, because why wouldn't I do it, I had to just look it up in the university FEM lecture notes. [31] In case of constant acceleration, $\alpha = 1/4$, and $\gamma = 1/2$, so the Newmark scheme becomes:

$$\left(\frac{4M}{h^2} + \frac{2C}{h} + K \right) u_{i+1} = f_{i+1} + M \left(\ddot{u}_i + \frac{4}{h} \dot{u}_i + \frac{4}{h^2} u_i \right) + C \left(\dot{u}_i + \frac{2}{h} u_i \right) \quad (8.9)$$

$$\dot{u}_{i+1} = \frac{2}{h} (u_{i+1} - u_i) \quad (8.10)$$

$$\ddot{u}_{i+1} = \frac{4}{h^2} (u_{i+1} - u_i) - \frac{4}{h} \dot{u}_i - \ddot{u}_i \quad (8.11)$$

I omitted the C++ but it's the same straightforward translation as before.

8.2.7 Building The Force Jacobian

After 87 pages of blood and sweat, we arrived at the C++ listing of the force Jacobian calculation. In Sec. 8.2.5, we called the function `BuildFintAndKeff()`:

```

void Solver::BuildFintAndKeff()
{
    for(int i = 0; i < numElements; ++i)
        ComputeElementJacobianAndHessian(i)
}

```

`ComputeElementJacobianAndHessian(i)` is then calculates f_{int} and K for the i^{th} element. According to the core algorithm from Sec. 8.1, it was something like this:

1. Calculate $F = D_s D_m^{-1}$ (As of Sec. 7.1.2)
2. Get the Jacobian $\frac{\partial \Psi}{\partial F}$ from the `EnergyFunction` (Like in Sec 5.2.1)
3. $f_{int} += -v \frac{\partial F^T}{\partial x} \frac{\partial \Psi}{\partial F}$ (See Eq. 5.5 and Eq. 5.18 in the beginning of Chapter 5.)
4. Get the Hessian $\frac{\partial^2 \Psi}{\partial F^2}$ from the `EnergyFunction`
5. Calculate $\frac{\partial^2 \Psi}{\partial x^2} = -v \frac{\partial F}{\partial x} \frac{\partial^2 \Psi}{\partial F^2} \frac{\partial F}{\partial x}$ (See Eq. 5.38)
6. $K += \frac{\partial^2 \Psi}{\partial x^2}$ according to Fig. 3.4.

8. Implementing a Finite Element Simulator

Let me just walk you through the explicit C++ from the simulator! We start by getting the 4 indices for the vertices we are going to work with, then calculate the deformation gradient \mathbf{F} following Sec. 7.1.2.

```
void Solver::ComputeElementJacobianAndHessian(int i)
{
    const int* indices = &(indexArray[4 * i]); //← not pretty ...

    Mat3 F;
    {
        // filling up vectors with the << operator
        Vec3 v0, v1, v2, v3;
        {
            v0 << x(indices[0] + 0), x(indices[0] + 1), x(indices[0] + 2);
            v1 << x(indices[1] + 0), x(indices[1] + 1), x(indices[1] + 2);
            v2 << x(indices[2] + 0), x(indices[2] + 1), x(indices[2] + 2);
            v3 << x(indices[3] + 0), x(indices[3] + 1), x(indices[3] + 2);
        }

        const Vec3 ds1 = v1 - v0;
        const Vec3 ds2 = v2 - v0;
        const Vec3 ds3 = v3 - v0;

        // you can use << to fill up matrices as well.
        Mat3 Ds;
        Ds <<
            ds1[0], ds2[0], ds3[0],
            ds1[1], ds2[1], ds3[1],
            ds1[2], ds2[2], ds3[2];
        const Mat3 DmInv = DmInvs[i];
        F = Ds * DmInv;
    }
}
```

That's it! We have discretized the kinematics! Loving this method for calculating \mathbf{F} ! Next up: calculating \mathbf{f}_{int} the the i^{th} element:

```
// get v
const double tetVol = tetVols[i];
// get the precomputed vec( $\frac{\partial \mathbf{F}}{\partial \mathbf{x}}$ )
const Mat9x12 dFdx = dFdxs[i];
{
    // and the Jacobian vec(P) = vec( $\frac{\partial \Psi}{\partial \mathbf{F}}$ )
    const Vec9 Pv = energyFunction->GetJacobian(F);

    // the internal forces then are nothing but
    //  $\mathbf{f}_{int}^{(i)} = -v \operatorname{vec}\left(\frac{\partial \mathbf{F}}{\partial \mathbf{x}}\right)^T \operatorname{vec}\left(\frac{\partial \Psi}{\partial \mathbf{F}}\right)$ 
    const Vec12 fEl = -tetVol * dFdx.transpose() * Pv;

    // then add  $\mathbf{f}_{int}^{(i)}$  to the global  $\mathbf{f}_{int}$ 
    for (int el = 0; el < 4; ++el)
        for (int incr = 0; incr < 3; ++incr)
            fInt(indices[el] + incr) += fEl(3 * el + incr);
}
```

8. Implementing a Finite Element Simulator

Yet again a straightforward transformation of math to C++! The jacobian can be constructed in the same manner

```
{
    // get vec  $\left( \frac{\partial^2 \Psi}{\partial F^2} \right)$ 
    const Mat9 dPdF = energyFunction->GetHessian(F);

    // we already have the precomputed vec  $\left( \frac{\partial F}{\partial x} \right)$ , so the Hessian is
    //  $K^{(i)} = -v \text{vec} \left( \frac{\partial F}{\partial x} \right)^T \text{vec} \left( \frac{\partial^2 \Psi}{\partial F^2} \right) \text{vec} \left( \frac{\partial F}{\partial x} \right)$ 
    const Mat12 dPdx = -tetVol * dFdx.transpose() * dPdF * dFdx;

    // now we just add  $K^{(i)}$  to the global  $K$  with this crazy loop
    for (int y = 0; y < 4; ++y)
        for (int x = 0; x < 4; ++x)
            for (int innerY = 0; innerY < 3; ++innerY)
                for (int innerX = 0; innerX < 3; ++innerX)
                    Keff.coeffRef(indices[x] + innerX, indices[y] + innerY) +=
                        dPdx(3 * x + innerX, 3 * y + innerY);
}

}
```

Except for that crazy loop at the end, it wasn't that hard either right? And we are just done! We finally has this mighty force Jacobian defined. However, this code is a bit slow, and we can tweak it a tiny bit to make it much-much faster with some trivial trickery.

8.2.8 Naive Parallelization of the Global Matrix Assembly

Did you know that computers now have more CPUs inside their CPU? It's crazy, right? The whole thing is called CPU, while the CPUs inside the CPU are called *cores*. E.g. if you have a CPU with 4 cores, you can run 4 streams of machine language instructions in parallel. In other words, instead of executing 1 instruction per cycle, you can execute 4! This is not that simple, but this is the core idea.

So we have this big K and f_{int} calculation loop, which we do *for each element*. This feels like an excellent example for parallel programming! E.g. if you have 1000 elements, instead of calculating all 1000 on a single core, you can devide it up between the cores, e.g. do 250 on each core parallel, if you have 4 of them.

However, the K and f_{int} calculation is not trivially parallelizable yet: the process of adding the element forces $f_{int}^{(i)}$ and stiffness matrices $K^{(i)}$ to the global vector/matrix can cause some trouble. More specifically, we can do some nonsense if we want to add to *the same element in the same time* in the global matrix/vector.

8. Implementing a Finite Element Simulator

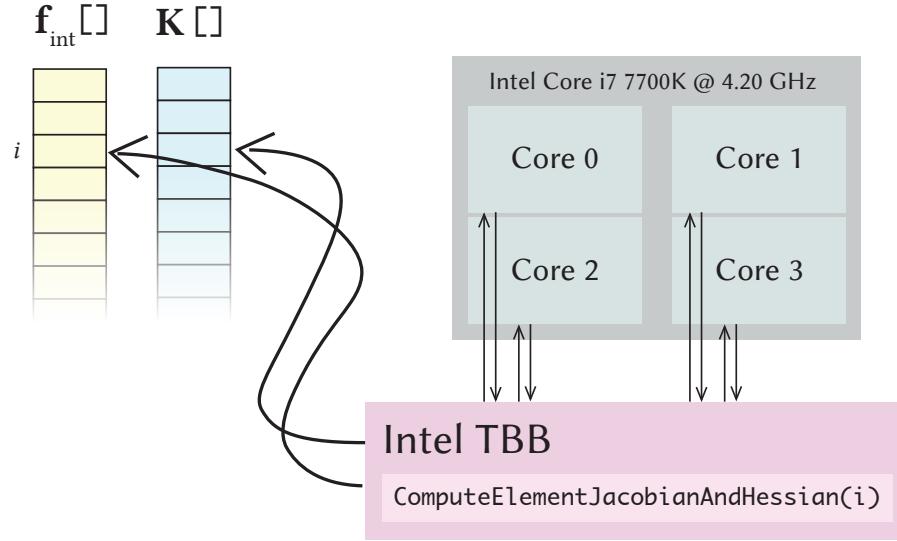


Figure 8.2: Naive parallelization of building \mathbf{f}_{int} and \mathbf{K}

The trivial solution to this problem would be, that we collect the element forces $\mathbf{f}_{int}^{(i)}$ and stiffness matrices $\mathbf{K}^{(i)}$ in an array, storing a \Re^{12} vector and a $\Re^{12 \times 12}$ matrix for each element. Doesn't sound that complicated, right? The process is visualized on Fig. 8.2. Then in the second step, is when add these element forces/stiffness matrices to the final, global system.

First we need to add these new arrays to the Solver class:

```
class Solver {
    // ...
    // parallel Keff building resources
    std::vector<Vec12> fIntArray;
    std::vector<Mat12> KelArray;
};
```

Then the method `ComputeElementJacobianAndHessian(i)` is modified as:

```
Solver::ComputeElementJacobianAndHessian(int i)
{
    // calculate F ...
    {
        // ...
        // calculate forces ...
        const Vec12 fEl = -tetVol * dfdx.transpose() * Pv;

        // instead of adding directly to the global f_int,
        // we assign it to the appropriate array element
        fIntArray[i] = fEl;
    }
}
```

8. Implementing a Finite Element Simulator

```
{
    // ...
    // get Jacobian
    const Mat12 dPdx = -tetVol * dFdx.transpose() * dPdF * dFdx;

    // do the same with K
    KelArray[i] = dPdx;
}
}
```

Then in the `Step()` method, we just replace the regular `for` with a `tbb::parallel_for`, provided by Intel's Threading Building Blocks interface (TBB). Yes, it's that easy – since we made the problem trivially parallelizable.

```
Vec Solver::Step()
{
    // setup ...
    for(;;)
    {
        tbb::parallel_for(
            tbb::blocked_range<size_t>(0, numElements),
            [=](const tbb::blocked_range<size_t>& r)
        {
            for (size_t i = r.begin(); i != r.end(); ++i)
                ComputeElementJacobianAndHessian(i);
        }
    );
    // ...
}
```

Then we just need to accumulate the content of the $\mathbf{f}_{int}^{(i)}$ and $\mathbf{K}^{(i)}$ vectors

```
FillFint();
FillKeff();
```

How much faster is this? E.g. in the turtle example (see Sec. 9.1 – it's 1185 elements) the assembling the global matrix/vector took ~0.38s without parallelization, while the parallelized version took ~0.1s. Wow! That's a ~3.8x performance increase with a small tweak! I love technology!

8.3 Drawing the Results On The Screen

Calling the `Step()` method returns \mathbf{x} , the current position of all the vertices. If a step would took less than $1/60\text{s} = 0.0167\text{s}$ we could animate the model in real-time, but this is not the case unfortunately. Even Eigen's conjugate gradient routine of e.g. the turtle example's (Sec. 9.1) $\sim 1000 \times 1000$ matrix takes about 0.05s, so we would need a significantly different application for that.

8. Implementing a Finite Element Simulator

Let's just cook from what we have. In my opinion, a 0.15s long timestep is not that bad, so the application I built around this solver is accumulates all the `xs` in a vector, and we can '*play*' this – using an industry term – '*baked*' animation.

8.3.1 A Basic Real-Time Graphics Application

We have this `std::vector<x> positionArray`, how to draw it to the screen? Probably DirectX 12 is not the best solution for this, because it's a low-level graphics API, and we have to do stuff which may seem unnecessary, but I have a little engine written in DirectX already, so I just used that. We again have to do the same things we did with the solver. First we initialize the app, then define what should be done when a frame is drawn.

initialization looks like this:

- Open window
- Parse config
- Start subsystems: App, Renderer and Simulator.

The App provides us a lil' GUI (`ImGui`), where we can manipulate stuff about the app, e.g. choose the frame we want to display. We then start the Simulator system. It is basically a thin wrapper around a `Solver` instance, which prepares the model for display and possesses the array of the resultant `xs`.

The Renderer does the rendering. In Sec. 2.1 I talked a bit what you need to render stuff. On initialization, the Renderer creates and uploads some resources on the GPU:

- Geometry: mesh defined by a list of vertices and a list of triangles.
Data at each vertex: position, *normal*, UV coordinates, etc.
- Texture(s): *Per-pixel* color – or any other – data mapped on the triangles
- Constant Buffer(s): *Per-object*, global data. E.g. model's position, rotation, etc.
- Shader(s): GPU code for shading each object
- Abstract resources for shading: camera and light sources

We push all this stuff on the GPU on startup.

8. Implementing a Finite Element Simulator

As I said our solver is *not* running at 60 fps, but around 6-7 fps. To be able to pleasantly move around the model, we need to step a simulation on a separate thread, while the main, rendering loop just displays the result. Stepping the simulation on a separate thread looks like this:

```
void Simulator::Step() {
    // the simulator class calls the solver's method to do a timestep
    Vec currentPosition = solver.Step();
    // and stores the result
    positionArray.push_back(currPos);
    stepNum++;
}

void StepSimulator() {
    // The Simulator is running on a separate thread forever
    // not bothering with any of the app/rendering stuff going on
    while(1) {
        sim.Step();
    }
}

// in main():
// creating the thread
std::thread simThread{ StepSimulator }
```

Now the **main loop** looks like this:

- Update the App's state
- Update the Simulator.
For the currently selected time step:
 1. Get mesh vertices
 2. Recalculate normals
 3. Upload mesh

- Draw Scene

What is up with that normal recalculation tough?

8.3.2 Drawing the Model

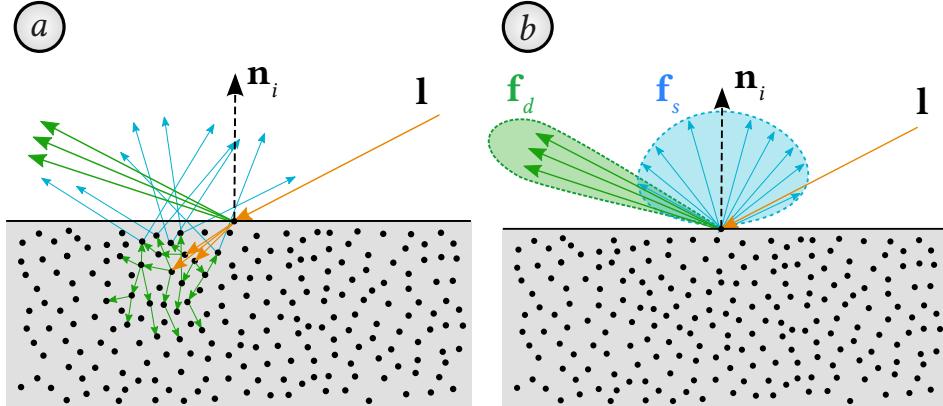


Figure 8.3: a – Photons interacting with a surface. b – Model applied in rendering. [22]

To color the triangle we need to integrate the rendering equation [13]. There are smart ways to do that, but I’m not going to talk about it, as this thesis is already far too long. If you’re interested, the best thing you can do is checking out the BME BSc lecture on Computer Graphics.

What we are essentially doing is visualized on Fig. 8.3. We simulate the light’s interaction with a surface. What’s important now is you need the position and the *normal vector* of the surface in order to determine its color. We just wrote a huge application which determines the position of a deformed surface, but what about the normal vector? We need to compute the normal vector at each vertex each time a frame is drawn. There is a nice trick to this I want to show you now.

Triangle Mesh Normal Vector Calculation Hack

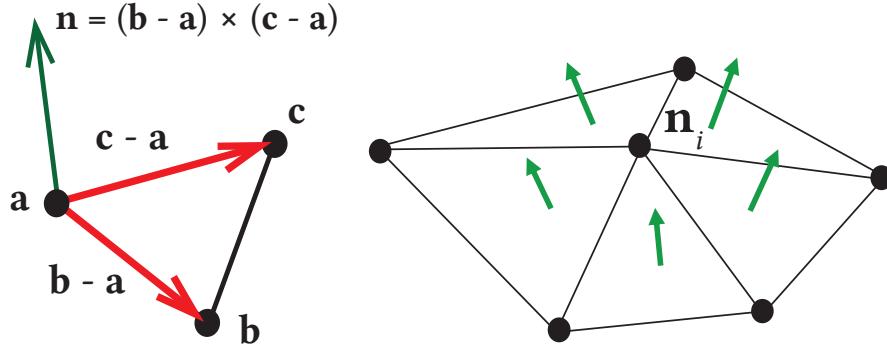


Figure 8.4: Computation of the normal vector.

This is really just a hack: first, initialize an array of 3 component vectors to zero for each triangle vertex. Then for each triangle, calculate the normal vector of the triangle

$$\mathbf{n}_i = (\mathbf{b} - \mathbf{a}) \times (\mathbf{c} - \mathbf{a}) \quad (8.12)$$

then calculate the triangle area

$$A_i = \frac{1}{2} \|\mathbf{n}_i\| \quad (8.13)$$

then '*normalize*' it with its area, so bigger triangles count more:

$$\mathbf{n}_i *= \frac{1}{2A_i} \quad (8.14)$$

then add this normal \mathbf{n}_i to each vertex the triangle is '*made of*'. You will end up with a nice approximation of the surface normals at the vertices. See Fig. 8.4 for visualization.

Chapter 9

Case Studies

9.1 Testing Rig, Example Models

Let's try our brand new simulator! The rig I used for the simulation is an average gaming PC from early 2017: 14 nm, quad core, hyperthreaded Intel 7700K running at 4.20 GHz. The cooling solution was cleaned a long-long time ago, which might contribute to the result. The simulator is a fully CPU side app, so the GPU doesn't matter, but it was a GeForce 1060 3GB.

I used 3 volumetric mesh examples from the Vega FEM library [4]: the 'turtle', (Fig. 9.1a) the '*cantilever plate*', (Fig. 9.1b) and the '*asian dragon*'. (Fig. 9.1c) You can see the fixed and the loaded vertices on Fig. 9.1a and b. For all the simulations, I applied some force incrementally with the `Interpolator` class till a specified f_{max} , then I sortof '*released*' it, just like in the loading situation on Fig. 8.1.

9.2 Testing the Core Algorithm

Believe it or not, the simulator presented in the previous chapter... works! You can either go down the Cauchy-Green way (Eq. 5.44) or the Smith et al. way (Eq 5.109), you just need to derive the scalar derivative of any `EnergyFunction` you find in the literature. However, without the Hessian projection, as of Chapter 6, you need some sufficiently small timesteps.

I tested the core algorithm on the *Saint Venant-Kirchhoff, Stable Neo-Hookean* [24] and of course *ARAP* material models. You can see the results on Fig. 9.1a and b. All of them worked pretty well, with *sufficiently small timesteps*.

9.3 Projected Newton Solver

With the analytical eigensystem of the ARAP energy, presented in Sec. 6.4, we can easily implement the projected Newton solver from [25]. Does it work? **drumrolls** It does!

As you can see on Fig. 9.1c, crushing the asian dragon model to a plane doesn't cause the simulation to explode, it rather successfully recovers to its rest state. On the other hand, if you delete lines doing the *Hessian projection*, the simulation *does* explode. I consider this an *epic win*.

9.4 Dynamics Simulations: What?

This is definitely the peak of sloppiness in this thesis, but I just can't figure out what – *if anything* – is wrong with the dynamic integration of my solver. First, I definitely should've paid more attention to the amazing vibration and dynamics lectures at the university. That's probably the biggest mistake.

The issue I have is that compared the quasistatic case (Sec. 8.2.6), dynamic integration with implicit Newmark or Backward Euler seems infinitely less stable to me. It sounds okay when I write this down but shouldn't implicit integration allow larger timesteps? It allowed *way larger* timesteps than the explicit method i tried, but compared to the same time integration in the Vega FEM library, it was 10 – 1000 times slower. Weird! Altough with sufficiently small timesteps, it was working, which is nice. And suprisingly, the backward Euler seemed a bit more stable for me.

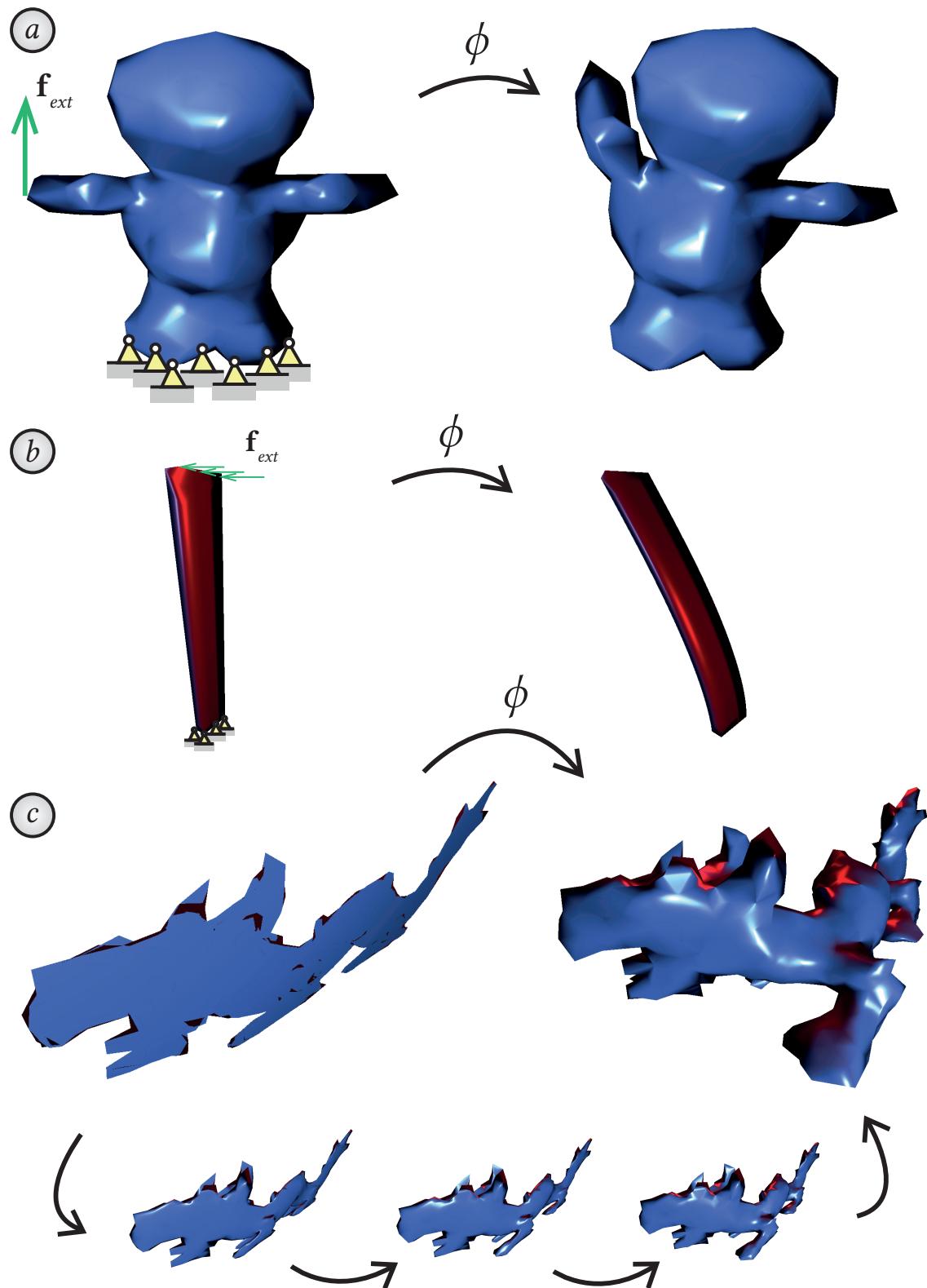


Figure 9.1: Results.

Chapter 10

Summary

In this thesis, after some brief introduction to the basics of computer graphics, FEM, and continuum mechanics, we looked at the craftsmanship of deformable simulators, mostly following the work of Smith et al. in [25] and [15]. Then, as an '*illustration*' of all the knowledge I collected from the literature, I implemented a sample simulator in C++, then tried it out with some example models to arrive at a pretty sloppy summary you read in the previous chapter.

I definitely had fun writing this thesis! All the work/research I've done brought not just FEM and continuum mechanics, but linear algebra and in general, math, much-much closer to me.

What's next? Here are the things I want to in the not so distant future:

1. The first thing I want to do is to make the simulator *faster*. Probably you can already find an updated version at marcisolti.github.io/ifem.
2. After that it would be really nice to look into the different *constitutive models*,
3. and what happens to them if plug them into the *analytical eigendecomposition*.
4. The thing that also bugs me is the dynamic integration schemes' *stability*. What's up with that? I surely didn't spend enough time with experimenting, and I should try other schemes as well.

Chapter A

An Introductory Example of FEM

Let's look at a nice example for FEM from [5]. Fig. A.1. shows a system of three rigid carts on a horizontal plane that are interconnected by a system of linear elastic springs. Calculate the displacements u_i of the carts for the f_i loading shown.

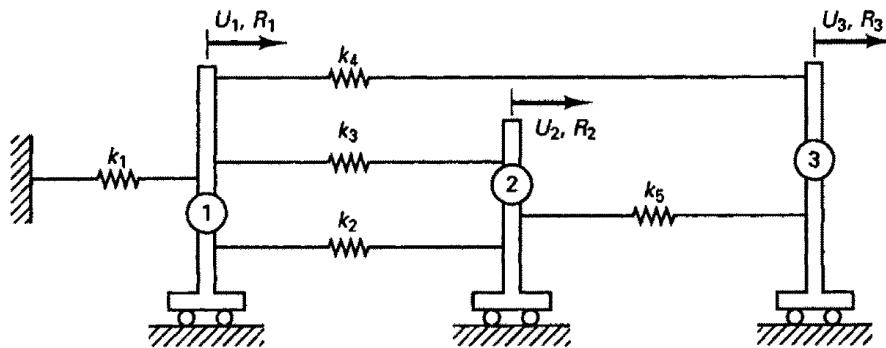


Figure A.1: Physical layout of the problem

A.1 System Idealization

We don't need to do much to *idealize this system*. The problem description already stated that the springs are *linear elastic* and the carts are *rigid*. The system is the assemblage of these idealized components – or *elements*.

A.2 Element Equilibrium

After *system idealization*, the next step is to establish the *equilibrium at each element i* in terms of the *state variables*. An *element* connects a number of points – called *nodes* –

A. An Introductory Example of FEM

together in space. The *state variables* of a *node* is usually refers to its *spatial position* and the *forces acting on it*.

As the carts are in the horizontal plane, it is straightforward to assume that it can only move horizontally. Hence, our state variables are the u_i s and the f_i s, as shown on Fig. A.1. The *structural elements* will be the *linear elastic springs* connecting the carts.

A.2.1 Linear Elastic Spring as a Structural Element

The response of a linear elastic spring, that has one of its end attached to some fixed object, while the other end is pulled by some force of magnitude f , is calculated according to Hooke's law as

$$ku = f \quad (\text{A.1})$$

Here the response is the displacement of the pulled end, denoted by u , while k is the *spring constant*. k is a positive, real number, completely characterizing this ideal spring. See Fig. A.2 for reference.

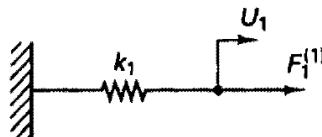


Figure A.2: A linear elastic spring with single degree of freedom

In this case, f can be thought of as some external '*energy*' which '*perturbs*' the system from its steady state. As a response, the system '*stores*' this energy in the spring itself, by equilibrating f with ku , hence the equation $ku = f$. In our problem, spring No. 1 is such a spring.

For the other springs in the system, both ends are free, so we have state variables u_1 , u_2 and f_1 , f_2 . See Fig. A.3.

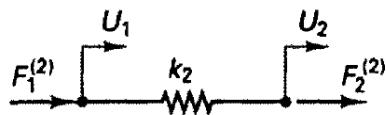


Figure A.3: A linear elastic spring with two degrees of freedom

The equilibrium now is a *system of equations*

$$\begin{aligned} k(u_1 - u_2) &= f_1 \\ k(-u_1 + u_2) &= f_2 \end{aligned} \quad (\text{A.2})$$

A. An Introductory Example of FEM

which can be reformulated as

$$k \begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} = \begin{bmatrix} f_1 \\ f_2 \end{bmatrix} \quad (\text{A.3})$$

We just defined the element equilibrium of a *1D linear elastic truss element!* In this equation,

$$k \begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix} = \mathbf{K}^{(i)} \quad (\text{A.4})$$

is the i^{th} element's *stiffness matrix*. Actually, Eq. A.3 and Eq. A.1 are the very same equation but with $u_1 = 0$ and $u_2 = u$. Nice!

Now we can go back to Fig. A.1 and define the element equilibrium for each element explicitly. The result can be seen on Fig. A.4.

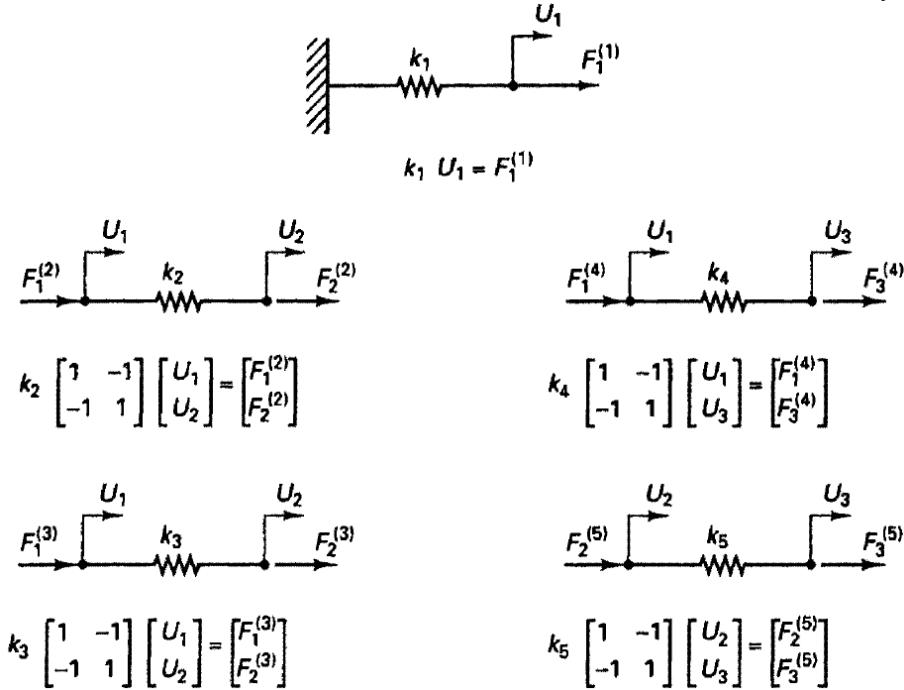


Figure A.4: Equilibrium at each element

A.3 Element Assemblage, Solution

To progress further, first we need to expand the element stiffness matrices to the global degrees of freedom. E.g. for the first element this means

$$\begin{bmatrix} k_1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \end{bmatrix} = \begin{bmatrix} f_1^{(1)} \\ 0 \\ 0 \end{bmatrix} \quad (\text{A.5})$$

A. An Introductory Example of FEM

or

$$\mathbf{K}^{(1)} \mathbf{u} = \mathbf{f} \quad (\text{A.6})$$

while for the second element

$$\begin{bmatrix} k_2 & -k_2 & 0 \\ -k_2 & k_2 & 0 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \end{bmatrix} = \begin{bmatrix} f_1^{(2)} \\ f_2^{(2)} \\ 0 \end{bmatrix} \quad (\text{A.7})$$

or similarly

$$\mathbf{K}^{(2)} \mathbf{u} = \mathbf{f} \quad (\text{A.8})$$

and so on.

To generate the governing equations for the state variables of the complete system, we need to invoke the element interconnection requirements. This will yield the *global stiffness matrix* \mathbf{K} , which completely characterizes the *whole system's response*. In contrast, the *element stiffness matrices* characterized the *elements' response*.

The ultimate method for constructing the global stiffness matrix \mathbf{K} is the *direct stiffness method*, but let's just not go down that road *yet*. Instead, consider the static equilibrium for each cart: $\sum f = 0$ According to the figure, this yields

$$\begin{aligned} f_1^{(1)} + f_1^{(2)} + f_1^{(3)} + f_1^{(4)} &= f_{\text{ext}1} \\ f_2^{(2)} + f_2^{(3)} + f_2^{(5)} &= f_{\text{ext}2} \\ f_3^{(4)} + f_3^{(5)} &= f_{\text{ext}3} \end{aligned} \quad (\text{A.9})$$

In other words, the sum of all spring's (j) forces $f_i^{(j)}$ acting on each cart (i) are equal to the external forces $f_{\text{ext}i}$ introduced to the system.

We can now substitute the equilibriums of the element stiffness matrices in the interconnection requirements for the element end forces. With some symbolic machete slashing, this reduces to the fundamental equation of the Finite Element Method:

$$\boxed{\mathbf{K}\mathbf{u} = \mathbf{f}_{\text{ext}}} \quad (\text{A.10})$$

where

$$\mathbf{K} = \begin{bmatrix} (k_1 + k_2 + k_3 + k_4) & -(k_2 + k_3) & -k_4 \\ -(k_2 + k_3) & (k_2 + k_3 + k_5) & k_5 \\ -k_4 & -k_5 & (k_4 + k_5) \end{bmatrix} \quad (\text{A.11})$$

Here, \mathbf{K} the global stiffness matrix, which completely characterizes the response of the system. Now you solve $\mathbf{K}\mathbf{u} = \mathbf{f}_{\text{ext}}$ for \mathbf{u} to acquire the displacements in case of a specific loading condition \mathbf{f}_{ext} .

A.4 Other Lumped-Paramter Models

It has to be noted that the lumped-parameter mathematical model can be used in many other areas of physics.

Two classic examples of that are the steady-state pressure and flow distributions in the hydraulic network shown in Fig. A.4, and the steady-state current distributions in a DC network, as of Fig. A.4. In both cases, you can write the system of equation for each element for the currents entering and leaving the element at the ends, then use the very same assembly process to get the global system of equations.

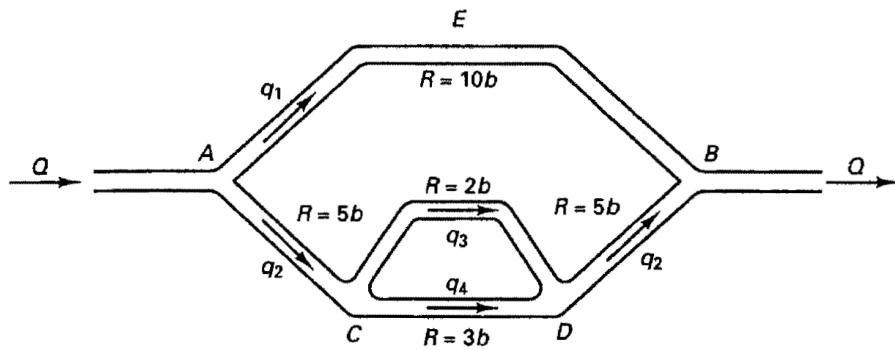


Figure A.5: Pipe network

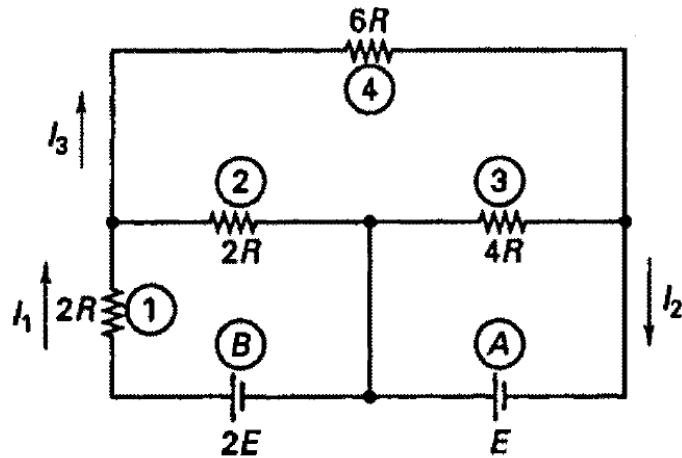


Figure A.6: DC network

Chapter B

The Method of Steepest Descent

B.1 What is a Linear System, Anyways?

Let's solve a linear system iteratively, following [23]. Consider the following linear system – illustrated on Fig B.1:

$$\mathbf{Ax} = \mathbf{b} \quad (\text{B.1})$$

$$\mathbf{A} = \begin{bmatrix} 3 & 2 \\ 6 & 6 \end{bmatrix} \quad \mathbf{b} = \begin{bmatrix} 2 \\ -8 \end{bmatrix} \quad (\text{B.2})$$

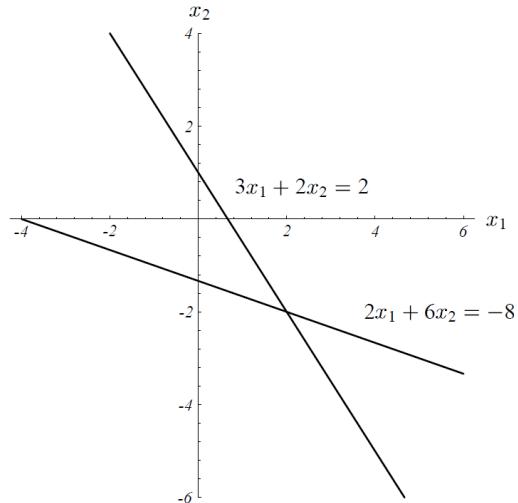


Figure B.1: Sample two-dimensional linear system. The solution lies at the intersection of the lines.

B. The Method of Steepest Descent

The *quadratic form* of such system is simply a scalar, quadratic function of a vector with the form

$$f(\mathbf{x}) = \frac{1}{2} \mathbf{x}^T \mathbf{A} \mathbf{x} - \mathbf{b}^T \mathbf{x} + c \quad (\text{B.3})$$

where c is a scalar constant, $c = 0$ in our case. The graph and the contour plot of the *quadratic form* can be seen on Fig. B.2 – a picture is indeed worth thousand words.

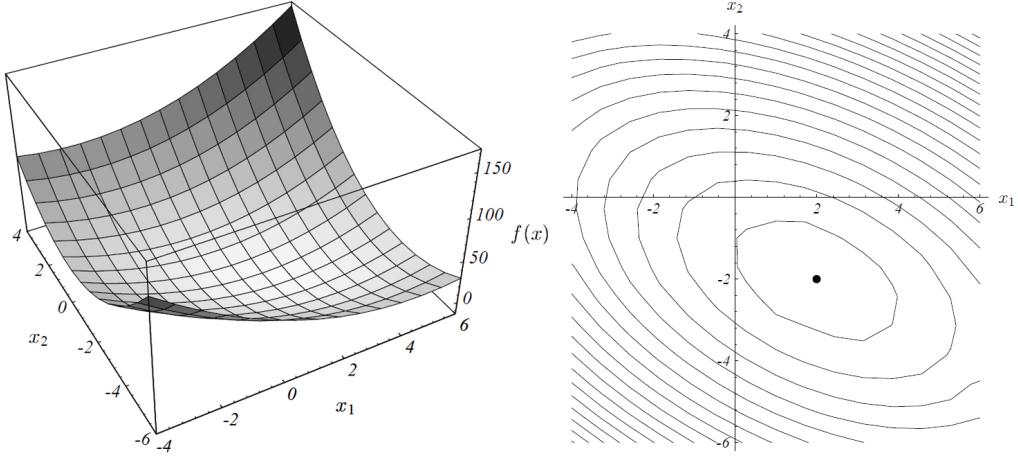


Figure B.2: a – Graph of a quadratic form $f(\mathbf{x})$. The minimum point of this surface is the solution to $\mathbf{Ax} = \mathbf{b}$. b – Contours of the quadratic form. Each ellipsoidal curve has constant $f(\mathbf{x})$

The *gradient* of a quadratic form is defined as

$$\mathbf{f}'(\mathbf{x}) = \begin{bmatrix} \frac{\partial}{\partial x_1} f(\mathbf{x}) \\ \frac{\partial}{\partial x_2} f(\mathbf{x}) \\ \vdots \\ \frac{\partial}{\partial x_n} f(\mathbf{x}) \end{bmatrix} \quad (\text{B.4})$$

This is visualized at Fig. B.3. With some tedious math, combining Eq. B.3 and B.4 yields

$$\mathbf{f}'(\mathbf{x}) = \frac{1}{2} \mathbf{A}^T \mathbf{x} + \frac{1}{2} \mathbf{A} \mathbf{x} - \mathbf{b} \quad (\text{B.5})$$

and if \mathbf{A} is symmetric, this further reduces to

$$\mathbf{f}(\mathbf{x}) = \mathbf{Ax} - \mathbf{b} \quad (\text{B.6})$$

Setting the gradient to zero, we obtain the original linear system we want to solve. Isn't this wonderful? If \mathbf{A} is positive-definite and symmetric, then solving $\mathbf{Ax} = \mathbf{b}$ for \mathbf{x} equals minimizing $f(\mathbf{x})$. Other cases are illustrated on Fig. B.4

B. The Method of Steepest Descent

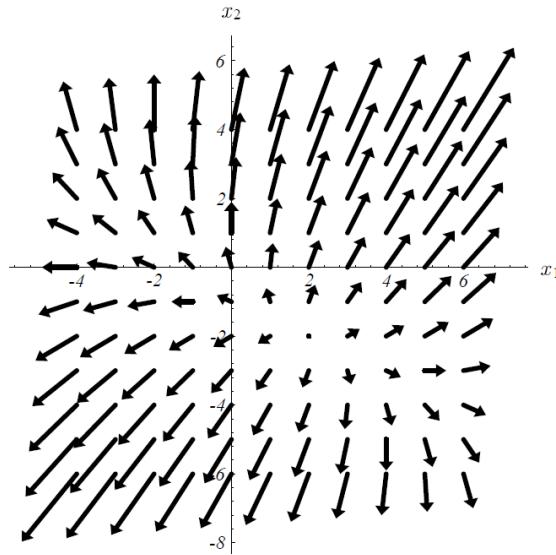


Figure B.3: Gradient $f'(\mathbf{x})$ of the quadratic form. For every x_i , the gradient points in the steepest increase of $f(\mathbf{x})$, and is orthogonal to the contour lines.

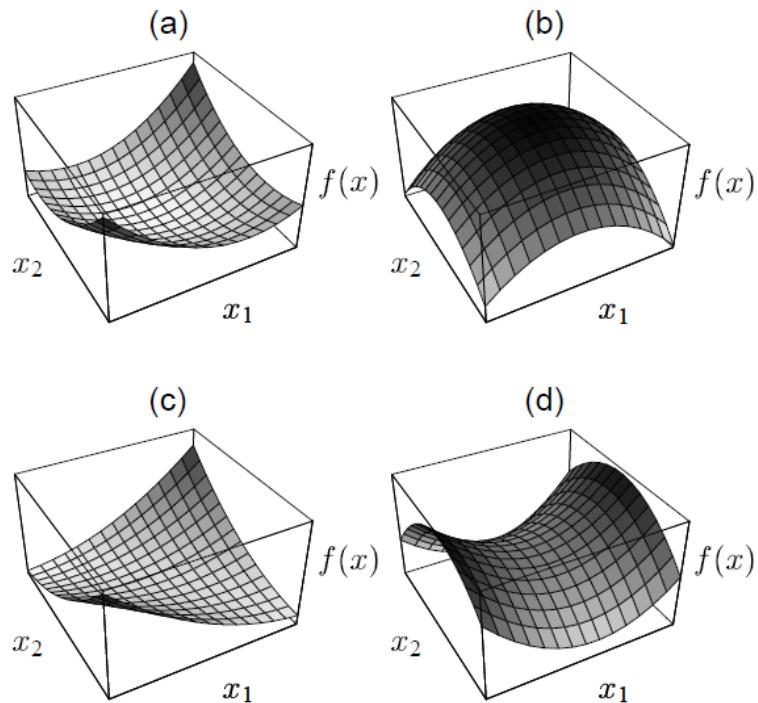


Figure B.4: a – quadratic form for a positive definite matrix. b – for a negative-definite matrix. c – for a singular (and positive-indefinite) matrix. A line that runs through the bottom of the valley is the set of solutions. d – For an indefinite matrix. Because the solution is a saddle point, Steepest Descent will not work. In three dimensions or higher, a singular matrix can also have a saddle.

In my opinion Fig. B.4 demonstrates pretty well why it is so important that A have to be positive- (or negative-) definite for an iterative method to work. You probably have an intuition already what we are going to do next.

B.2 The Method of Steepest Descent

Next up: apply the method of Steepest Descent to solve the system of equations in discussion. As I stated in the introduction of the chapter, iterative method needs some initial value $\mathbf{x}^{(0)}$ as a starting point. We can choose this randomly now, but in case of a real, n -dimensional problem, we need to choose one near the equilibrium.

We have now this arbitrary starting point; in the method of Steepest Descent we slide down to the bottom of the paraboloid, by taking a series of steps $\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \dots$ until we are satisfied that we are close enough to the solution.

When we take a step, we choose the direction in which f decreases most quickly, which is the direction opposite $f'(\mathbf{x}^{(i)})$. According to Eq. B.6, this direction is $-f'(\mathbf{x}^{(i)}) = \mathbf{b} - \mathbf{A}\mathbf{x}^{(i)}$.

This direction is so important, that it has its own name... and we already heard about it! The residual $\mathbf{r}^{(i)} = \mathbf{b} - \mathbf{A}\mathbf{x}^{(i)}$ indicates how far we are from the correct value of \mathbf{b} ; so whenever you read/hear *residual* think of it as the *direction of steepest descent*.

Suppose we start at $\mathbf{x}^{(0)} = [-2, -2]^T$ Our first step, along the direction of steepest descent, will fall somewhere on the solid line in Fig. B.5a In other words, we will choose a point

$$\mathbf{x}^{(1)} = \mathbf{x}^{(0)} + \alpha \mathbf{r}^{(0)} \quad (\text{B.7})$$

The question is, how big a step should we take?

A line search is a procedure that chooses α to minimize α along a line. Fig. B.5b illustrates this task: we are restricted to choosing a point on the intersection of the vertical plane and the paraboloid. Fig. B.5c is the parabola defined by the intersection of these surfaces. What is the value of α the base of the parabola?

Of course, we know from basic calculus that α minimizes f when its directional derivative is zero – see Fig. B.5d. Note that $\mathbf{r}^{(0)}$ and $f'(\mathbf{x}^{(1)})$ are orthogonal.

The example is run until it converges in Fig. B.6. Not bad, however, note the zigzag path, which appears because each gradient is orthogonal to the previous gradient. Wouldn't it be better if, every time we took a step, we got it right the first time? Sure it would and the method of conjugate gradients does that. If you've got excited, read the original paper: [23].

B. The Method of Steepest Descent

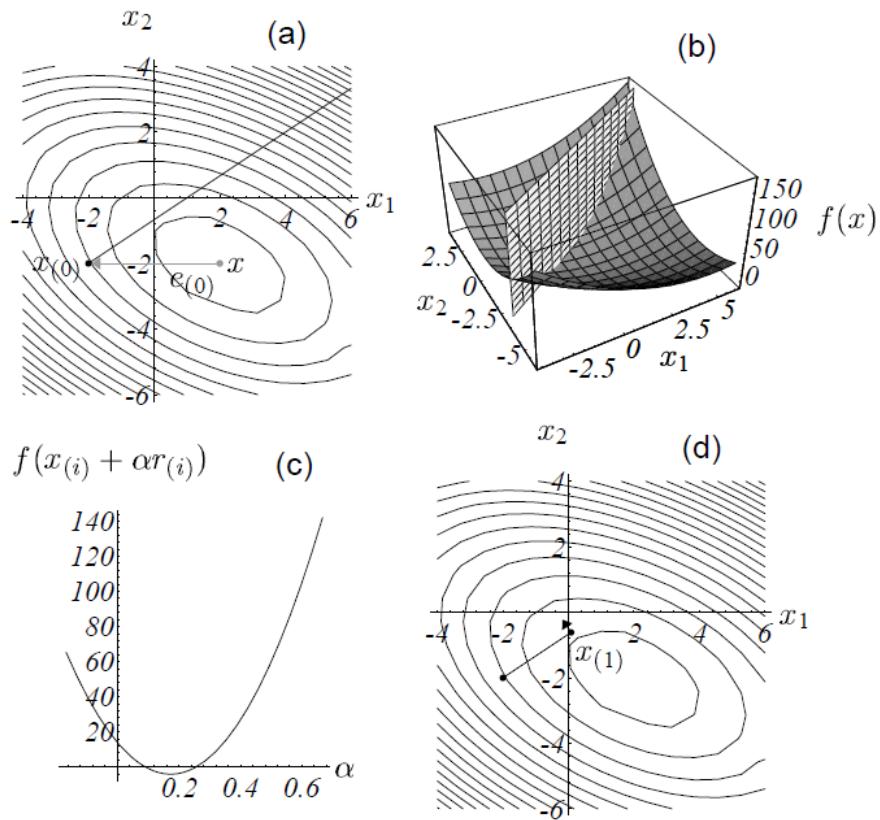


Figure B.5: The method of Steepest Descent. a – Starting at $[-2, -2]^T$, take a step in the direction of steepest descent of f . b – Find the point on the intersection of these two surfaces that minimizes f . c – This parabola is the intersection of surfaces. The bottommost point is our target. d – The gradient at the bottommost point is orthogonal to the gradient of the previous step.

B. The Method of Steepest Descent

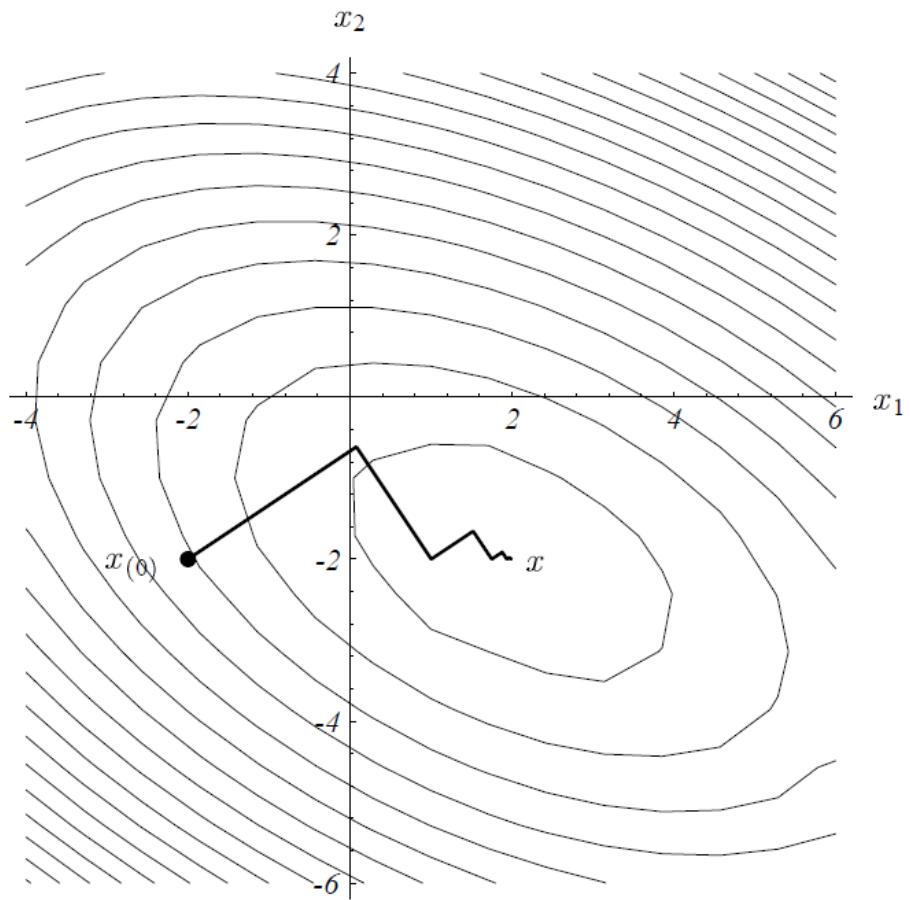


Figure B.6: The method of Steepest Descent starts at $[-2, -2]^T$ and converges at $[2, -2]^T$.

Chapter C

A Config File

```
{  
    // ... other stuff about the app  
    "sim": {  
        "model": "turtle",  
        "material": {  
            "energyFunction": "ARAP",  
            "E": 20.0,  
            "nu": 0.2,  
            "rho": 1000.0,  
            "alpha": 0.01,  
            "beta": 1.0  
        },  
        "integrator": "qStatic",  
        "stepSize": 0.01,  
        "magicConstant": 1.0,  
        "numSubsteps": 1,  
        "initConfig": "r",  
        "fixedVerts": [ 1, 3, 6, ... ],  
        "loadedVerts": [ 890, ... ]  
        "loadSteps": [  
            {  
                "f": 0.0,  
                "t": 0.0  
            },  
            {  
                "f": 1000.0,  
                "t": 1.0  
            },  
            ...  
        ]  
    }  
}
```

Bibliography

- [1] Pierre Alliez, David Cohen-Steiner, Mariette Yvinec, and Mathieu Desbrun. Variational tetrahedral meshing. *ACM Transactions on Graphics*, 24, 07 2005.
- [2] David Baraff and Andrew Witkin. Large steps in cloth simulation. In *Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '98, page 43–54, New York, NY, USA, 1998. Association for Computing Machinery.
- [3] Jernej Barbic, Fun Shing Sin, and Daniel Schroeder. Vega FEM Library, 2012. <http://www.jernejbarbic.com/vega>.
- [4] Jernej Barbic, Fun Shing Sin, and Daniel Schroeder. Vega FEM Library, 2012.
- [5] K.J. Bathe. *Finite Element Procedures*. Prentice Hall, 2006.
- [6] Javier Bonet and Richard D. Wood. *Nonlinear Continuum Mechanics for Finite Element Analysis*. Cambridge University Press, 2 edition, 2008.
- [7] E.G. Gilbert, D.W. Johnson, and S.S. Keerthi. A fast procedure for computing the distance between complex objects in three-dimensional space. *IEEE Journal on Robotics and Automation*, 4(2):193–203, 1988.
- [8] Gaël Guennebaud, Benoît Jacob, et al. Eigen v3. <http://eigen.tuxfamily.org>, 2010.
- [9] Gerhard A. Holzapfel. *Nonlinear Solid Mechanics: A Continuum Approach for Engineering Science*. Wiley, 2002.
- [10] Kai Hormann and Guenther Greiner. Mips: An efficient global parametrization method. *Curve and Surface Design: Saint-Malo*, 2000:10, 11 2012.
- [11] Bussiness Insider. How pixar's movement animation became so realistic, <https://www.youtube.com/watch?v=qbhsmld9hb0>.

C. Bibliography

- [12] G. Irving, J. Teran, and R. Fedkiw. Invertible finite elements for robust simulation of large deformation. In *Proceedings of the 2004 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '04, page 131–140, Goslar, DEU, 2004. Eurographics Association.
- [13] James T. Kajiya. The rendering equation. In *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '86, pages 143–150, New York, NY, USA, 1986. ACM.
- [14] Avneet Kaur, Maryann Simmons, and Brian Whited. Hierarchical controls for art-directed hair at disney. In *ACM SIGGRAPH 2018 Talks*, SIGGRAPH '18, New York, NY, USA, 2018. Association for Computing Machinery.
- [15] Theodore Kim and David Eberle. Dynamic deformables: Implementation and production practicalities. In *ACM SIGGRAPH 2020 Courses*, SIGGRAPH '20, New York, NY, USA, 2020. Association for Computing Machinery.
- [16] Tiantian Liu, Adam W. Bargteil, James F. O'Brien, and Ladislav Kavan. Fast simulation of mass-spring systems. *ACM Trans. Graph.*, 32(6), November 2013.
- [17] Niels Lohmann. <https://github.com/nlohmann/json>.
- [18] Aleka McAdams, Yongning Zhu, Andrew Selle, Mark Empey, Rasmus Tamstorf, Joseph Teran, and Eftychios Sifakis. Efficient elasticity for character skinning with contact and collisions. *ACM Trans. Graph.*, 30(4), July 2011.
- [19] Matthias Müller, Bruno Heidelberger, Marcus Hennix, and John Ratcliff. Position based dynamics. *J. Vis. Comun. Image Represent.*, 18(2):109–118, April 2007.
- [20] Matt Pharr, Wenzel Jakob, and Greg Humphreys. *Physically Based Rendering: From Theory to Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3rd edition, 2016.
- [21] Science Behind Pixar. <https://sciencebehindpixar.org/pipeline/rigging>.
- [22] Charles de Rousiers Sebastien Lagarde. Moving frostbite to pbr.
- [23] Jonathan R Shewchuk. An introduction to the conjugate gradient method without the agonizing pain. Technical report, USA, 1994.
- [24] Breannan Smith, Fernando De Goes, and Theodore Kim. Stable neo-hookean flesh simulation. *ACM Trans. Graph.*, 37(2), March 2018.

C. Bibliography

- [25] Breannan Smith, Fernando De Goes, and Theodore Kim. Analytic eigensystems for isotropic distortion energies. *ACM Trans. Graph.*, 38(1), February 2019.
- [26] Breannan Smith, Chenglei Wu, He Wen, Patrick Peluse, Yaser Sheikh, Jessica Hodgins, and Takaaki Shiratori. Constraining dense hand surface tracking with elasticity. *ACM Transactions on Graphics (TOG)*, 39(6), December 2020.
- [27] Olga Sorkine and Marc Alexa. As-rigid-as-possible surface modeling. In *Proceedings of the Fifth Eurographics Symposium on Geometry Processing*, SGP ’07, page 109–116, Goslar, DEU, 2007. Eurographics Association.
- [28] O. Sorkine-Hornung and M. Rabinovich. *Least-squares rigid motion using SVD*. ETH Zurich, Department of Computer Science, 2017.
- [29] Alexey Stomakhin, Russell Howes, Craig Schroeder, and Joseph M. Teran. Energetically consistent invertible elasticity. In *Proceedings of the 11th ACM SIGGRAPH / Eurographics Conference on Computer Animation*, EUROSCA’12, page 25–32, Goslar, DEU, 2012. Eurographics Association.
- [30] Alexey Stomakhin, Craig Schroeder, Lawrence Chai, Joseph Teran, and Andrew Selle. A material point method for snow simulation. *ACM Trans. Graph.*, 32(4), July 2013.
- [31] Dr. Andras Szekrenyes. *FEA TM08: Newmark time integration scheme*. BME-MM, 2020.
- [32] Rasmus Tamstorf, Toby Jones, and Stephen F. McCormick. Smoothed aggregation multigrid for cloth simulation. *ACM Trans. Graph.*, 34(6), October 2015.
- [33] Joseph Teran, Eftychios Sifakis, Geoffrey Irving, and Ronald Fedkiw. Robust quasistatic finite elements and flesh simulation. In *Proceedings of the 2005 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA ’05, page 181–190, New York, NY, USA, 2005. Association for Computing Machinery.
- [34] Demetri Terzopoulos, John Platt, Alan Barr, and Kurt Fleischer. Elastically deformable models. *SIGGRAPH Comput. Graph.*, 21(4):205–214, August 1987.
- [35] Hongyi Xu, Funshing Sin, Yufeng Zhu, and Jernej Barbič. Nonlinear material design using principal stretches. *ACM Trans. Graph.*, 34(4), July 2015.
- [36] O.C. Zienkiewicz and R.L Taylor. *Finite Element Method: Volume 1 - The Basis*. Butterworth-Heinemann, Oxford, 5th edition, 2000.

List of Figures

2.1	A typical <i>scene</i>	3
2.2	Skeletal animation [21]	4
2.3	Examples of Physically Based Animation: a – Material Point Method for snow simulation [30]; b – Implicit backward Euler formulation for cloth simulation [2]; c – Stable Neo-Hookean flesh simulation for character animation [24]; d – Art-directed hair simulation [14]; e – Constraining dense hand surface tracking with elasticity for VR hands [26];	7
3.1	a - Calculating the response of a deformable object, one end is built in to the wall, while the other end is loaded with some forces. b - The problem is idealized as an assemblage of brick elements (c). d - the response of the assemblage of elements – the movement of the all the nodes, ϕ – is calculated according to the loads/boundary conditions.	8
3.2	A linear elastic spring with single degree of freedom	10
3.3	Different Structural Elements	11
3.4	The general pattern: a – The system is an assemblage of structural elements. b – Element stiffness matrices are formulated. c – The element stiffness matrices are accumulated to the global stiffness matrix by adding the element matrix to the corresponding degrees of freedom, according to the element interconnection. [36]	13
3.5	Mass-Spring Simulation of cloth. [16]	14
3.6	The Newton-Raphson scheme.	17
3.7	As-Rigid-As-Possible surface modelling. [27]	23
3.8	Optimizing surface parametrization: Initial surface parametrization (<i>a</i>) results poor utilization of the per-pixel color data (<i>b</i>) a.k.a. texture of the surface. Optimizing for consistent ' <i>deformation energy</i> ' Ψ achieves the desired paraetrisation automatically (<i>c & d</i>). [25]	24

C. List of Figures

4.1	General motion of the body	26
4.2	The Deformation Gradient	27
4.3	Singular value decomposition of some matrix A.	30
4.4	The concept of stress in the <i>material</i> coordinate system.	31
5.1	Thinking about 3 rd -order tensors as a vector of matrices.	37
5.2	Thinking about 4 th -order tensors as a matrix of matrices.	42
5.3	I_C measures the overall edge lengths of the entire cube. It i.e. sums up the squared lengths of each side of the cube.	51
5.4	II_C measures the overall face areas of the cube. It i.e. sums up the squared areas of each side of the cube.	51
5.5	III_C measures squared volume of the cube.	52
6.1	a – quadratic form for a positive definite matrix. b – for a negative definite matrix. c – for a singular (and positive-indefinite) matrix. A line that runs through the bottom of the valley is the set of solutions. d – for an indefinite matrix. Because the solution is a saddle point, steepest descent will not work. In three dimensions or higher, a singular matrix can also have a saddle.	61
6.2	The method of Steepest Descent. a – Starting at $[-2, -2]^T$, take a step in the direction of steepest descent of f . b – Find the point on the intersection of these two surfaces that minimizes f . c – This parabola is the intersection of surfaces. The bottommost point is our target. d – The gradient at the bottommost point is orthogonal to the gradient of the previous step.	62
6.3	Black: The method of Steepest Descent starts at $[-2, -2]^T$ and converges at $[2, -2]^T$. Red: Method of Conjugate Gradient.	63
7.1	Let's eliminate the translation by pulling the rest triangle, back to the origin. We do the same thing to the deformed triangle.	71
7.2	Optimizing force computation	73
7.3	A high-res surface mesh embeded inside the volumetric FEM mesh.	76
8.1	Linear interpolation for the load steps	82
8.2	Naive parallelization of building \mathbf{f}_{int} and \mathbf{K}	90
8.3	a – Photons interacting with a surface. b – Model applied in rendering. [22]	94
8.4	Computation of the normal vector.	95

C. List of Figures

9.1	Results	98
A.1	Physical layout of the problem	100
A.2	A linear elastic spring with single degree of freedom	101
A.3	A linear elastic spring with two degrees of freedom	101
A.4	Equilibrium at each element	102
A.5	Pipe network	104
A.6	DC network	104
B.1	Sample two-dimensional linear system. The solution lies at the intersection of the lines.	105
B.2	a – Graph of a quadratic form $f(\mathbf{x})$. The minimum point of this surface is the solution to $\mathbf{Ax} = \mathbf{b}$. b – Contours of the quadratic form. Each ellipsoidal curve has constant $f(\mathbf{x})$	106
B.3	Gradient $f'(\mathbf{x})$ of the quadratic form. For every x_i , the gradient points in the steepest increase of $f(\mathbf{x})$, and is orthogonal to the contour lines.	107
B.4	a – quadratic form for a positive definite matrix. b – for a negative-definite matrix. c – for a singular (and positive-indefinite) matrix. A line that runs through the bottom of the valley is the set of solutions. d – For an indefinite matrix. Because the solution is a saddle point, Steepest Descent will not work. In three dimensions or higher, a singular matrix can also have a saddle.	107
B.5	The method of Steepest Descent. a – Starting at $[-2, -2]^T$, take a step in the direction of steepest descent of f . b – Find the point on the intersection of these two surfaces that minimizes f . c – This parabola is the intersection of surfaces. The bottommost point is our target. d – The gradient at the bottommost point is orthogonal to the gradient of the previous step.	109
B.6	The method of Steepest Descent starts at $[-2, -2]^T$ and converges at $[2, -2]^T$	110