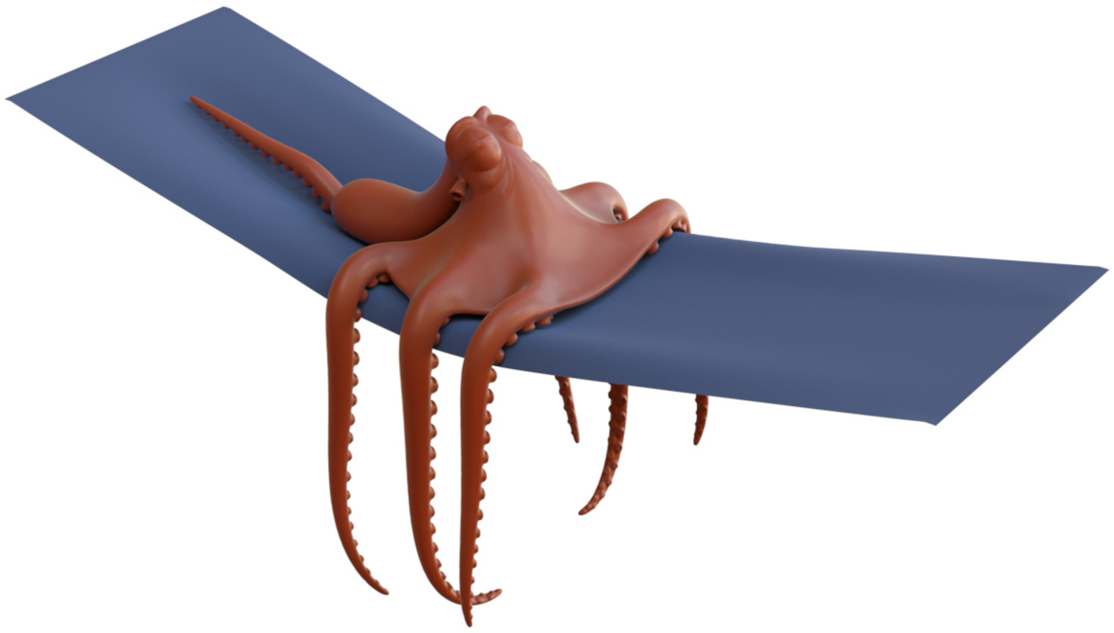
Course, SIGGRAPH 2022

Dynamic Deformables: Implementation and Production Practicalities *(Now With Code!)*

Instructors:
Theodore Kim, Yale University
David Eberle, Pixar Animation Studios



Built on: August 30, 2022

Abstract

Simulating dynamic deformation has been an integral component of Pixar's storytelling since Boo's shirt in *Monsters, Inc.* (2001). Recently, several key transformations have been applied to Pixar's core simulator *Fizt* that improve its speed, robustness, and generality. Starting with *Coco* (2017), improved collision detection and response were incorporated into the cloth solver, then with *Cars 3* (2017) 3D solids were introduced, and in *Onward* (2020) clothing is allowed to interact with a character's body with two-way coupling.

The 3D solids are based on a fast, compact, and powerful new formulation that we have published over the last few years at SIGGRAPH. Under this formulation, the construction and eigendecomposition of the force gradient, long considered the most onerous part of the implementation, becomes fast and simple. We provide a detailed, self-contained, and unified treatment here that is not available in the technical papers. We also provide, for the first time, open-source C++ implementations of many of the described algorithms.

This new formulation is only a starting point for creating a simulator that is up challenges of a production environment. One challenge is performance: we discuss our current best practices for accelerating system assembly and solver performance. Another challenge that requires considerable attention is robust collision detection and response. Much has been written about collision detection approaches such as proximity-queries, continuous collisions and global intersection analysis. We discuss our strategies for using these techniques, which provides us with valuable information that is needed to handle challenging scenarios.

Contents

Chapter 1

Introduction

1.1 The Goals of This Course

Fizt[1] has been the core simulator at Pixar ever since *Monsters, Inc.* in 2001. While many of the basics of the battle-tested Baraff and Witkin (1998) model that it is based on have remained the same over the years, a variety of key improvements and refinements have also been introduced. Many of these improvements have been spread across technical papers and talks presented the last two years at SIGGRAPH:

- *Stable Neo-Hookean Flesh Simulation*, Breannan Smith, Fernando de Goes, Theodore Kim (Technical Paper presented at SIGGRAPH 2018)
- *Better Collisions and Faster Cloth for Pixar's Coco*, David Eberle (Talk presented at SIGGRAPH 2018)
- *Clean Cloth Inputs: Removing Character Self-Intersections with Volume Simulation*, Audrey Wong, David Eberle, Theodore Kim (Talk presented at SIGGRAPH 2018)
- *Robust Skin Simulation in Incredibles 2*, Ryan Kautzman, Gordon Cameron, Theodore Kim (Talk presented at SIGGRAPH 2018)
- *Analytic Eigensystems for Isotropic Distortion Energies*, Breannan Smith, Fernando de Goes, Theodore Kim (Technical Paper presented at SIGGRAPH 2019)
- *Anisotropic Elasticity for Inversion-Safety and Element Rehabilitation*, Theodore Kim, Fernando de Goes, Hayley Iben (Technical Paper presented at SIGGRAPH 2019)

Together, these improvements constitute the next major version of the simulator, which is referred to internally as *Fizt2*. One of the goals of this course is to give a unified overview of these developments in a way that no single paper or talk has been able to before.

[1]Pronounced "fizz" as in soda and "tea" as in Earl Grey, hot.

In addition to these, we have also include a new introduction to the fundamentals of deformation mechanics (Chapter 2), based on a university course taught by one of the authors over the last ten years. This treatment specifically focuses on introducing the specific tensor-based formulation that was used to obtain the results in the aforementioned technical papers. Matlab and C++ implementations of several subtle pieces of code have been inlined as well, such as the $\frac{\partial \mathbf{F}}{\partial \mathbf{x}}$ matrix for tetrahedra (Fig. E.1) and shape derivative matrices (Fig. D.3) for hexahedra.

1.2 Now With Code!

In the spirit of the `pbrt` renderer of Pharr et al. (2016) we provide `HOBAK`[2], a prototype C++ implementation of many of algorithms described in these notes. Just like `pbrt` is *not RenderMan*, `HOBAK` is *not Fizt*. It was written entirely independently, and is intended to serve as a research test bed, reference implementation, and pedagogical tool. Whenever we have to choose between code clarity and run-time performance, we choose clarity.

`HOBAK` is not a production system. As such, it has very few dependencies, and should build and run right off the bat. If you want to jump right into the C++, a starter guide is in Appendix J

1.3 Author Biographies

Theodore Kim is an Associate Professor of Computer Science are Yale University. Previously, he was a Senior Research Scientist at Pixar Animation Studios, where he designed and implemented many of the solid mechanics algorithms described here. He holds a B.S. in Computer Science from Cornell University as well as an M.S. and Ph.D. from the University of North Carolina, Chapel Hill. He received a Scientific and Technical Academy Award in 2012 for his work on fluid simulation.

David Eberle is the Simulation Tools R&D Lead at Pixar Animation Studios. He has worked on numerous simulators used for animation and visual effects at Disney, Havok, PDI/Dreamworks and Pixar. He holds a B.S. in Mathematics from the University of South Carolina and an M.S. in Mathematics from Texas A&M University. His film credits include *Reign of Fire*, *Shrek 2*, *Shrek 3*, *Kung Fu Panda*, *Madagascar*, *Madagascar 2*, *Megamind*, *Inside Out*, *Finding Dory*, *Cars 3*, *Coco*, *Onward*, *Soul*, *Luca*, *Turning Red* and *Lightyear*.

1.4 Acknowledgements

These notes are an **ongoing draft**, so the authors would like to thank the following people for finding errors and omissions in the text: J.C. Chang, Towaki Takikawa, Peng Yu, Kai Jia, Kirill Klimov, Samaksh Ajay (Avi) Goyal, and especially Shinjiro Sueda. If you find other errors (there are sure to be many), feel free to reach out to the authors.

[2]*Hobak* (호박) is Korean for *pumpkin* or *squash*. We will use it to squash things.

Chapter 2

Deformation Fundamentals

2.1 Do I Need to Read This Chapter?

If you are brand-new to deformation simulation, or are coming back to it after some time away, the answer is: yes.

There are several existing introductions to deformation geared towards computer graphics, including the classic Witkin and Baraff (1997) as well as the more modern Sifakis and Barbic (2012) or Bargteil and Shinar (2018). If you are a practitioner that is already familiar with those materials, you can probably skip most of this chapter. The one caveat is that many of the fast, compact structures we show later on will depend on the notation for higher order tensors from Golub and Van Loan (2013). This does not show up that often in computer graphics, so you may still want to read Chapter 3.

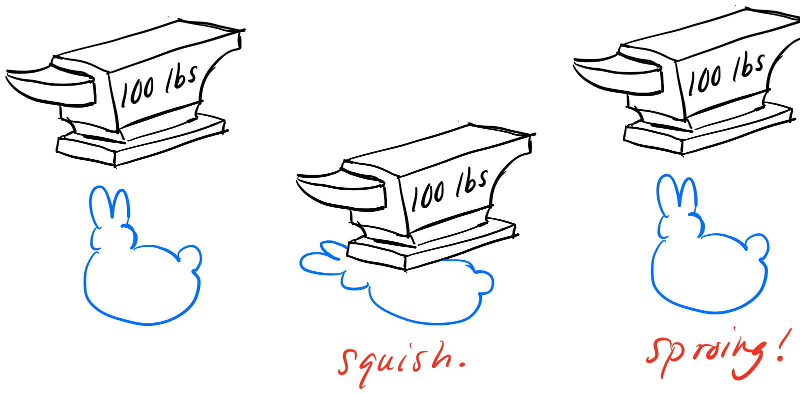


Figure 2.1.: The behavior of a hyperelastic bunny. Even when it is crushed by a 100 lb. weight, it recovers its original shape afterwards.

2.2 What Kind of Squashing Are We Talking About?

We will be looking at a specific form of solid deformation known as *hyperelastic* deformation. Say we take a bunny and squash it with a 100 lb. weight (Fig.2.1). When we remove the weight, it will sproing back to its *exact original shape*. This is the essence of hyperelasticity: when all external forces are removed, a hyperelastic object returns to its original shape. It does not matter if the bunny was stretched to a million times its original length, or crushed down into a pancake. Once the forces are removed, it bounces back to its original bunny shape.[1]

The original shape is thus considered quite special, as it is what the object "remembers" and always "tries" to return to, and is assigned a special name: the **rest shape**. In fact, when we squash a bunny with a 100 lb. weight, it is taking on the *best possible* or *closest possible* shape to the rest shape that it can muster under the current conditions.

But what do we mean by *best* or *closest*? Best compared to what, or closest with respect to what? This is really two questions.

1. How squashed am I? Alternately, how far am I from my original shape? Or: how badly deformed am I compared to my original shape? Most concretely: can I compute a quantitative *deformation score* that tells me exactly how stretched or squashed I am, relative to my original shape?
2. How much should I push back? Once I know how exactly how deformed I am, what should I do with this knowledge? Is being *really* deformed considered *extra*-bad, so I should push *extra*-hard to return to my original shape?

Let's look at these questions in turn.

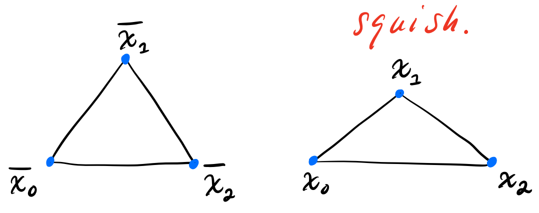


Figure 2.2.: On the left is the original, rest-shape triangle. On the right we (slightly) deform the triangle by moving $\mathbf{x}_1$ downwards.

2.3 How Squashed Am I?

First, let's look at an *obvious* way to measure deformation that will turn out to be *wrong*. From there, we can think about what went wrong and come up with something better.

[1]Clearly, this does not always happen in reality. If you stretch a rubber bunny too much, the fibers inside will slightly tear. When you let go, it will "remember" its rowdy treatment, and instead return to a slightly-stretched-out-looking-bunny shape. This is called *plastic* deformation, and can be layered on top of hyperelastic deformation, so we put it aside for the moment.

2.3.1 Measuring The Wrong Way

Let's look at a triangle (Fig. 2.2). We will denote its vertices in the **rest shape** using an overbar as $\bar{\mathbf{x}}_0$, $\bar{\mathbf{x}}_1$, $\bar{\mathbf{x}}_2$. After the triangle has been squashed away from it original shape, we denote its **deformed shape**[2] as $\mathbf{x}_0$, $\mathbf{x}_1$, $\mathbf{x}_2$. One obvious way to *compute a score* for how deformed we are is then:

$$\Psi_{\text{wrong}} = \sum_{i=0}^{2} \|\bar{\mathbf{x}}_i - \mathbf{x}_i\|_2. \tag{2.1}$$

Here, we use $\|\cdot\|_2$ to denote the 2-norm. Don't fret if you haven't seen this notation before; we have a description of it in Appendix C.1.1. It is a generalized shorthand for denoting "distance". We will also interchangeably refer to scoring functions as *energy functions*. For our purposes, scoring and energy functions have the exact same meaning.

The scoring function Ψ_{wrong}, certainly looks reasonable. As you squash $\mathbf{x}_1$ downwards more, or stretch it upwards more, the score (energy) definitely increases. We definitely want the score function to equal zero when no deformation is occurring, and some big number when lots of deformation is occurring. If that's all that is required, Ψ_{wrong} certainly seems to get the job done.

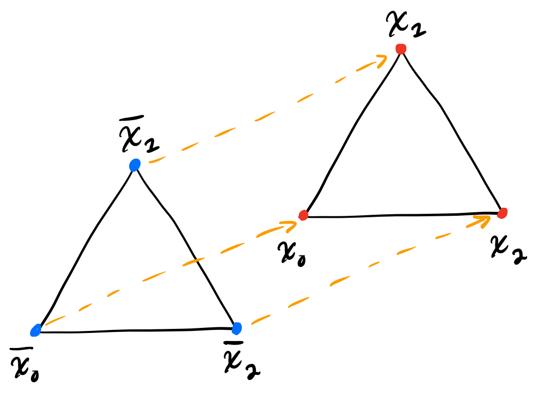


Figure 2.3.: We translate the original triangle (blue vertices) to the upper left (red vertices). Does this triangle count as "deformed"?

Or does it? The problem is that a bunch of things that we *would not* intuitively consider deformation also gets lumped in with the score. In Fig. 2.3, we have translated the triangle a small distance to the upper right. If we use Ψ_{wrong} on this new triangle, then it will show up as a non-zero score.

However, by any reasonable definition, this triangle is not *deformed*. Instead, it has been *moved*, or *translated*. Fig. 2.4 shows another problem case. If we *rotate* the triangle, then again, Ψ_{wrong} will return a non-zero score, mistakenly regarding this triangle as

[2]Other texts may use the terms *material* and *world* coordinates, so if you ever come across these, just remember that **rest = material** and **deformed = world**.

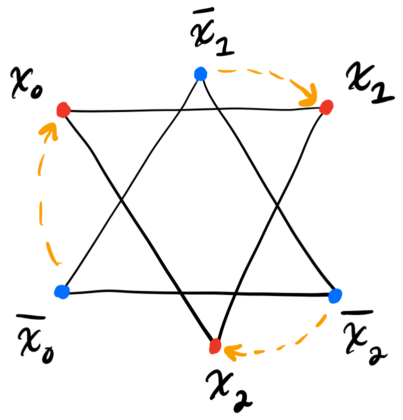


Figure 2.4.: We rotate the original triangle (blue vertices) clockwise (red vertices). Does this triangle count as "deformed"?

"deformed". However, by any reasonable definition, the triangle has been *spun* or *rotated*. It has not been deformed.

What we want is a scoring function Ψ that is *translation and rotation invariant*. If a triangle has been merely translated or rotated, its deformation score should show up as zero. Let's see if we can compute a score that will accomplish this.

2.3.2 Removing the Translation (Easy)

One way to encode the transformations that a triangle can undergo to use an affine map:

$$\begin{aligned} \phi(\bar{\mathbf{x}}) &= \mathbf{F}\bar{\mathbf{x}} + \mathbf{t} \\ &= \mathbf{x}. \end{aligned} \tag{2.2}$$

Here, $\mathbf{t} \in \Re^2$ and $\mathbf{F} \in \Re^{2\times 2}$ in 2D, and $\mathbf{t} \in \Re^3$ and $\mathbf{F} \in \Re^{3\times 3}$ in 3D (see the symbol table in Appendix A). There are only a few things that an affine transformation can encode:

- Rotation
- Scaling
- Reflection
- Translation.[3]

We saw that *rotation* and *translation* kept showing up and polluting our deformation score, even though we don't intuitively consider those as "deformations". However, *scaling* matches our notion of deformation quite nicely. Can we just isolate and measure that?

[3]You may ask: what about *shearing*? Wasn't that in linear algebra class too? That can be written in terms of rotations and scalings, so we can omit it here.

As a first step, let's see if we can carve away the translation, which is relatively easy. The translation lives entirely in $\mathbf{t}$, so if we perform some manipulation that cuts away that component, we are done. The canonical way of doing this is to take the derivative with respect to the rest shape:

$$\begin{aligned} \frac{\partial \phi(\bar{\mathbf{x}})}{\partial \bar{\mathbf{x}}} &= \frac{\partial}{\partial \bar{\mathbf{x}}} \left(\mathbf{F}\bar{\mathbf{x}} + \mathbf{t} \right) \\ &= \mathbf{F}. \end{aligned} \tag{2.3}$$

Since we are taking the derivative (gradient) of the affine map (deformation), the matrix $\mathbf{F}$ that pops out is called the *deformation gradient*. This matrix now contains rotation, scaling, and reflection, but *not translation*. If we base some sort of deformation score on $\mathbf{F}$, there is no way that translation can now pollute the result.

2.3.3 Deformation Gradient: Really Important

It is hard to overstate how important the deformation gradient is. Almost everything we do for the rest of this chapter will deal with $\mathbf{F}$. The variable has an unwieldy name, and everybody (Bonet and Wood (2008); Bower (2009); Marsden and Hughes (1994); Belytschko et al. (2013)) uses $\mathbf{F}$ to denote it, even though neither "deformation" nor "gradient" starts with an 'f'. Ideally the variable would have a snappy name and an intuitive symbol, but that is not the world we live in.[4] We are stuck with $\mathbf{F}$, so just remember that $\mathbf{F}$ is important.

You may be asking yourself: if $\mathbf{F}$ is so important, are you going to tell me how to compute it? Absolutely. It is a little involved, and takes a bit of space, so it has its own section in Appendix D. Taking for granted that we know *how* to compute this quantity, let's examine *what it means*.

2.3.4 Removing the Rotation (Not So Easy)

We now have $\mathbf{F}$, where translation has been mercifully subtracted off, but scaling, rotation and reflection remain entangled. How can we remove the rotation? This will turn out to be more difficult than removing the translation, and there will actually be more than one answer.

Let's try to build a simple score function out of $\mathbf{F}$. The simplest way to boil a matrix down to a scalar score is to take its *squared Frobenius norm*, denoted $\|\cdot\|_F$. As with the 2-norm in §2.3.1, don't worry if you're not intimately familiar with this norm. We document it in detail in Appendix C.1.2, but since this is the first time it is appearing, we will write it

[4]In geometry processing, it is sometimes denoted with J, presumably because it is the deformation *Jacobian*. Better, but still not great.

out explicitly,

$$\begin{aligned}\Psi_{\text{Dirichlet}} &= \|\mathbf{F}\|_F^2 \\ &= \sum_{i=0}^{2}\sum_{j=0}^{2} f_{ij}^2,\end{aligned} \tag{2.4}$$

where we have numbered the entries of $\mathbf{F}$ thusly:

$$\mathbf{F} = \begin{bmatrix} f_{00} & f_{01} & f_{02} \\ f_{10} & f_{11} & f_{12} \\ f_{20} & f_{21} & f_{22} \end{bmatrix}. \tag{2.5}$$

This score function is also known as the *Dirichlet energy*, and it *does not get the job done*. Remember, we want the deformation score to return zero when there has been zero deformation. However, the Dirichlet energy will return zero when all the entries of $\mathbf{F}$ are zero, i.e. $\mathbf{F} = \mathbf{0}$.

If we plug $\mathbf{F} = \mathbf{0}$ into our original deformation function $\phi(\bar{\mathbf{x}}) = \mathbf{F}\bar{\mathbf{x}} + \mathbf{t}$, it corresponds to the case where all the points $\bar{\mathbf{x}}$ are crushed into a zero-volume black hole centered at $\mathbf{t}$. As a scoring function, the Dirichlet energy might be a great black hole detector, but it's a crummy deformation measure.

2.3.4.1 A First (Wrong) Attempt

The Dirichlet energy returns a zero when $\mathbf{F} = \mathbf{0}$. What does an energy (score) that returns zero under zero deformation even look like? Well, if $\mathbf{F} = \mathbf{I}$, then certainly no deformation is occurring. In that case, $\phi(\bar{\mathbf{x}}) = \mathbf{F}\bar{\mathbf{x}} + \mathbf{t} = \mathbf{I}\bar{\mathbf{x}} + \mathbf{t} = \bar{\mathbf{x}} + \mathbf{t}$, which is purely a translation.

Can we design a function that returns zero when $\mathbf{F} = \mathbf{I}$? Well, if $\mathbf{F} = \mathbf{I}$, then $\|\mathbf{F}\|_F^2 = 3$. So, let's try the scoring function:

$$\Psi_{\text{Neo-Hookean}} = \|\mathbf{F}\|_F^2 - 3. \tag{2.6}$$

This is not a good idea. While it is true that when $\mathbf{F} = \mathbf{I}$, the score will equal zero, we also want zero to be the *smallest score possible*. Otherwise, the score becomes difficult to interpret. A zero score means zero deformation, a score greater than zero means some deformation, but what does a score less than zero mean? Less than zero deformation is happening?

In the case of $\Psi_{\text{Neo-Hookean}}$, if $\mathbf{F} = \mathbf{0}$, then $\Psi_{\text{Neo-Hookean}} = -3$. So, it's a pretty bad deformation measure. Things get worse if

$$\mathbf{F} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & \sqrt{\frac{3}{2}} & 0 \\ 0 & 0 & \sqrt{\frac{3}{2}} \end{bmatrix}. \tag{2.7}$$

This refers to a Wile-E-Coyote-style *pancake state*, which is definitely *not* a zero deformation state. And yet, the $\Psi_{\text{Neo-Hookean}}$ score returns a zero. Apparently there are many sneaky configurations that can fool this scoring function into thinking that no deformation has occurred. We could try to apply a squaring Band-Aid, e.g. $\Psi_{\text{Neo-Hookean,Band-Aid}} = \left(\|\mathbf{F}\|_F^2 - 3\right)^2$, but this would only address the negative scores problem. The scoring function would still return zero for the same sneaky deformed states.[5]

However, for some reason it has the fancy name "Neo-Hookean", so it must be important. It has been studied extensively in mechanical engineering and material science for the last 80 years (Mooney (1940); Rivlin (1948)), because we will see later that if you add a few important components, it becomes a *very interesting* deformation measure. But, it's not very good in its current form, so let's put it aside for now.

2.3.4.2 A Second (Still Wrong) Attempt

Next, let's try something capricious and stupid-looking (C&SL), which I promise will eventually lead somewhere. Let's just *subtract* **F** *and* **I** and take its norm. Then, we are at least guaranteed that when $\mathbf{F} = \mathbf{I}$, our scoring function will equal zero:

$$\Psi_{\text{C\&SL}} = \|\mathbf{F} - \mathbf{I}\|_F^2. \tag{2.8}$$

Additionally, since the **I** is inside the norm, everything gets squared, so there is no possibility of our function producing meaningless negative scores. We can use identity B.16 from Appendix B to expand this out to:

$$\begin{aligned}\Psi_{\text{C\&SL}} &= \|\mathbf{F} - \mathbf{I}\|_F^2 \\ &= \|\mathbf{F}\|_F^2 + \|\mathbf{I}\|_F^2 - 2\operatorname{tr}\mathbf{F} \\ &= \|\mathbf{F}\|_F^2 + 3 - 2\operatorname{tr}\mathbf{F}.\end{aligned}$$

I have assumed that we're looking at the 3D case, which is why $\|\mathbf{I}\|_F^2 = 3$. Looking at the expansion, the score still isn't great. The first term is the black-hole favoring Dirichlet energy, which doesn't do us any favors. Then, we have the constant of 3, which looks suspiciously similar to our not-great Neo-Hookean energy, followed by a new $\operatorname{tr}\mathbf{F}$ term. If you haven't seen a matrix trace in a while, there is a refresher in Appendix C.2. Since this is its first appearance here, we will write it out explicitly:

$$\operatorname{tr}\mathbf{F} = \sum_{i=0}^{2} f_{ii} = f_{00} + f_{11} + f_{22}. \tag{2.9}$$

This scoring function *will* equal zero when $\mathbf{F} = \mathbf{I}$, but when $\mathbf{F} \neq \mathbf{I}$, weird things happen. For example, if **F** happens to be some rotation matrix,

$$\mathbf{F}(\theta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{bmatrix}, \tag{2.10}$$

[5]Fun fact: This squaring is equivalent to the (yet-to-be-seen) volume preservation term from the St. Venant Kirchhoff model. It is still not terribly useful all on its own though.

then $\Psi_{\text{C\&SL}}$ will return a non-zero score whenever θ is not zero or some multiple of 2π. But, these would still clearly correspond to deformation-free states! It's better than a black-hole detector, but not by much.

Overall, it's weird that $\operatorname{tr}\mathbf{F}$ is shows up at all in this formula. Since $\mathbf{F}$ can be an arbitrary, non-symmetric matrix, it is not clear that its trace is terribly meaningful outside of cancelling off $\|\mathbf{F}\|_F^2 + 3$ in the overly-specific case of $\mathbf{F} = \mathbf{I}$.

2.3.4.3 *A Third Attempt: St. Venant Kirchhoff*

How can we make a version of $\Psi_{\text{C\&SL}}$ that instead equals zero whenever $\mathbf{F}$ is a pure rotation? One way to do this is to exploit a nice linear algebra identity. If $\mathbf{F}$ is a pure rotation, then we know that it must be the case that $\mathbf{F}^T\mathbf{F} = \mathbf{I}$. This follows from the fact that $\mathbf{F}^{-1} = \mathbf{F}^T$ when $\mathbf{F}$ is orthonormal.

Well, why don't we compute $\mathbf{F}^T\mathbf{F}$, subtract off the rotation part, i.e. the $\mathbf{I}$ part, and see what's left over? These leftovers probably characterize the deformation in some reasonable way. We can write this down as:

$$\Psi_{\text{StVK, stretch}} = \frac{1}{2}\|\mathbf{F}^T\mathbf{F} - \mathbf{I}\|_F^2. \tag{2.11}$$

This is a pretty good idea! The idea is good enough that people have given it a special name: The St. Venant-Kirchhoff stretching energy, or "StVK, stretch", for short.[6] The individual components of $\mathbf{F}^T\mathbf{F} - \mathbf{I}$ are popular enough that they have special names and symbols:

$\mathbf{C} = \mathbf{F}^T\mathbf{F}$	The right Cauchy-Green tensor
$\mathbf{E} = \frac{1}{2}\left(\mathbf{F}^T\mathbf{F} - \mathbf{I}\right)$	Green's strain

Using these, the energy can be written in a highly compact form:

$$\Psi_{\text{StVK, stretch}} = \|\mathbf{E}\|_F^2. \tag{2.12}$$

Again, this is a decent way to measure deformation. We're using $\mathbf{F}$, so the translation has been subtracted off, and we using an $\mathbf{F}^T\mathbf{F}$ identity to factor off rotation. If we were going to complain about one thing though, it would be that the scoring function is *overly non-linear*. Performing the same expansion as we did for $\Psi_{\text{C\&SL}}$, we can get:

$$\begin{aligned} \Psi_{\text{StVK, stretch}} &= \|\mathbf{F}^T\mathbf{F} - \mathbf{I}\|_F^2 \\ &= \|\mathbf{F}^T\mathbf{F}\|_F^2 + \operatorname{tr}\mathbf{I} - 2\operatorname{tr}(\mathbf{F}^T\mathbf{F}). \end{aligned} \tag{2.13}$$

The leading term, $\|\mathbf{F}^T\mathbf{F}\|_F^2$, is 4th-order, or *quartic* in $\mathbf{F}$. The $\mathbf{F}$ gets squared by $\mathbf{F}^T\mathbf{F}$, and then squared again by the norm, $\|\cdot\|_F^2$, which makes it 4th order overall. We will see later that this 4th-order-ness adds additional computational complexity, but in a way that is not actually desirable.

[6] Actually, *I* call it the StVK stretching energy, because it's the first term (the stretching term) in the StVK energy. So if you look up "St. Venant-Kirchhoff", you'll see this term, plus another one. We'll take a look at that other one later.

2.3.4.4 A Fourth Attempt: As-Rigid-As-Possible

We took advantage of the $\mathbf{F}^T\mathbf{F} = \mathbf{I}$ identity, but are there other identities that we could have used? If we're specifically concerned with teasing apart the rotation and non-rotation component of a matrix, the *polar decomposition* is helpful:

$$\mathbf{F} = \mathbf{R}\mathbf{S}. \tag{2.14}$$

Here, the matrix $\mathbf{R}$ is the rotational part of $\mathbf{F}$, and $\mathbf{S}$ is the non-rotational (scaling) component of $\mathbf{F}$. This decomposition seems custom-built to solve exactly the problem we are interested in.

However, you need to be a little careful about using the polar decomposition, because its classic definition is slightly different from the version we want. Usually $\mathbf{R}$ is only defined as a *unitary* matrix, not a rotation matrix, so it only needs to satisfy $\mathbf{R}^T\mathbf{R} = \mathbf{I}$. 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 $\mathbf{S}$. We will call this specific flavor the *rotation variant* of the polar decomposition. Details on how to compute this are in Appendix F.

In short: if you end up calling the polar decomposition code in some library, make sure it is computing the *rotation variant*, not the traditional version. If it is not, you will need to write a small wrapper for the library call in the style of Figs. F.1 and F.2.

With that out of the way, let's assume you can compute the rotation-variant polar decomposition, and now have $\mathbf{R}$ and $\mathbf{S}$ in hand. What can you do with them? In StVK, we took $\mathbf{F}^T\mathbf{F}$ and subtracted off its rotational component, $\mathbf{I}$. Well, now we have a different representation for the rotation, $\mathbf{R}$, but can we do the same thing? Subtract the rotation component off of $\mathbf{F}$, and assume that the leftovers must characterize the deformation somehow?

The most basic, gee-I-wonder-if-this-will-work way of writing this down is:

$$\Psi_{\text{ARAP}} = \|\mathbf{F} - \mathbf{R}\|_F^2. \tag{2.15}$$

There are a few worrying-looking things about this formulation. Usually when you have two matrices, multiplying them together is a first thing to try, because you can see what the multiplication is *really* doing by peeking at the SVD or eigendecomposition. But, instead of multiplying, we're subtracting. Can we really just subtract two arbitrary-looking matrices from each other and get something that isn't total garbage? They are related via the polar decomposition, but does that really mean we can just start subtracting these entries from each other?

Surprisingly, the answer is yes. In fact, this energy function is so indispensable that it has been given a special name in geometry processing: the As-Rigid-As-Possible (ARAP) energy (Sorkine and Alexa (2007)). We'll do a little more analysis of this energy a little later to see how it is actually more clever than it first appears.

This energy does not have the "overly non-linear" problem of StVK. If we apply identity B.16, we get"

$$\begin{aligned}\Psi_{\text{ARAP}} &= \|\mathbf{F}-\mathbf{R}\|_F^2 \\ &= \|\mathbf{F}\|_F^2 + \|\mathbf{R}\|_F^2 - 2\operatorname{tr}(\mathbf{F}^T\mathbf{R}) \\ &= \|\mathbf{F}\|_F^2 + 3 - 2\operatorname{tr}(\mathbf{S}).\end{aligned} \tag{2.16}$$

The energy is at most *quadratic* in $\mathbf{F}$. We will see later that the introduction of $\mathbf{R}$ and $\mathbf{S}$ creates some computational challenges, but for the time being, it is undeniable that Ψ_{ARAP} is appealingly less non-linear than $\Psi_{\text{StVK,stretch}}$.

In physical simulation, the ARAP energy is actually the first term in the popular *co-rotational* energy. This energy has been known in mechanical engineering since at least the 1980s (Rankin and Brogan (1986)), and the ARAP energy corresponds to the specific mechanical case where Poisson's ratio is set to zero. This energy was rediscovered by several graphics researchers in the early 2000s (Müller et al. (2002); Etzmuss et al. (2003); Irving et al. (2004)), which created some confusion because they each gave it a different name, such as "stiffness warping" or "rotated linear". However, these models are all equivalent, and nowadays everybody seems to have settled on "co-rotational" as the name for this energy.

2.3.4.5 *Summary of Energy (Scoring) Functions*

We have now seen a bunch of different energy functions. Let's summarize their pluses and minuses:

Energy	**Pros**	**Cons**
Dirichlet	Not many. A good black hole detector?	Doesn't measure deformation effectively.
Neo-Hookean	Returns zero when $\mathbf{F}=\mathbf{I}$	Returns zero for other configurations too, and can also return negative scores.
C&SL	Returns zero when $\mathbf{F}=\mathbf{I}$, and does not return negative scores.	Does not return zero under pure rotations.
StVK	Returns zero when $\mathbf{F}$ is a pure rotation, and reasonable scores otherwise.	Overly non-linear (quartic).
ARAP	Returns zero when $\mathbf{F}$ is a rotation, reasonable scores otherwise, and is quadratic	That $\mathbf{R}$ term is going to cause trouble.

As I said at the top, removing the rotation from the score is not easy, and there is more than one strategy. Now that we have a bunch of options, what should we do with them?

2.4 Force Computation: How Much Should I Push Back?

Now that we can measure how deformed we are, we can return to the second question from §2.2: *How much should I push back?*

The deformation measure directly generates forces. Surprise! When we were muddling over different ways to measure deformation, we were actually trying out different elastic energy potentials. The two are completely equivalent for our purposes. The deformation measure directly dictates the force response. If you want to design a new material model, come up with a different way to measure deformation.

To get the forces $\mathbf{f}$ on the nodes of a triangle in 2D, we stack the vertices of our triangle ($\mathbf{x}_0$, $\mathbf{x}_1$ and $\mathbf{x}_2$) into a big vector,

$$\mathbf{x} = \begin{bmatrix} \mathbf{x}_0 \\ \mathbf{x}_1 \\ \mathbf{x}_2 \end{bmatrix} = \begin{bmatrix} x_0 \\ y_0 \\ x_1 \\ y_1 \\ x_2 \\ y_2 \end{bmatrix} \in \Re^6, \tag{2.17}$$

and then take the gradient:

$$\mathbf{f} = -a\frac{\partial \Psi}{\partial \mathbf{x}}. \tag{2.18}$$

Here, a is the area of the original triangle at rest. A really tiny deformed triangle the size of an ant is not going to exert the same force as a deformed triangle the size of the Empire State Building, so we should scale it.

That's all well and good, and Eqn. 2.18 looks straightforward, except that all the energies we looked at in §2.3.4.5 were written in terms of $\mathbf{F}$, not $\mathbf{x}$. In fact, we made a big deal out of the fact that $\mathbf{F}$ doesn't have all these junky measurement-polluting properties of $\mathbf{x}$, like translation, and that it was clearly the better primary variable. Having gone down that path, how are we supposed to take the gradient with respect to $\mathbf{x}$ now?

2.4.1 Computing $\frac{\partial \Psi}{\partial \mathbf{x}}$

We pushed the computation of $\mathbf{F}$ to Appendix D, but we're going to pull the final result of that discussion back up here. For a triangle, we pack its original $\bar{\mathbf{x}}_i$ and deformed $\mathbf{x}_i$ vertices into two matrices, $\mathbf{D}_m$ and $\mathbf{D}_s$, thusly:

$$\mathbf{D}_m = \left[\begin{array}{c|c} \bar{\mathbf{x}}_1 - \bar{\mathbf{x}}_0 & \bar{\mathbf{x}}_2 - \bar{\mathbf{x}}_0 \end{array}\right] \qquad \mathbf{D}_s = \left[\begin{array}{c|c} \mathbf{x}_1 - \mathbf{x}_0 & \mathbf{x}_2 - \mathbf{x}_0 \end{array}\right]. \tag{2.19}$$

The deformation gradient is then a composition of these two:

$$\mathbf{F} = \mathbf{D}_s\mathbf{D}_m^{-1}. \tag{2.20}$$

First the good news: we need the gradient with respect to $\mathbf{x}$, not $\bar{\mathbf{x}}$. If we instead needed $\frac{\partial \Psi}{\partial \bar{\mathbf{x}}}$ things would get ugly real fast, because $\bar{\mathbf{x}}_i$ appears in $\mathbf{D}_m$, which is inside an inverse, $\mathbf{D}_m^{-1}$, so we'd need to figure out the derivative of a matrix inverse.

Now the bad news: it's still pretty ugly. Let's take a really simple energy, $\Psi_{\text{Dirichlet}}$ as an example:

$$\Psi_{\text{Dirichlet}} = \|\mathbf{F}\|_F^2. \tag{2.21}$$

Here's what we're going to have to do:

- Plug $\mathbf{F} = \mathbf{D}_s \mathbf{D}_m^{-1}$ in to $\Psi_{\text{Dirichlet}}$.
- Multiply everything through to get a massive one-line equation for $\|\mathbf{F}\|_F^2$.
- Take the derivative of this massive equation six times, once for each entry in $\mathbf{x}$, and stack the results into a force vector.
- Hope you didn't make a mistake in your derivation somewhere. Port the expressions into C++, and hope you don't make a mistake transferring these equations from paper to code.

2.4.2 Don't Do It This Way

Let's gaze at this ugliness for just a minute longer. Just a minute though. After that I have to look away, and tell you there's a better way. To start, we can tag each entry in $\mathbf{D}_m^{-1}$ thusly,

$$\mathbf{D}_m^{-1} = \begin{bmatrix} m_0 & m_2 \\ m_1 & m_3 \end{bmatrix} \tag{2.22}$$

then we'll try to multiply out the matrix $\mathbf{F}^T\mathbf{F}$ in anticipation of cramming everything together for $\|\mathbf{F}\|_F^2$.

$$\begin{aligned} \mathbf{F}^T\mathbf{F} &= \begin{bmatrix} m_0 & m_1 \\ m_2 & m_3 \end{bmatrix} \left[\begin{array}{c|c} \mathbf{x}_1 - \mathbf{x}_0 & \mathbf{x}_2 - \mathbf{x}_0 \end{array} \right]^T \left[\begin{array}{c|c} \mathbf{x}_1 - \mathbf{x}_0 & \mathbf{x}_2 - \mathbf{x}_0 \end{array} \right] \begin{bmatrix} m_0 & m_2 \\ m_1 & m_3 \end{bmatrix} \\ &= \begin{bmatrix} m_0 & m_1 \\ m_2 & m_3 \end{bmatrix} \begin{bmatrix} (\mathbf{x}_1 - \mathbf{x}_0)^T(\mathbf{x}_1 - \mathbf{x}_0) & (\mathbf{x}_2 - \mathbf{x}_0)^T(\mathbf{x}_1 - \mathbf{x}_0) \\ (\mathbf{x}_2 - \mathbf{x}_0)^T(\mathbf{x}_1 - \mathbf{x}_0) & (\mathbf{x}_2 - \mathbf{x}_0)^T(\mathbf{x}_2 - \mathbf{x}_0) \end{bmatrix} \begin{bmatrix} m_0 & m_2 \\ m_1 & m_3 \end{bmatrix} \end{aligned} \tag{2.23}$$

Just to clean things up, let's introduce more temporary variables:

$$\begin{bmatrix} (\mathbf{x}_1 - \mathbf{x}_0)^T(\mathbf{x}_1 - \mathbf{x}_0) & (\mathbf{x}_2 - \mathbf{x}_0)^T(\mathbf{x}_1 - \mathbf{x}_0) \\ (\mathbf{x}_2 - \mathbf{x}_0)^T(\mathbf{x}_1 - \mathbf{x}_0) & (\mathbf{x}_2 - \mathbf{x}_0)^T(\mathbf{x}_2 - \mathbf{x}_0) \end{bmatrix} = \begin{bmatrix} d_0 & d_1 \\ d_1 & d_2 \end{bmatrix}. \qquad \text{(Note it's symmetric)}$$

Now let's forge ahead further,

$$\mathbf{F}^T\mathbf{F} = \begin{bmatrix} m_0 & m_1 \\ m_2 & m_3 \end{bmatrix} \begin{bmatrix} d_0 & d_1 \\ d_1 & d_2 \end{bmatrix} \begin{bmatrix} m_0 & m_2 \\ m_1 & m_3 \end{bmatrix}$$
$$= \begin{bmatrix} m_0^2 d_0 + 2m_0 m_1 d_1 + m_1^2 d_2 & m_0 m_2 d_0 + (m_0 m_3 + m_1 m_2) d_1 + m_1 m_3 d_2 \\ m_0 m_2 d_0 + (m_0 m_3 + m_1 m_2) d_1 + m_1 m_3 d_2 & m_2^2 d_0 + 2m_2 m_3 d_1 + m_3^2 d_2 \end{bmatrix},$$

and then finally smash everything together with the Frobenius norm:

$$\|\mathbf{F}\|_F^2 = \left(m_0^2 d_0 + 2m_0 m_1 d_1 + m_1^2 d_2\right)^2 + \left(m_0 m_2 d_0 + (m_0 m_3 + m_1 m_2) d_1 + m_1 m_3 d_2\right)^2$$
$$\left(m_0 m_2 d_0 + (m_0 m_3 + m_1 m_2) d_1 + m_1 m_3 d_2\right)^2 + \left(m_2^2 d_0 + 2m_2 m_3 d_1 + m_3^2 d_2\right)^2.$$

Okay, I have to stop now. Cripes, it hurts to even look at this. The expression is big enough that it barely fits on two lines. I probably made a mistake in there somewhere; good luck finding it. Next we'll have to substitute back in $d_0 = (\mathbf{x}_1 - \mathbf{x}_0)^T(\mathbf{x}_1 - \mathbf{x}_0)$ and from there, $\mathbf{x}_0 = [x_0 \quad y_0]^T$, as well as $d_1, d_2, \mathbf{x}_1$ and $\mathbf{x}_2$, and then take multiple derivatives of the resulting hideousness. Also, things get worse in 3D, and the Dirichlet energy isn't even that complicated an energy! Things sink to even deeper depths of hideousness for more complex material models.

You could use a computer algebra system like Mathematica or Matlab's Symbolic Toolkit to handle all this. That would just offload the ugliness generation to the computer, and the final result would still be ugly. Surely there is a better way.

2.4.3 What Now?

There *is* a better way, which we will describe in the next chapter. When we do, you will see that the *forces* are now cleaner to compute, but the *force gradients* (a.k.a the *energy Hessians*) are still ugly.

You may be tempted to take the coward's way out here. All of this ugliness started because I insisted that we use $\mathbf{F}$ to compute our deformation measure instead of $\mathbf{x}$. If we had just stuck to $\mathbf{x}$, we wouldn't be in this mess, because computing $\frac{\partial \Psi}{\partial \mathbf{x}}$ when Ψ is written purely in terms of $\mathbf{x}$ is *much easier.* Just look at this gorgeous-looking spring-mass energy:

$$\Psi = \frac{\mu}{2}\|\mathbf{x} - \mathbf{x}_0\|_2^2 \tag{2.24}$$
$$\frac{\partial \Psi}{\partial \mathbf{x}} = \mu(\mathbf{x} - \mathbf{x}_0). \tag{2.25}$$

SO EASY. Don't be seduced though. If allow ourselves be sucked down into this Charybdis, we'll be limiting ourselves to the world of spring-mass models, and the richer universes of squishing and squashing volumetric responses will be inaccessible. Let's not do that; let's see how far we can take this $\mathbf{F}$ thing.

Chapter 3

Computing Forces the Tensor Way

If you're just joining us after skipping Chapter 1, the following is our current predicament. We are trying to compute forces, $\mathbf{f} = a\frac{\partial \Psi}{\partial \mathbf{x}}$, but it turns out that directly taking the $\mathbf{x}$ derivative when your energy Ψ is based on the deformation gradient $\mathbf{F}$ is extremely ugly. It's ugly enough that we are starting to question whether using an $\mathbf{F}$-based energy is even a good idea at all. Now you're all caught up, so on with the show.

We will clean things up using an application of the chain rule that separates $\frac{\partial \Psi}{\partial \mathbf{x}}$ into two components:

$$\frac{\partial \Psi}{\partial \mathbf{x}} = \frac{\partial \Psi}{\partial \mathbf{F}}\frac{\partial \mathbf{F}}{\partial \mathbf{x}}. \tag{3.1}$$

Each piece, $\frac{\partial \Psi}{\partial \mathbf{F}}$ and $\frac{\partial \mathbf{F}}{\partial \mathbf{x}}$, is relatively clean in isolation. The ugliness we saw in the previous chapter appears if you try to bite off more than you can chew and handle both pieces simultaneously. However, if you haven't worked with higher-order tensors before[1], you'll run into trouble the moment you start trying to work out these terms. The whole thing will look like a dirty calculus trick that some professor played on you where he[2] left all the difficult details out. Let's look at the details now.

3.1 Thinking About 3rd-Order Tensors

The first term in Eqn. 3.1, the $\frac{\partial \Psi}{\partial \mathbf{F}}$ term, is relatively easy. Since $\Psi \in \Re$ and $\mathbf{F} \in \Re^{3\times 3}$ the derivative is just a matrix:

$$\frac{\partial \Psi}{\partial \mathbf{F}} = \begin{bmatrix} \frac{\partial \Psi}{\partial f_{00}} & \frac{\partial \Psi}{\partial f_{01}} & \frac{\partial \Psi}{\partial f_{02}} \\ \frac{\partial \Psi}{\partial f_{10}} & \frac{\partial \Psi}{\partial f_{11}} & \frac{\partial \Psi}{\partial f_{12}} \\ \frac{\partial \Psi}{\partial f_{20}} & \frac{\partial \Psi}{\partial f_{21}} & \frac{\partial \Psi}{\partial f_{22}} \end{bmatrix} \in \Re^{3\times 3}. \tag{3.2}$$

[1]Running a few TensorFlow Python scripts *definitely doesn't count.* If you *have* actually worked with tensors before, feel free to skip ahead to §3.4.

[2]That would be me.

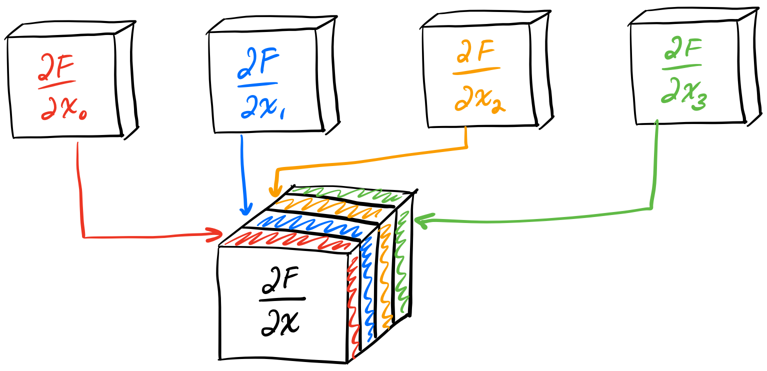
The f_{ij} scalars are the entries of $\mathbf{F}$, as described in Appendix A. The result is clearly a 3×3 matrix, also known as a *2nd-order tensor*.

The confusion begins when we try to compute $\frac{\partial \mathbf{F}}{\partial \mathbf{x}}$. Since $\mathbf{F} \in \Re^{3\times3}$ is a matrix and $\mathbf{x} \in \Re^{12}$ is a vector, what form should the derivative take? Let's look at x_0, the first entry of $\mathbf{x}$. We can take the derivative of $\mathbf{F}$ with respect to that:

$$\frac{\partial \mathbf{F}}{\partial x_0} = \begin{bmatrix} \frac{\partial f_{00}}{\partial x_0} & \frac{\partial f_{01}}{\partial x_0} & \frac{\partial f_{02}}{\partial x_0} \\ \frac{\partial f_{10}}{\partial x_0} & \frac{\partial f_{11}}{\partial x_0} & \frac{\partial f_{12}}{\partial x_0} \\ \frac{\partial f_{20}}{\partial x_0} & \frac{\partial f_{21}}{\partial x_0} & \frac{\partial f_{22}}{\partial x_0} \end{bmatrix} \in \Re^{3\times3}. \tag{3.3}$$

Already, the result is a matrix. What are we supposed to do with the next derivative, $\frac{\partial \mathbf{F}}{\partial x_1}$, and the one after that, and the one after that? Eventually I'll have a pile of 12 matrices, but what am I supposed to do with them? Is there some standard way of arranging these matrices together?

The answer is no.[3] The problem is that we now have a *3rd-order tensor*, which usually doesn't show up in an introductory linear algebra texts. If you look in texts on higher-order tensor manipulation (e.g. Golub and Van Loan (2013); Simmonds (2012); Kolda and Bader (2009)), they describe a variety of ways of arranging this pile of matrices, some of which have important-sounding names like *Einstein notation*. Surely that's the one? If it's good enough for Albert Einstein, it's good enough for us? In fact, it's not; we should use the right tool for the right situation, and we're not looking at relativity right now.



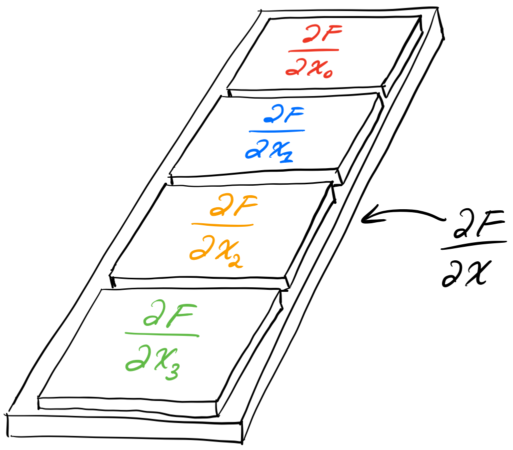
$\frac{\partial F}{\partial x_0}$
$\frac{\partial F}{\partial x_1}$
$\frac{\partial F}{\partial x_2}$
$\frac{\partial F}{\partial x_3}$
$\frac{\partial F}{\partial x}$

Figure 3.1.: We *could* take each matrix along the top and stack them all into a cube. For the full $\frac{\partial \mathbf{F}}{\partial \mathbf{x}}$ there are 12 slices, but hopefully you get the picture with just 4. We're not going to do it this way though.

The dimension of the derivative is $\frac{\partial \mathbf{F}}{\partial \mathbf{x}} \in \Re^{3\times3\times12}$, where the third dimension is the big giveaway that we're looking at a 3rd-order tensor. Many people think of this is as a *cube*

[3]Some of your colleagues may dispute my claim and say that there is *definitely* a standard notation for tensors that *everybody who's anybody* uses. I would recommend being suspicious of any colleague who tries to make such an intimidation-based argument.

of matrices. We can take our pile of 12 matrices, and stack them like a deck of cards into a cube (Fig. 3.1). Just to sow further confusion, if you look at my paper, Kim and James (2012), you'll see that I decided to use this mental representation in that paper too.[4]



$\frac{\partial F}{\partial x_0}$
$\frac{\partial F}{\partial x_1}$
$\frac{\partial F}{\partial x_2}$
$\frac{\partial F}{\partial x_3}$
$\frac{\partial F}{\partial x}$

Figure 3.2.: We prefer to think of 3rd-order tensors as a vector of matrices. It's like a long cutting board on a kitchen counter, with the matrices stacked along it, like individually-wrapped slices of (festively-colored) American cheese. There should be 12 matrices in total, but I only show 4 because if I show all 12 they'd be so small you couldn't read the writing on them.

We're not going to take that view here. Instead, we're going to think of 3rd-order tensors as a *vector of matrices*. Think of the $\frac{\partial \mathbf{F}}{\partial \mathbf{x}}$ tensor like a big wooden cutting board on your kitchen counter. Each matrix $\frac{\partial \mathbf{F}}{\partial x_i}$ is then placed in order, along a line, down the board (Fig. 3.2), like slices of individually-wrapped American cheese. When we write out a 3rd-order tensor explicitly, we'll use the square brackets to imitate our mental picture of a cutting-board-with-cheese. For example:

$$\begin{bmatrix} \begin{bmatrix} a & c \\ b & d \end{bmatrix} \\ \\ \begin{bmatrix} e & g \\ f & h \end{bmatrix} \\ \\ \begin{bmatrix} i & k \\ j & l \end{bmatrix} \end{bmatrix} = \begin{bmatrix} [\mathbf{A}] \\ \\ [\mathbf{B}] \\ \\ [\mathbf{C}] \end{bmatrix}. \tag{3.4}$$

On the right hand side, we wrapped each matrix in a $[\cdot]$ as well. This is to distinguish the 3rd-order tensor from the more familiar 2nd-order block matrix.

[4]Cubes are a good way to think about things when you're doing *model reduction*, but that's not what we're doing right now.

3.2 Multiplication With 3rd-Order Tensors

Once we have a mental picture of the 3rd-order tensor in place, how should we think of stuff like multiplication? This is pretty important, since once of our initial claims was that the chaining $\frac{\partial \Psi}{\partial \mathbf{x}} = \frac{\partial \Psi}{\partial \mathbf{F}} \frac{\partial \mathbf{F}}{\partial \mathbf{x}}$ would make things easier, and so far it sure doesn't look easier. What's the product of $\frac{\partial \Psi}{\partial \mathbf{F}}$, a matrix, and $\frac{\partial \mathbf{F}}{\partial \mathbf{x}}$, a cutting board?

This is the part where you get annoyed with the professor for gliding past an important detail. If we're writing down rock-solid, 100%-corresponds to the actual computation you'll be doing we should really write:

$$\frac{\partial \Psi}{\partial \mathbf{x}} = \frac{\partial \mathbf{F}}{\partial \mathbf{x}} : \frac{\partial \Psi}{\partial \mathbf{F}}, \tag{3.5}$$

where the ':' is a double-contraction. If you have not seen double-contractions before, we review what it means to contract two *matrices* together in Appendix C.3. The key takeaway from that section is:

$$\mathbf{A} : \mathbf{B} = \begin{bmatrix} a_0 & a_2 \\ a_1 & a_3 \end{bmatrix} \begin{bmatrix} b_0 & b_2 \\ b_1 & b_3 \end{bmatrix} = a_0 b_0 + a_1 b_1 + a_2 b_2 + a_3 b_3. \tag{3.6}$$

However, $\frac{\partial \mathbf{F}}{\partial \mathbf{x}}$ is a 3rd-order tensor, not a matrix. What does a 3rdvs. 2nddouble-contraction mean? We define it as:

$$\mathbb{A} : \mathbf{B} = \begin{bmatrix} \begin{bmatrix} a_0 & a_2 \\ a_1 & a_3 \end{bmatrix} \\ \\ \begin{bmatrix} a_4 & a_6 \\ a_5 & a_7 \end{bmatrix} \\ \\ \begin{bmatrix} a_8 & a_{10} \\ a_9 & a_{11} \end{bmatrix} \end{bmatrix} \begin{bmatrix} b_0 & b_2 \\ b_1 & b_3 \end{bmatrix} = \begin{bmatrix} a_0 b_0 + a_1 b_1 + a_2 b_2 + a_3 b_3 \\ a_4 b_0 + a_5 b_1 + a_6 b_2 + a_7 b_3 \\ a_8 b_0 + a_9 b_1 + a_{10} b_2 + a_{11} b_3 \end{bmatrix}. \tag{3.7}$$

We have now used the blackboard font $\mathbb{A}$ to denote a higher-order tensor for the first time. We will use these to denote tensors that are 3rd-order and above; the order should be (I hope) evident from the context.

To form another mental picture, $\frac{\partial \Psi}{\partial \mathbf{F}}$ is a cheese slice off to the side that has the exact same size as those on our cutting board, $\frac{\partial \mathbf{F}}{\partial \mathbf{x}}$. We then perform a double-contraction of this new slice against all of those on the cutting board (Fig. 3.3). The result is then a good ole' vector.

3.3 Multiplication With Flattened Tensors

Just to drive you crazy, there's a second, more conventional way to write all this down. Any tensor, be it 3rd-order, 4th-order, or 100th order, can always be *flattened* or *vectorized*

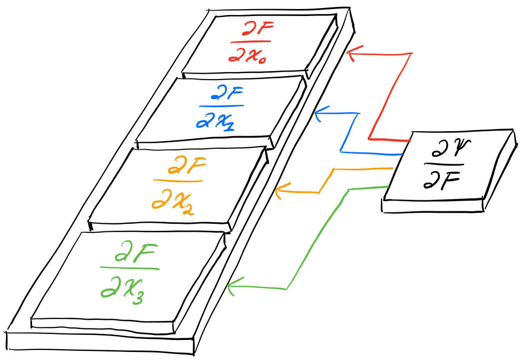

$\frac{\partial F}{\partial x_0}$
$\frac{\partial F}{\partial x_1}$
$\frac{\partial F}{\partial x_2}$
$\frac{\partial F}{\partial x_3}$
$\frac{\partial \Psi}{\partial F}$

Figure 3.3.: Double-contraction of a 3rd-order tensor against a 2nd-order tensor is like taking a cheese slice (right) and double-contracting it against each multi-colored slice on the cutting board (left). The result here would be an $\Re^4$ vector.

back into the familiar 2nd-order matrix form. We're more-or-less going to follow the convention of Golub and Van Loan (2013) here.

Unfortunately, it will turn out that *both the tensor and matrix representations* are useful. Just like you had to learn both radians and degrees for trigonometric functions, you're going to have to learn both the tensor and flattened versions here. It will be worth it, I promise.

We introduce the vectorization operator $\mathrm{vec}(\cdot)$ 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:

$$\mathrm{vec}\left(\mathbf{A}\right) = \mathrm{vec}\left(\begin{bmatrix} a_0 & a_2 \\ a_1 & a_3 \end{bmatrix}\right) = \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix} \tag{3.8}$$

The important pattern to notice is that we *stacked the columns on top of each other*. We did not stack the rows, though we very well could have. This is merely the convention that we have found most useful. Getting this convention right is sufficiently important that I'm just going to go ahead and give you the Matlab and C++ code that does this the right way (Fig. 3.4). If you're using Octave, it has vec built-in already, and it follows the same convention we do.

```matlab
function [x] = vec(A)
    [rows cols] = size(A);
    x = reshape(A, rows * cols, 1);
end
```

```cpp
static Vector9 flatten(const Matrix3x3& A)
{
    Vector9 flattened;
    int index = 0;
    for (int y = 0; y < 3; y++)
        for (int x = 0; x < 3; x++, index++)
            flattened[index] = A(x, y);
    return flattened;
}
```

Figure 3.4.: Our matrix flattening convention, in Matlab and C++.

Next up, we define the following convention for 3rd-order tensors:

$$\operatorname{vec}(\mathbb{A}) = \operatorname{vec}\begin{bmatrix} \begin{bmatrix}\mathbf{A}\end{bmatrix} \\ \begin{bmatrix}\mathbf{B}\end{bmatrix} \\ \begin{bmatrix}\mathbf{C}\end{bmatrix} \end{bmatrix} = \begin{bmatrix}\operatorname{vec}(\mathbf{A}) & \operatorname{vec}(\mathbf{B}) & \operatorname{vec}(\mathbf{C})\end{bmatrix}. \tag{3.9}$$

First we unfurl the matrices in the 3rd-order tensor into a row, and then vectorize them all in turn. Here's a concrete example:

$$\operatorname{vec}\begin{bmatrix} \begin{bmatrix} a_0 & a_2 \\ a_1 & a_3 \end{bmatrix} \\ \begin{bmatrix} a_4 & a_6 \\ a_5 & a_7 \end{bmatrix} \\ \begin{bmatrix} a_8 & a_{10} \\ a_9 & a_{11} \end{bmatrix} \end{bmatrix} = \begin{bmatrix} \operatorname{vec}\begin{bmatrix} a_0 & a_2 \\ a_1 & a_3 \end{bmatrix} & \operatorname{vec}\begin{bmatrix} a_4 & a_6 \\ a_5 & a_7 \end{bmatrix} & \operatorname{vec}\begin{bmatrix} a_8 & a_{10} \\ a_9 & a_{11} \end{bmatrix} \end{bmatrix} = \begin{bmatrix} a_0 & a_4 & a_8 \\ a_1 & a_5 & a_9 \\ a_2 & a_6 & a_{10} \\ a_3 & a_7 & a_{11} \end{bmatrix} \tag{3.10}$$

It may seem a little antithetical to the convention that we don't just stack everything into one long vector. All the matrix multiplies wouldn't work out in that case though, which would mess up your day even worser.

Finally, let's look at the flattened version of that $\mathbb{A} : \mathbf{B}$ double-contraction we saw in

Eqn. 3.7:

$$(\operatorname{vec}\mathbb{A})^T \operatorname{vec}\mathbf{B} = \left(\operatorname{vec} \begin{bmatrix} \begin{bmatrix} a_0 & a_2 \\ a_1 & a_3 \end{bmatrix} \\ \begin{bmatrix} a_4 & a_6 \\ a_5 & a_7 \end{bmatrix} \\ \begin{bmatrix} a_8 & a_{10} \\ a_9 & a_{11} \end{bmatrix} \end{bmatrix} \right)^T \operatorname{vec} \begin{bmatrix} b_0 & b_2 \\ b_1 & b_3 \end{bmatrix} \tag{3.11}$$

$$= \begin{bmatrix} a_0 & a_4 & a_8 \\ a_1 & a_5 & a_9 \\ a_2 & a_6 & a_{10} \\ a_3 & a_7 & a_{11} \end{bmatrix}^T \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} \tag{3.12}$$

$$= \begin{bmatrix} a_0 b_0 + a_1 b_1 + a_2 b_2 + a_3 b_3 \\ a_4 b_0 + a_5 b_1 + a_6 b_2 + a_7 b_3 \\ a_8 b_0 + a_9 b_1 + a_{10} b_2 + a_{11} b_3 \end{bmatrix}. \tag{3.13}$$

The result is the same as Eqn. 3.7, but arrived at (at least in the final step) through a conventional matrix multiply. It works!

3.4 Computing Forces (Finally)

With all this tensor stuff in place, we can finally compute some forces. Once again, the force is $\mathbf{f} = -a\frac{\partial \Psi}{\partial \mathbf{x}}$, so the equation we need to compute is:

$$\frac{\partial \Psi}{\partial \mathbf{x}} = \frac{\partial \mathbf{F}}{\partial \mathbf{x}} : \frac{\partial \Psi}{\partial \mathbf{F}}. \tag{3.14}$$

The trick is that the 3rd-order tensor, $\frac{\partial \mathbf{F}}{\partial \mathbf{x}}$, has a *static, simple, and energy-independent* structure that we can code up once and forget about. The other term, $\frac{\partial \Psi}{\partial \mathbf{F}}$, which will have a simple and clean structure. If you want to implement a new energy, all you need to do is derive a new $\frac{\partial \Psi}{\partial \mathbf{F}}$.

3.4.1 Computing the $\frac{\partial \mathbf{F}}{\partial \mathbf{x}}$ Tensor

To see where $\mathbf{x}$ appears in $\mathbf{F}$, we're going to use one of the expressions we derived in Appendix D:

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

$$= \left[\begin{array}{c|c|c} \mathbf{x}_1 - \mathbf{x}_0 & \mathbf{x}_2 - \mathbf{x}_0 & \mathbf{x}_3 - \mathbf{x}_0 \end{array} \right] \left[\begin{array}{c|c|c} \bar{\mathbf{x}}_1 - \bar{\mathbf{x}}_0 & \bar{\mathbf{x}}_2 - \bar{\mathbf{x}}_0 & \bar{\mathbf{x}}_3 - \bar{\mathbf{x}}_0 \end{array} \right]^{-1}. \tag{3.16}$$

Fortunately we're differentiating with respect to $\mathbf{x} = \begin{bmatrix} \mathbf{x}_0 \\ \mathbf{x}_1 \\ \mathbf{x}_2 \\ \mathbf{x}_3 \end{bmatrix}$ and not $\bar{\mathbf{x}} = \begin{bmatrix} \bar{\mathbf{x}}_0 \\ \bar{\mathbf{x}}_1 \\ \bar{\mathbf{x}}_2 \\ \bar{\mathbf{x}}_3 \end{bmatrix}$, so we don't have to figure out how to take the derivative of a matrix inverse (yuck). In the end, we're going to get a bunch of expressions that look like:

$$\frac{\partial \mathbf{F}}{\partial x_i} = \frac{\partial \mathbf{D}_s}{\partial x_i} \mathbf{D}_m^{-1}. \tag{3.17}$$

We already have $\mathbf{D}_m^{-1}$ in hand, and $\frac{\partial \mathbf{D}_s}{\partial x_i}$ will work out to a bunch of simple matrices. There will be a lot of them (twelve!), but they will be simple. It takes a lot of space to list these, so instead I've stashed it all in Appendix E.

We could compute things the purely matrix way as well,

$$\frac{\partial \Psi}{\partial \mathbf{x}} = \operatorname{vec}\left(\frac{\partial \mathbf{F}}{\partial \mathbf{x}}\right)^T \operatorname{vec}\left(\frac{\partial \Psi}{\partial \mathbf{F}}\right). \tag{3.18}$$

There's C++ code in Fig. E.1 that explicitly lays out the flattened version of $\operatorname{vec}\left(\frac{\partial \mathbf{F}}{\partial \mathbf{x}}\right)$ for you. I've had to derive this matrix several times already in my lifetime, and I'm getting a little tired of it, so I'm caching it here for posterity. You're welcome, Future Me.

3.5 Dirichlet Forces: So Easy

Back in §2.4.2, we tried to compute forces for the Dirichlet energy,

$$\Psi_{\text{Dirichlet}} = \|\mathbf{F}\|_F^2,$$

but gave up because the expressions rolled out of control the moment you tried to differentiate an $\mathbf{F}$-based energy using $\mathbf{x}$. We've now skimmed off all the difficulty and distilled it into $\frac{\partial \mathbf{F}}{\partial \mathbf{x}}$. All that's left is to take $\frac{\partial \Psi}{\partial \mathbf{F}}$, and this is probably pretty easy for an $\mathbf{F}$-based energy, right?

The matrix $\frac{\partial \Psi}{\partial \mathbf{F}}$ is also known as the **first Piola-Kirchhoff stress tensor**, which is an unwieldy name, so we abbreviate it to the **PK1**, and sometimes write it as $\mathbf{P}(\mathbf{F})$. The PK1 of the Dirichlet energy is:

$$\frac{\partial \Psi_{\text{Dirichlet}}}{\partial \mathbf{F}} = \mathbf{P}(\mathbf{F}) = 2\mathbf{F}. \tag{3.19}$$

SO EASY. It's almost as easy as computing the forces of the spring-mass system in Eqn. 2.25. In fact, it looks like an intro-to-calculus scalar derivative. The $\mathbf{F}^T\mathbf{F}$ expression kind of looks like the matrix version of $\mathbf{F}^2$, and then we took the derivative to get $2\mathbf{F}$. There are indeed a bunch of matrix calculus identities that look like this, and a few useful ones are listed in Appendix B. Now to compute the forces, all we do is call the C++ code I gave you in Fig. E.1, and multiply it by $\operatorname{vec}(2\mathbf{F})$.

3.6 Other Forces: Still Pretty Easy

If we want to compute the forces for some of the other energies we looked at, it becomes straightforward.

3.6.1 St. Venant Kirchhoff, Stretching Only

Let's look at the stretching term for the St. Venant-Kirchhoff model from before:

$$\Psi_{\text{StVK, stretch}} = \|\mathbf{E}\|_F^2, \tag{3.20}$$

where $\mathbf{E} = \frac{1}{2}\left(\mathbf{F}^T\mathbf{F} - \mathbf{I}\right)$. If we learned any lesson before, it's that we should take the derivative with respect to whatever basic variable the energy is written in terms of ($\mathbf{F}$, for the Dirichlet case), and then bake out all the difficult change-of-variable stuff into some other tensor, $\frac{\partial \mathbf{F}}{\partial \mathbf{x}}$.

The basic variable for this energy is sure seems like $\mathbf{E}$, not $\mathbf{F}$. We could indeed write things out as $\frac{\partial \Psi}{\partial \mathbf{x}} = \frac{\partial \Psi}{\partial \mathbf{E}}\frac{\partial \mathbf{E}}{\partial \mathbf{x}}$. Then we'd have to derive a new change-of-basis tensor, $\frac{\partial \mathbf{E}}{\partial \mathbf{x}}$. But, once we had that in hand, computing $\frac{\partial \Psi}{\partial \mathbf{E}}$, otherwise known as the *second Piola-Kirchhoff stress tensor (PK2)* would be really easy:

$$\frac{\partial \Psi_{\text{StVK, stretch}}}{\partial \mathbf{E}} = \mathbf{E}. \tag{3.21}$$

Before we go down the PK2 road though, you should know that *StVK is the only energy that really uses* $\mathbf{E}$. All the other energies[5] use $\mathbf{F}$. So, before we build a customized $\frac{\partial \mathbf{E}}{\partial \mathbf{x}}$ tensor for StVK, let's see how difficult it is to go down the $\mathbf{F}$ road.

$$\begin{aligned}
\Psi_{\text{StVK, stretch}} &= \frac{1}{4}\|\mathbf{F}^T\mathbf{F} - \mathbf{I}\|_F^2 \\
&= \frac{1}{4}\left(\|\mathbf{F}^T\mathbf{F}\|_F^2 + \operatorname{tr}\mathbf{I} - 2\operatorname{tr}(\mathbf{F}^T\mathbf{F})\right) \\
&= \frac{1}{4}\left(\|\mathbf{F}^T\mathbf{F}\|_F^2 + \operatorname{tr}\mathbf{I} - 2\|\mathbf{F}\|_F^2\right).
\end{aligned} \tag{3.22}$$

We already know how to take the derivative of $\|\mathbf{F}\|_F^2$, since it's just the Dirichlet energy, and the derivative of $\|\mathbf{F}^T\mathbf{F}\|_F^2$ is in Appendix B.

$$\begin{aligned}
\mathbf{P}_{\text{StVK, stretch}}(\mathbf{F}) &= \frac{1}{4}\left(4\mathbf{F}\mathbf{F}^T\mathbf{F} - 4\mathbf{F}\right) \\
&= \mathbf{F}\left(\mathbf{F}^T\mathbf{F} - \mathbf{I}\right) \\
&= \mathbf{F}\mathbf{E}
\end{aligned} \tag{3.23}$$

STILL PRETTY EASY. The pattern holds here where $\|\mathbf{F}^T\mathbf{F}\|_F^2$ looks suspiciously like a matrix version of $\mathbf{F}^4$, and its derivative $4\mathbf{F}\mathbf{F}^T\mathbf{F}$ looks an awful lot like $4\mathbf{F}^3$.

[5]The ones I care about, anyway.

3.6.2 The Complete St. Venant Kirchhoff

Up until now, we've only looked at the stretching term from StVK, but now is a good moment to look at the complete energy:

$$\Psi_{\text{StVK}} = \mu \|\mathbf{E}\|_F^2 + \frac{\lambda}{2} (\operatorname{tr} \mathbf{E})^2 \tag{3.24}$$

The $\|\mathbf{E}\|_F^2$ term we've been examining is there of course, but a new volume preservation term $(\operatorname{tr} \mathbf{E})^2$ has been added[6]. The constants μ and λ are present to you can tell the model how much relative stretching resistance vs. volume preservation you want. The PK1 is then:

$$\mathbf{P}_{\text{StVK}}(\mathbf{F}) = \mu \mathbf{F}\mathbf{E} + \lambda (\operatorname{tr} \mathbf{E})\, \mathbf{F}. \tag{3.25}$$

To arrive at this, we used identities B.12 and B.32 from Appendix B. Some minor gymnastics are required if you're not used to this sort of thing, but the final expression is still short and easy to digest.

3.6.3 As-Rigid-As-Possible

The As-Rigid-As-Possible (ARAP) energy can be expanded to:

$$\begin{aligned}
\Psi_{\text{ARAP}} &= \frac{\mu}{2} \|\mathbf{F} - \mathbf{R}\|_F^2 && (3.26)\\
&= \frac{\mu}{2} \left(\|\mathbf{F}\|_F^2 + \|\mathbf{R}\|_F^2 - 2 \operatorname{tr} \left(\mathbf{F}^T \mathbf{R} \right) \right) && (3.27)\\
&= \frac{\mu}{2} \left(\|\mathbf{F}\|_F^2 + \|\mathbf{R}\|_F^2 - 2 \operatorname{tr} \mathbf{S} \right) && (3.28)
\end{aligned}$$

We used identity B.16 and the polar decomposition $\mathbf{F} = \mathbf{R}\mathbf{S} \implies \mathbf{S} = \mathbf{F}^T \mathbf{R}$ (since $\mathbf{S}$ is symmetric) to arrive at this form. The PK1 is then:

$$\mathbf{P}_{\text{ARAP}}(\mathbf{F}) = \mu(\mathbf{F} - \mathbf{R}) \tag{3.29}$$

We used identities B.19 anad B.34 to arrive at this expression. This one looks quite appealing! We just extract the expression inside the Frobenius norm, and that becomes the PK1. This is extremely similar to what we saw in §2.4.3 with the spring-mass system, where we just extracted the vector from inside the 2-norm. This is not a coincidence, because as we'll see later, ARAP has quite a strong claim on the title of "Most-Spring-Mass-Like in $\mathbf{F}$-based World".

3.6.4 The Many Forms of Neo-Hookean

Back in §2.3.4.1, we looked at the Neo-Hookean energy:

$$\Psi_{\text{Neo-Hookean}} = \|\mathbf{F}\|_F^2 - 3. \tag{3.30}$$

[6] We'll see later that it doesn't actually preserve volume that well if you deform the object too much.

If we get the PK1 of this energy,

$$\mathbf{P}_{\text{Neo-Hookean}}(\mathbf{F}) = 2\mathbf{F}, \tag{3.31}$$

it's *exactly the same* PK1 as the Dirichlet energy. The only difference between this energy and Dirichlet was the minus three, and that gets burned off under differentiation. From any practical force-generating standpoint, this energy is equivalent to Dirichlet.

Why does it even exist then? In the original paper by Mooney (1940), an additional equation appeared in the form of a hard constraint, $\det \mathbf{F} = 1$. We're not going to look at methods like this, because this constraint is too severe for the kinds of deformations we see in computer animation.

There are many energies that call themselves "Neo-Hookean", so we'll look at one of the more popular ones from Bonet and Wood (2008):

$$\Psi_{\text{BW08}} = \frac{\mu}{2}(\|\mathbf{F}\|_F^2 - 3) - \mu \log(J) + \frac{\lambda}{2}\left(\log(J)\right)^2. \tag{3.32}$$

I've put this off as long as possible, but here we are starting to use the shorthand of $J = \det \mathbf{F}$. This $\det \mathbf{F}$ term is extremely important because it tells us how much volume has or hasn't been preserved. As we'll see later, any energy that claims to be "volume preserving" but doesn't contain $\det \mathbf{F}$ is just a sham pretender to the throne.

Since $\det \mathbf{F}$ is going to start showing up a lot, it's time we started using a symbol that's simpler to read, and J is extremely common elsewhere in the literature (e.g. Bonet and Wood (2008), Marsden and Hughes (1994), Bower (2009)).

The PK1 of Ψ_{BW08} is then:

$$\mathbf{P}_{\text{BW08}}(\mathbf{F}) = \mu\left(\mathbf{F} - \frac{1}{J}\frac{\partial J}{\partial \mathbf{F}}\right) + \lambda \frac{\log J}{J}\frac{\partial J}{\partial \mathbf{F}}. \tag{3.33}$$

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

$$\mathbf{F} = \left[\begin{array}{c|c|c} \mathbf{f}_0 & \mathbf{f}_1 & \mathbf{f}_2 \end{array}\right]. \tag{3.34}$$

The gradient of J is then the pair-wise cross-products of these columns,

$$\frac{\partial J}{\partial \mathbf{F}} = \left[\begin{array}{c|c|c} \mathbf{f}_1 \times \mathbf{f}_2 & \mathbf{f}_2 \times \mathbf{f}_0 & \mathbf{f}_0 \times \mathbf{f}_1 \end{array}\right]. \tag{3.35}$$

This identity is important enough that we stacked it into Appendix B, so when you need it later, you can find it there too. Once you have this identity in hand, deriving the PK1 for an energy that contains a $\det \mathbf{F} \equiv J$ term is not a big deal. Take scalar derivatives like usual, and then you finally hit a $\frac{\partial J}{\partial \mathbf{F}}$ you can stop, because we've got an identity for that.

As we'll see later, the $\frac{\partial J}{\partial \mathbf{F}}$ term appears in lots of places. Just to underscore how important it is, I'm giving you a Matlab implementation in Fig. 3.5

```
function [final] = gradientJ(F)
  f0 = F(:,1);
  f1 = F(:,2);
  f2 = F(:,3);
  final = [cross(f1,f2) cross(f2,f0) cross(f0,f1)];
end
```

Figure 3.5.: Matlab/Octave code for the 3D version of $\frac{\partial J}{\partial \mathbf{F}}$ from Eqn. 5.6.

3.7 What Now?

We've now seen how to compute forces the tensor way. This is enough to get a simulator off the ground that does explicit time integration, but that's not good enough. We want implicit timestepping, since the Fizt integrator from Baraff and Witkin (1998), and the large timesteps it enables, is our target deployment.

For that, we need *force gradients*, i.e. $\frac{\partial \mathbf{f}}{\partial \mathbf{x}}$. All the tensor stuff we just saw made force computation easier and prettier, but all the ulcer-inducing ugliness comes rushing back the moment we try to take a force gradient. It'll take some more work to make force gradient computation equivalently easy and pretty. Let's take a look at that next.

Chapter 4

Computing Force Gradients the Tensor Way

If you're just joining us, we're looking at Baraff and Witkin (1998)-style implicit integration and how it uses force gradients to enable large, stable timesteps. If you thought that computing forces directly from the strain energy was painful,

$$\mathbf{f} = -v\frac{\partial \Psi}{\partial \mathbf{x}}, \tag{4.1}$$

then, hoo nelly, are these force gradients excruciating:

$$\frac{\partial \mathbf{f}}{\partial \mathbf{x}} = -v\frac{\partial^2 \Psi}{\partial \mathbf{x}^2}. \tag{4.2}$$

The way we cleaned everything up in Chapter 3 was to apply the chain rule,

$$\frac{\partial \Psi}{\partial \mathbf{x}} = \frac{\partial \mathbf{F}}{\partial \mathbf{x}} : \frac{\partial \Psi}{\partial \mathbf{F}}, \tag{4.3}$$

which baked all the difficulty into an energy-agnostic tensor $\frac{\partial \mathbf{F}}{\partial \mathbf{x}}$ that we only have to derive once. Then for each energy model, we only have to deal with the relatively easier $\frac{\partial \Psi}{\partial \mathbf{F}}$, a.k.a the first Piola-Kirchhoff stress tensor (PK1).

Can we do something similar for the force gradient? Something like

$$\frac{\partial^2 \Psi}{\partial \mathbf{x}^2} = \frac{\partial \mathbf{F}}{\partial \mathbf{x}}^T \frac{\partial^2 \Psi}{\partial \mathbf{F}^2} \frac{\partial \mathbf{F}}{\partial \mathbf{x}} \tag{4.4}$$

sure seems promising. But just like in Chapter 3, you'll discover that the whole thing starts looking like a clear-as-mud, dirty professor trick as soon as you try to apply this equation. Let's look at it in more detail, because it'll take some work to actually use it.

4.1 4th-order Tensors

The first observation that will make you curse the sky is that $\frac{\partial^2 \Psi}{\partial \mathbf{F}^2}$ is a *4^{th}-order tensor*. To see this more clearly, recall that we started writing $\frac{\partial \Psi}{\partial \mathbf{F}}$ as the matrix $\mathbf{P}(\mathbf{F}) \in \Re^{3\times 3}$. We can write $\frac{\partial^2 \Psi}{\partial \mathbf{F}^2}$ as $\frac{\partial \mathbf{P}}{\partial \mathbf{F}}$[1], which is the derivative of a matrix with respect to a matrix. We can write this out explicitly as:

$$\frac{\partial \mathbf{P}}{\partial \mathbf{F}} = \begin{bmatrix} \left[\frac{\partial \mathbf{P}}{\partial f_{00}}\right] & \left[\frac{\partial \mathbf{P}}{\partial f_{01}}\right] & \left[\frac{\partial \mathbf{P}}{\partial f_{02}}\right] \\ \left[\frac{\partial \mathbf{P}}{\partial f_{10}}\right] & \left[\frac{\partial \mathbf{P}}{\partial f_{11}}\right] & \left[\frac{\partial \mathbf{P}}{\partial f_{12}}\right] \\ \left[\frac{\partial \mathbf{P}}{\partial f_{20}}\right] & \left[\frac{\partial \mathbf{P}}{\partial f_{21}}\right] & \left[\frac{\partial \mathbf{P}}{\partial f_{22}}\right] \end{bmatrix}. \tag{4.5}$$

As in the 3rd-order case, we put brackets around each $\frac{\partial \mathbf{P}}{\partial f_{ij}}$ to emphasize that they are in the entries in a higher-order tensor, not the blocks in a traditional 2nd-order matrix.

Just like the 3rd-order tensor was a *vector of matrices*, a 4th-order tensor is a *matrix of matrices*. Instead of a cutting board containing a line of cheese slices, the cutting board now has cheese slices *arranged in a grid* (Fig. 4.1).

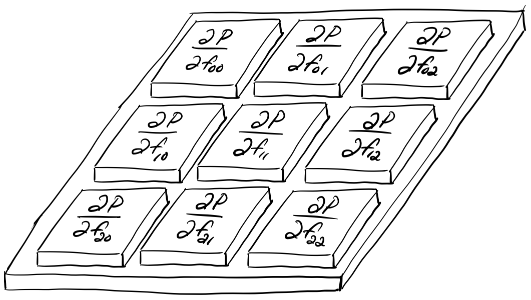

$\frac{\partial P}{\partial f_{00}}$
$\frac{\partial P}{\partial f_{01}}$
$\frac{\partial P}{\partial f_{02}}$
$\frac{\partial P}{\partial f_{10}}$
$\frac{\partial P}{\partial f_{11}}$
$\frac{\partial P}{\partial f_{12}}$
$\frac{\partial P}{\partial f_{20}}$
$\frac{\partial P}{\partial f_{21}}$
$\frac{\partial P}{\partial f_{22}}$

Figure 4.1.: A 4th-order tensor: like a cutting board covered with a *grid* of individually wrapped slices of American cheese. A matrix of matrices.

A brief digression: this mental picture does indeed generalize to even higher orders. If you ever needed the next derivative $\frac{\partial^3 \Psi}{\partial \mathbf{F}^3}$, you would need to compute a 6th-order tensor, i.e. a matrix of matrices of matrices.[2] You can just stack another layer on top (Fig. 4.2). For 8th-order, you would take four copies and arrange them in a grid again. And so on to ∞^{th}-order, like a giant fractal pagoda.

[1]The $(\mathbf{F})$ argument of $\mathbf{P}$ has been dropped for brevity.
[2]We won't cover it here, but I have needed this before, so you may encounter a case where you do too.

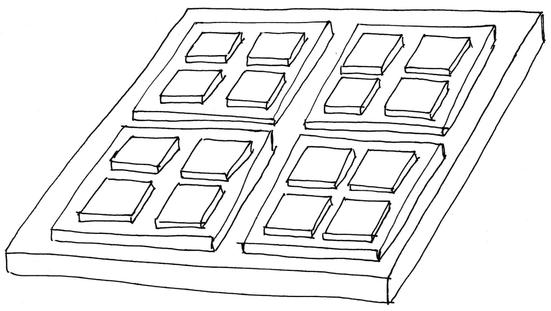


Figure 4.2.: A 6th-order tensor: we take the cutting boards of cheese from Fig. 4.1, make four copies, get an even bigger cutting board for underneath, and arrange them all into another grid. So much cheddar.

Now back to the 4th-order case. Double-contraction looks similar to the 3rd-order case,

$$\begin{bmatrix} \begin{bmatrix} a_0 & a_2 \\ a_1 & a_3 \end{bmatrix} & \begin{bmatrix} a_8 & a_{10} \\ a_9 & a_{11} \end{bmatrix} \\ \begin{bmatrix} a_4 & a_6 \\ a_5 & a_7 \end{bmatrix} & \begin{bmatrix} a_{12} & a_{14} \\ a_{13} & a_{15} \end{bmatrix} \end{bmatrix} : \begin{bmatrix} b_0 & b_2 \\ b_1 & b_3 \end{bmatrix} = \begin{bmatrix} (a_0b_0 + a_1b_1 + a_2b_2 + a_3b_3) & (a_8b_0 + a_9b_1 + a_{10}b_2 + a_{11}b_3) \\ (a_4b_0 + a_5b_1 + a_6b_2 + a_7b_3) & (a_{12}b_0 + a_{13}b_1 + a_{14}b_2 + a_{15}b_3) \end{bmatrix}, \tag{4.6}$$

except instead of producing a vector, the results are arranged into a matrix.

If we want to flatten the following tensor,

$$\mathbb{C} = \begin{bmatrix} \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix} & \begin{bmatrix} 9 & 11 \\ 10 & 12 \end{bmatrix} \\ \begin{bmatrix} 5 & 7 \\ 6 & 8 \end{bmatrix} & \begin{bmatrix} 13 & 15 \\ 14 & 16 \end{bmatrix} \end{bmatrix} \tag{4.7}$$

$$= \begin{bmatrix} [\mathbf{C}_{00}] & [\mathbf{C}_{01}] \\ [\mathbf{C}_{10}] & [\mathbf{C}_{11}] \end{bmatrix} \tag{4.8}$$

then we first flatten in *column-wise* order, and then each matrix in turn:

$$\mathrm{vec}(\mathbb{C}) = \left[\begin{array}{c|c|c|c} \mathrm{vec}(\mathbf{C}_{00}) & \mathrm{vec}(\mathbf{C}_{10}) & \mathrm{vec}(\mathbf{C}_{01}) & \mathrm{vec}(\mathbf{C}_{11}) \end{array} \right] \tag{4.9}$$

$$= \begin{bmatrix} 1 & 5 & 9 & 13 \\ 2 & 6 & 10 & 14 \\ 3 & 7 & 11 & 15 \\ 4 & 8 & 12 & 16 \end{bmatrix}. \tag{4.10}$$

One thing that we have *not* defined is how to multiply a 4th-order tensor with a 3rd-order tensor. Rather than do that, we are instead going to flatten everything so that it all reduces to a matrix-matrix multiply. Not only will this be easier to think about, but we will see some interesting structures appear.

4.2 Computing Force Gradients

We now have enough operators in hand that we can write the useful-looking chain rule that we saw before in more concrete terms:

$$\frac{\partial^2 \Psi}{\partial \mathbf{x}^2} = \operatorname{vec}\left(\frac{\partial \mathbf{F}}{\partial \mathbf{x}}\right)^T \operatorname{vec}\left(\frac{\partial^2 \Psi}{\partial \mathbf{F}^2}\right) \operatorname{vec}\left(\frac{\partial \mathbf{F}}{\partial \mathbf{x}}\right). \tag{4.11}$$

We already know how to compute $\operatorname{vec}\left(\frac{\partial \mathbf{F}}{\partial \mathbf{x}}\right)$ from Appendix E, and in fact have the C++ code for it Fig. E.1. The only term that remains to be derived is $\operatorname{vec}\left(\frac{\partial^2 \Psi}{\partial \mathbf{F}^2}\right)$. We will abbreviate this matrix as $\mathbf{H}$, because it represents the *energy Hessian*.

4.2.1 Dirichlet Hessian: So Easy

Let's try it out on the simplest energy we know, the Dirichlet energy:

$$\Psi_{\text{Dirichlet}} = \|\mathbf{F}\|_F^2$$
$$\frac{\partial \Psi_{\text{Dirichlet}}}{\partial \mathbf{F}} = \mathbf{P} = 2\mathbf{F}.$$

Each entry $\frac{\partial \mathbf{P}}{\partial f_{ij}}$ in the 4th-order tensor $\frac{\partial \mathbf{P}}{\partial \mathbf{F}}$ is then:

$$\frac{\partial \mathbf{P}}{\partial f_{ij}} = 2\frac{\partial \mathbf{F}}{\partial f_{ij}} \tag{4.12}$$

What does each $\frac{\partial \mathbf{F}}{\partial f_{ij}}$ look like? There's a lot of them (nine), but they are all quite simple:

$$\frac{\partial \mathbf{F}}{\partial f_{00}} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad \frac{\partial \mathbf{F}}{\partial f_{01}} = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad \frac{\partial \mathbf{F}}{\partial f_{02}} = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$
$$\frac{\partial \mathbf{F}}{\partial f_{10}} = \begin{bmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad \frac{\partial \mathbf{F}}{\partial f_{11}} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad \frac{\partial \mathbf{F}}{\partial f_{12}} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}$$
$$\frac{\partial \mathbf{F}}{\partial f_{20}} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix} \quad \frac{\partial \mathbf{F}}{\partial f_{21}} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \quad \frac{\partial \mathbf{F}}{\partial f_{22}} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

Each just has a one at the corresponding entry (i, j). Here's what we get when we flatten this out:

$$\operatorname{vec}\frac{\partial \mathbf{F}}{\partial \mathbf{F}} = \begin{bmatrix} 1 &&&&&&&& \\ & 1 &&&&&&& \\ && 1 &&&&&& \\ &&& 1 &&&&& \\ &&&& 1 &&&& \\ &&&&& 1 &&& \\ &&&&&& 1 && \\ &&&&&&& 1 & \\ &&&&&&&& 1 \end{bmatrix} \tag{4.13}$$

IT'S JUST IDENTITY. SO EASY. The Hessian of the Dirichlet energy is then:

$$\operatorname{vec}\left(\frac{\partial \mathbf{P}}{\partial \mathbf{F}}\right) = 2\mathbf{I}_{9\times 9}. \tag{4.14}$$

When we flatten out Hessians, we'll often see patterns like this appear. Let's see how far we can take it.

4.2.2 Neo-Hookean: Not So Easy

Let's look at the Bonet and Wood (2008) version of Neo-Hookean again:

$$\Psi_{\text{BW08}} = \frac{\mu}{2}(\|\mathbf{F}\|_F^2 - 3) - \mu\log(J) + \frac{\lambda}{2}\left(\log(J)\right)^2 \tag{4.15}$$

$$\mathbf{P} = \mu\left(\mathbf{F} - \frac{1}{J}\frac{\partial J}{\partial \mathbf{F}}\right) + \lambda\frac{\log J}{J}\frac{\partial J}{\partial \mathbf{F}}. \tag{4.16}$$

We're going to have to derive several grisly expressions to get its flattened Hessian, but it will all be worth it, because these same expressions will appear in *lots* of different energies. In the moment, this will all feel like some sadistic exercise, but in the end, I promise that we will have forged a tool that can be used over and over.

Let's start by trying to compute *one* matrix-valued entry in its 4th-order tensor:

$$\frac{\partial \mathbf{P}}{\partial f_{ij}} = \mu\frac{\partial \mathbf{F}}{\partial f_{ij}} + \frac{\mu}{J^2}\frac{\partial J}{\partial f_{ij}}\frac{\partial J}{\partial \mathbf{F}} - \frac{\mu}{J}\frac{\partial^2 J}{\partial \mathbf{F}\partial f_{ij}} + \frac{\lambda\log J}{J}\frac{\partial^2 J}{\partial \mathbf{F}\partial f_{ij}} - \frac{\lambda\log J}{J^2}\frac{\partial J}{\partial f_{ij}}\frac{\partial J}{\partial \mathbf{F}} + \frac{\lambda}{J^2}\frac{\partial J}{\partial f_{ij}}\frac{\partial J}{\partial \mathbf{F}}.$$

Not very clean. However, if we move things around, we can make things slightly nicer:

$$\frac{\partial \mathbf{P}}{\partial f_{ij}} = \mu\frac{\partial \mathbf{F}}{\partial f_{ij}} + \left[\frac{\mu + \lambda(1-\log J)}{J^2}\right]\frac{\partial J}{\partial f_{ij}}\frac{\partial J}{\partial \mathbf{F}} + \left[\frac{\lambda\log J - \mu}{J}\right]\frac{\partial^2 J}{\partial \mathbf{F}\partial f_{ij}}.$$

It's still not that nice, so I'll go ahead and put boxes around the non-scalar terms you should care about:

$$\frac{\partial \mathbf{P}}{\partial f_{ij}} = \mu\boxed{\frac{\partial \mathbf{F}}{\partial f_{ij}}} + \left[\frac{\mu + \lambda(1-\log J)}{J^2}\right]\boxed{\frac{\partial J}{\partial f_{ij}}\frac{\partial J}{\partial \mathbf{F}}} + \left[\frac{\lambda\log J - \mu}{J}\right]\boxed{\frac{\partial^2 J}{\partial \mathbf{F}\partial f_{ij}}}.$$

The $\mu \frac{\partial \mathbf{F}}{\partial f_{ij}}$ term is relatively friendly-looking. If we enumerate all nine copies of this in full 4th-order tensor form, we get $\mu \frac{\partial \mathbf{F}}{\partial \mathbf{F}}$, and if we flatten that out, we get the same easy-looking term as the Dirichlet energy: $\operatorname{vec}\left(\mu \frac{\partial \mathbf{F}}{\partial \mathbf{F}}\right) = \mu \mathbf{I}_{9\times 9}$.

There are two other scalar coefficients: $\left[\frac{\mu + \lambda(1-\log J)}{J^2}\right]$ and $\left[\frac{\lambda \log J - \mu}{J}\right]$. They're not pretty, but at least they're scalar-valued. If we can figure out what the flattened versions of the 4th-order tensors $\frac{\partial J}{\partial f_{ij}} \frac{\partial J}{\partial \mathbf{F}}$ and $\frac{\partial^2 J}{\partial \mathbf{F} \partial f_{ij}}$ look like, then we'll have the complete 9×9 matrix in hand, and we're in business.

4.2.2.1 *The Gradient of J*

First up, the $\frac{\partial J}{\partial f_{ij}} \frac{\partial J}{\partial \mathbf{F}}$ term. Let's call it $\mathbb{G}_J$ in 4th-order form, since it involves two $\mathbb{G}$radients of J:

$$\mathbb{G}_J = \begin{bmatrix} \left[\frac{\partial J}{\partial f_{00}} \frac{\partial J}{\partial \mathbf{F}}\right] & \left[\frac{\partial J}{\partial f_{01}} \frac{\partial J}{\partial \mathbf{F}}\right] & \left[\frac{\partial J}{\partial f_{02}} \frac{\partial J}{\partial \mathbf{F}}\right] \\ \left[\frac{\partial J}{\partial f_{10}} \frac{\partial J}{\partial \mathbf{F}}\right] & \left[\frac{\partial J}{\partial f_{11}} \frac{\partial J}{\partial \mathbf{F}}\right] & \left[\frac{\partial J}{\partial f_{12}} \frac{\partial J}{\partial \mathbf{F}}\right] \\ \left[\frac{\partial J}{\partial f_{20}} \frac{\partial J}{\partial \mathbf{F}}\right] & \left[\frac{\partial J}{\partial f_{21}} \frac{\partial J}{\partial \mathbf{F}}\right] & \left[\frac{\partial J}{\partial f_{22}} \frac{\partial J}{\partial \mathbf{F}}\right] \end{bmatrix}. \tag{4.17}$$

Writing this out in its full gruesome form doesn't add much, does it? However, when we perform the vectorization, something friendlier-looking appears:

$$\operatorname{vec} \mathbb{G}_J = \operatorname{vec}\left(\frac{\partial J}{\partial \mathbf{F}}\right) \operatorname{vec}\left(\frac{\partial J}{\partial \mathbf{F}}\right)^T. \tag{4.18}$$

It's just the outer product of the gradient of J! If you don't remember what outer products look like, don't fret, they're reviewed in Appendix C.4. With this in hand, we can go ahead and define

$$\mathbf{g}_J = \operatorname{vec}\left(\frac{\partial J}{\partial \mathbf{F}}\right),$$

i.e. the flattened gradient of J, and make the whole thing look even cleaner:

$$\operatorname{vec} \mathbb{G}_J = \mathbf{g}_J \mathbf{g}_J^T. \tag{4.19}$$

The flattened version of that gnarly-looking $\left[\frac{\mu + \lambda(1-\log J)}{J^2}\right] \frac{\partial J}{\partial f_{ij}} \frac{\partial J}{\partial \mathbf{F}}$ from before is just:

$$\left[\frac{\mu + \lambda(1-\log J)}{J^2}\right] \mathbf{g}_J \mathbf{g}_J^T.$$

The scalar in front is still ugly, but the rest now looks more like standard linear algebra.

4.2.2.2 *The Hessian of J*

Next up, the $\frac{\partial^2 J}{\partial \mathbf{F} \partial f_{ij}}$ term, the Hessian of J. We will refer to its full 4th-order form as $\frac{\partial^2 J}{\partial \mathbf{F}^2}$.

This Hessian is a little trickier, because it has a different form in 2D and 3D. Let's look at the 2D version first, because a *really easy* structure appears in that case.

The 2D version: Remember that $J = \det \mathbf{F}$, so in 2D

$$J = f_{00}f_{11} - f_{10}f_{01},$$

and the gradient then becomes

$$\frac{\partial J}{\partial \mathbf{F}} = \begin{bmatrix} f_{11} & -f_{10} \\ -f_{01} & f_{00} \end{bmatrix}.$$

The 4th-order tensor can then be written out:

$$\frac{\partial^2 J}{\partial \mathbf{F}^2} = \begin{bmatrix} \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix} & \begin{bmatrix} 0 & 0 \\ -1 & 0 \end{bmatrix} \\ \begin{bmatrix} 0 & -1 \\ 0 & 0 \end{bmatrix} & \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} \end{bmatrix}. \tag{4.20}$$

The flattened form is then a *really simple anti-diagonal matrix:*

$$\operatorname{vec}\left(\frac{\partial^2 J}{\partial \mathbf{F}^2}\right) = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & -1 & 0 \\ 0 & -1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}. \tag{4.21}$$

Just like we abbreviated the **g**radient of J to $\mathbf{g}_J$, let's call this matrix $\mathbf{H}_J$, since it's the **H**essian of J:

$$\mathbf{H}_J = \operatorname{vec}\left(\frac{\partial^2 J}{\partial \mathbf{F}^2}\right). \tag{4.22}$$

That's the 2D version.

The 3D version: In 3D, things get messier, but it also reveals a more interesting structure. For this version, we first to have to define an operator that converts a vector into a cross-product matrix:

$$\hat{\mathbf{x}} = \begin{bmatrix} 0 & -x_2 & x_1 \\ x_2 & 0 & -x_0 \\ -x_1 & x_0 & 0 \end{bmatrix}. \tag{4.23}$$

The full 3D Hessian takes a long time to work out, but after lots of symbolic-machete-slashing, we find this form:

$$\mathbf{H}_J = \begin{bmatrix} \mathbf{0}_{3\times3} & -\hat{\mathbf{f}}_2 & \hat{\mathbf{f}}_1 \\ \hat{\mathbf{f}}_2 & \mathbf{0}_{3\times3} & -\hat{\mathbf{f}}_0 \\ -\hat{\mathbf{f}}_1 & \hat{\mathbf{f}}_0 & \mathbf{0}_{3\times3} \end{bmatrix}. \tag{4.24}$$

```
function [H] = hessianJ(F)
  zero3 = zeros(3,3);
  f0 = F(:,1);
  f1 = F(:,2);
  f2 = F(:,3);
  f0hat = crossMatrix(f0);
  f1hat = crossMatrix(f1);
  f2hat = crossMatrix(f2);
  H = [ zero3 -f2hat  f1hat;
        f2hat  zero3 -f0hat;
       -f1hat  f0hat  zero3];
end

function [final] = crossMatrix(f)
  final = [    0 -f(3)  f(2);
            f(3)     0 -f(1);
           -f(2)  f(1)     0];
end
```

Figure 4.3.: Matlab/Octave code for the 3D version of $\text{vec}\left(\frac{\partial^2 J}{\partial \mathbf{F}^2}\right) = \mathbf{H}_J$ from Eqn. 5.7.

The $\mathbf{0}$s above are all 3×3 zero matrices, and $\mathbf{f}_i$ refer to the columns of $\mathbf{F}$. Now we see the interesting structure. The matrix $\mathbf{H}_J$ is full of cross-product matrices, but compare its structure to Eqn. 4.23. Its block structure follows the cross-product form! It's a *block cross-product matrix of cross-product matrices*, or in other words *a fractal cross-product*. What does it mean? Is there some basic property of Kronecker products that gives rise to this structure? I'm not sure. Sure is interesting though.[3]

Coding this matrix up right can be somewhat delicate, so I've provided a reference Matlab implementation for you in Fig. 4.3.

4.2.2.3 *The Full Hessian*

Coming back to the Hessian of the Neo-Hookean energy:

$$\frac{\partial \mathbf{P}}{\partial f_{ij}} = \boxed{\mu \frac{\partial \mathbf{F}}{\partial f_{ij}}} + \left[\frac{\mu + \lambda(1 - \log J)}{J^2}\right] \boxed{\frac{\partial J}{\partial f_{ij}} \frac{\partial J}{\partial \mathbf{F}}} + \left[\frac{\lambda \log J - \mu}{J}\right] \boxed{\frac{\partial^2 J}{\partial \mathbf{F} \partial f_{ij}}}.$$

Now we have all the pieces to write the boxed terms out in flattened form:

$$\text{vec}\left(\frac{\partial \mathbf{P}}{\partial \mathbf{F}}\right) = \mu \mathbf{I}_{9\times 9} + \left[\frac{\mu + \lambda(1 - \log J)}{J^2}\right] \mathbf{g}_J \mathbf{g}_J^T + \left[\frac{\lambda \log J - \mu}{J}\right] \mathbf{H}_J. \tag{4.25}$$

Not *too* bad. We're still stuck with the ugly scalar coefficients, but at least the matrix terms, $\mathbf{I}_{9\times 9}$, $\mathbf{g}_J$ and $\mathbf{H}_J$ all have compact forms. We can definitely build this, left- and right-multiply it with $\frac{\partial \mathbf{F}}{\partial \mathbf{x}}$, and get the force gradient.

[3]To my knowledge, our paper Smith et al. (2018) was the first to observe this structure. Maybe 4D versions of J will create another fractal level? Sounds like a good homework problem.

You may start to worry here: am I going to have to go through this process for every stinking energy? Will there be some new and potentially more hideous $\mathbf{g}_*$ and $\mathbf{H}_*$ terms that I'll need to derive every single time? Mercifully, the answer is no.

As we'll see later, there are a very small number of $\mathbf{g}$ and $\mathbf{H}$ terms that can ever appear, and $\mathbf{g}_J$ and $\mathbf{H}_J$ are actually the worst of them. Ugliness will appear elsewhere, but it won't be because there's an infinite explosion of g- and $\mathbf{H}$-like terms that you need to tediously arrange every single time.

4.2.3 St. Venant-Kirchhoff: Things Get Worse

Let's try another energy. This time, we'll do St. Venant-Kirchhoff, but only the stretching term, because even then, things will get ugly really fast. The energy and its PK1 are:

$$\begin{aligned}\Psi_{\text{StVK, stretch}} &= \|\mathbf{E}\|_F^2 \\ \mathbf{P} &= \mathbf{F}\mathbf{E} = \mathbf{F}(\mathbf{F}^T\mathbf{F} - \mathbf{I}) \\ &= \mathbf{F}\mathbf{F}^T\mathbf{F} - \mathbf{F}.\end{aligned}$$

We're going to see with this energy, we need to introduce the concept of *invariants* to get at its flattened Hessian. This concept will become all-consumingly important in §5, so let's initially dip our toe into the topic by seeing how it crops up in StVK.

Once again, let's derive a matrix entry of the 4th-order tensor:

$$\frac{\partial \mathbf{P}}{\partial f_{ij}} = \left(\frac{\partial \mathbf{F}}{\partial f_{ij}}\mathbf{F}^T\mathbf{F} + \mathbf{F}\frac{\partial \mathbf{F}}{\partial f_{ij}}^T\mathbf{F} + \mathbf{F}\mathbf{F}^T\frac{\partial \mathbf{F}}{\partial f_{ij}}\right) - \frac{\partial \mathbf{F}}{\partial f_{ij}} \tag{4.26}$$

On the right, the $\frac{\partial \mathbf{F}}{\partial f_{ij}}$ is our friend the Dirichlet term, whose flattened Hessian will at least contain an easy $\frac{\mu}{2}\mathbf{I}_{9\times 9}$. The rest of the terms cause more trouble.

To make further progress, we're going to need the *Cauchy-Green invariants*[4]. The invariants are a set of *three* scalars that often arise when dealing with hyperelastic energies. We've actually already seen the first invariant:

$$I_{\mathbf{C}} = \|\mathbf{F}\|_F^2$$

We've plumbed its depths already by getting all its derivatives:

$$\frac{\partial I_{\mathbf{C}}}{\partial \mathbf{F}} = 2\mathbf{F} \qquad \frac{\partial^2 I_{\mathbf{C}}}{\partial \mathbf{F}\partial f_{ij}} = 2\frac{\partial \mathbf{F}}{\partial f_{ij}} \qquad \operatorname{vec}\left(\frac{\partial^2 I_{\mathbf{C}}}{\partial \mathbf{F}^2}\right) = 2\mathbf{I}_{9\times 9}.$$

It's just the Dirichlet energy! The new trouble we're seeing arises from the *second* Cauchy-Green invariant:

$$II_{\mathbf{C}} = \|\mathbf{F}^T\mathbf{F}\|_F^2$$

[4]Technically the *right* Cauchy-Green invariants, but we'll drop the "right" here, because it's not relevant.

The derivatives we're currently slashing through are:

$$\frac{\partial II_{\mathbf{C}}}{\partial \mathbf{F}} = 4\mathbf{F}\mathbf{F}^T\mathbf{F} \qquad \frac{\partial^2 II_{\mathbf{C}}}{\partial \mathbf{F} \partial f_{ij}} = 4\left(\frac{\partial \mathbf{F}}{\partial f_{ij}}\mathbf{F}^T\mathbf{F} + \mathbf{F}\frac{\partial \mathbf{F}}{\partial f_{ij}}^T\mathbf{F} + \mathbf{F}\mathbf{F}^T\frac{\partial \mathbf{F}}{\partial f_{ij}}\right).$$

Hey, that's the same hairy term from the Hessian of StVK (Eqn. 4.26)! Up to a constant factor. In total, the StVK stretching energy is really just a combination of the first and second invariants. If we can determine the flattened Hessian for the second invariant, then we have all the pieces we need to obtain the flattened Hessian for the StVK stretching term. It's not pretty, but here it is:

$$\mathbf{H}_{II} = 4\left(\mathbf{I}_{3\times3} \otimes \mathbf{F}\mathbf{F}^T + \mathbf{F}^T\mathbf{F} \otimes \mathbf{I}_{3\times3} + \mathbf{D}\right).$$

The $\otimes$ denotes a Kronecker product, and the $\mathbf{D}$ term works out to:[5]

$$\mathbf{D} = \begin{bmatrix} \mathbf{f}_0\mathbf{f}_0^T & \mathbf{f}_1\mathbf{f}_0^T & \mathbf{f}_2\mathbf{f}_0^T \\ \mathbf{f}_0\mathbf{f}_1^T & \mathbf{f}_1\mathbf{f}_1^T & \mathbf{f}_2\mathbf{f}_1^T \\ \mathbf{f}_0\mathbf{f}_2^T & \mathbf{f}_1\mathbf{f}_2^T & \mathbf{f}_2\mathbf{f}_2^T \end{bmatrix}. \tag{4.27}$$

If you haven't seen a Kronecker product before, or haven't used them in a long time, they are reviewed in Appendix C.5. Like I said, it's not pretty. But, at least we've baked all the ugliness into $\mathbf{H}_{II}$, so we can go straight from this version of the StVK Hessian,

$$\frac{\partial \mathbf{P}}{\partial f_{ij}} = \left(\frac{\partial \mathbf{F}}{\partial f_{ij}}\mathbf{F}^T\mathbf{F} + \mathbf{F}\frac{\partial \mathbf{F}}{\partial f_{ij}}^T\mathbf{F} + \mathbf{F}\mathbf{F}^T\frac{\partial \mathbf{F}}{\partial f_{ij}}\right) - \frac{\partial \mathbf{F}}{\partial f_{ij}}, \tag{4.28}$$

down to this short thing:

$$\text{vec}\left(\frac{\partial \mathbf{P}}{\partial \mathbf{F}}\right) = \frac{1}{4}\mathbf{H}_{II} - \mathbf{I}_{9\times9}. \tag{4.29}$$

At least we got *something*, and as with Neo-Hookean, it will turn out that the pieces we've just obtained can be used over and over elsewhere. Things could be worse.

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

With the As-Rigid-As-Possible (ARAP) energy, things finally go terribly wrong. It's a little surprising that such a friendly-looking energy would cause such trouble:

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

If you recall from §3.6.3, ARAP was voted "Most Spring-Mass-Like" because, just like with spring-mass forces, its PK1 simply took the expression inside the norm and placed it outside of the norm.

[5] This may look like an outer product of vec $\mathbf{F}$, but don't be fooled. It's slightly different. Is there another clean explanation? Not that I've found, so if you find one, send me an email.

We can write down a matrix entry in its 4th-order tensor thusly:

$$\frac{\partial \mathbf{P}}{\partial f_{ij}} = \mu \left(\frac{\partial \mathbf{F}}{\partial f_{ij}} - \frac{\partial \mathbf{R}}{\partial f_{ij}} \right).$$

The full 4th-order tensor doesn't look much different:

$$\frac{\partial \mathbf{P}}{\partial \mathbf{F}} = \mu \left(\frac{\partial \mathbf{F}}{\partial \mathbf{F}} - \frac{\partial \mathbf{R}}{\partial \mathbf{F}} \right).$$

The first term is easy: $\frac{\partial \mathbf{F}}{\partial \mathbf{F}}$ is our Dirichlet buddy yet again. The second term is the trouble-maker. *What the heck is* $\frac{\partial \mathbf{R}}{\partial \mathbf{F}}$*?* The $\mathbf{R}$ comes from the polar decomposition $\mathbf{F} = \mathbf{RS}$, so it arrives as a result of a numerical factorization of $\mathbf{F}$. How are we supposed to take the derivative of a numerical factorization? There are no explicit entries of $\mathbf{F}$ that appear in $\mathbf{R}$, so running back to your old calculus textbook will not help.

We could try applying auto-differentiation to the numerical algorithm used for the polar decomposition. I did this once (Gao et al. (2009)), and it is by far the ugliest and most complicated code that I have ever written. It was also brittle and slow. Don't try it.

This term is so odd-looking that early works either missed or glossed over it (Müller et al. (2002); Irving et al. (2004)). Later work (Chao et al. (2010); McAdams et al. (2011); Barbic (2012)) noticed its existence, and devised a variety of techniques, of varying complexity, for dealing with it. One thing is for sure: if we want to get an expression for the ARAP energy's Hessian, we're going to need something more. We need a way to compute $\frac{\partial \mathbf{R}}{\partial \mathbf{F}}$, the *rotation gradient*.

This will turn out to be a can of worms that opens a trapdoor under our feet, throws us down a dusty chute, through roots and vines, and dumps us out on an entirely new forest path. It'll need its own chapter.

Chapter 5

A Better Way For Isotropic Solids

This chapter is a loose retelling of §4.1 and §4.2 from Smith et al. (2019). If you want to just cut to the chase, the final algorithm is given in §5.5. Everything preceding it will explain where the pieces of this algorithm are coming from, and why they are necessary.

5.1 The Situation So Far

In the previous chapter, we looked at a whole bunch of different deformation energies:

$$\begin{aligned}
\Psi_{\text{Dirichlet}} &= \|\mathbf{F}\|_F^2 \\
\Psi_{\text{BW08}} &= \frac{\mu}{2}(\|\mathbf{F}\|_F^2 - 3) - \mu \log(J) + \frac{\lambda}{2}\left(\log(J)\right)^2 \\
\Psi_{\text{StVK, stretch}} &= \|\mathbf{E}\|_F^2 \\
\Psi_{\text{ARAP}} &= \frac{\mu}{2}\|\mathbf{F} - \mathbf{R}\|_F^2
\end{aligned}$$

After jumping through lots of tensor hoops and doing a bunch of tedious manipulations, getting the force gradients for these energies turned out to be possible.[1] Except for ARAP. What's going on? Why does this friendly-looking energy cause so much trouble?

What we're going to see is that the first three energies, $\Psi_{\text{Dirichlet}}$, Ψ_{BW08} and $\Psi_{\text{StVK, stretch}}$ are all *Cauchy-Green* energies. I mentioned the Cauchy-Green thing in §4.2.3 when we were looking at $\Psi_{\text{StVK, stretch}}$, but now it will take center stage. Just to offer a spoiler, it will turn out that ARAP is *not* part of the Cauchy-Green Club, and its non-membership is the source of all the trouble. What does this even mean?

This chapter is longer than the previous ones, so again, if you want to cut to the chase, the final result is in §5.5. Otherwise, let's get started on our journey.

[1]Just so we don't lose sight of our goal, we need these gradients if we want to use a Baraff and Witkin (1998)-style simulator.

5.2 The Cauchy-Green Invariants (a.k.a The Wrong Way)

5.2.1 The Gradients and Hessians of the Invariants

There are lots of ways to boil the matrix $\mathbf{F}$ down to a scalar, and depending on the method you choose, it can reveal different aspects of what's going on inside $\mathbf{F}$. The Cauchy-Green invariants are one method, and while we will later see that they're not *the one true* method, we should first see what they're good at and what they're bad at. Then we'll see if we can fix the bad while keeping the good.

Here are the three Cauchy-Green invariants:

$$I_{\mathbf{C}} = \|\mathbf{F}\|_F^2 \tag{5.1}$$
$$II_{\mathbf{C}} = \|\mathbf{F}^T\mathbf{F}\|_F^2 \tag{5.2}$$
$$III_{\mathbf{C}} = \det\left(\mathbf{F}^T\mathbf{F}\right). \tag{5.3}$$

There are lots of identities you can apply to these, so these invariants can show up in different places in lots of sneaky disguises. For example, the first invariant has been known to use the following aliases:

$$I_{\mathbf{C}} = \operatorname{tr}\left(\mathbf{F}^T\mathbf{F}\right) = \operatorname{tr}(\mathbf{C}) = \|\mathbf{F}\|_F^2 = \mathbf{F} : \mathbf{F} = \sum_i \sum_j \mathbf{F}_{ij}^2. \tag{5.4}$$

Meanwhile, the third invariant can show up in these forms:

$$III_{\mathbf{C}} = \det\left(\mathbf{F}^T\mathbf{F}\right) = \det(\mathbf{C}) = \det(\mathbf{F})^2 = J^2. \tag{5.5}$$

Above, $\mathbf{C} = \mathbf{F}^T\mathbf{F}$, and that's where the $\mathbf{C}$ subscript for the invariants comes from.[2]

Why are these invariants useful? Why are we even talking about them? We got a small hint of this in the previous chapter, when figuring out the force gradient of $\Psi_{\text{StVK, stretch}}$ got a lot easier once we introduced $II_{\mathbf{C}}$. Also, I mentioned in §4.2.2.3 that there is a *limited rogue's gallery* of $\mathbf{g}_*$ and $\mathbf{H}_*$ terms that can ever appear.

The invariants bring these two advantages together. If we can get the gradients and Hessians of all three invariants, then deriving the an energy's force gradient becomes *much easier*. This is because the gradients and Hessians of the invariants *are the complete rogue's gallery*. Once we have these, everything becomes a mix-and-match exercise involving the same pieces over and over.[3]

[2]If you've taken a theory-slanted linear algebra class, or maybe a visualization class, you may recognize these as the *tensor invariants* that arise from the characteristic polynomial of $\mathbf{C}$. If you don't know what those words mean, that's okay too.

[3]Except when we try it on the ARAP energy! But, let's first see how high we can fly with this Cauchy-Green thing before ARAP causes us to face-plant.

We saw the gradient and Hessian of $I_{\mathbf{C}}$ before in §4.2.3:

$$\frac{\partial I_{\mathbf{C}}}{\partial \mathbf{F}} = 2\mathbf{F} \qquad \frac{\partial^2 I_{\mathbf{C}}}{\partial \mathbf{F}^2} = 2\frac{\partial \mathbf{F}}{\partial \mathbf{F}}.$$

We flatten these to get the gradient and Hessian versions:

$$\begin{aligned} \mathbf{g}_I &= \operatorname{vec}\left(\frac{\partial I_{\mathbf{C}}}{\partial \mathbf{F}}\right) = 2\operatorname{vec}(\mathbf{F}) \\ \mathbf{H}_I &= \operatorname{vec}\left(\frac{\partial^2 I_{\mathbf{C}}}{\partial \mathbf{F}^2}\right) = 2\mathbf{I}_{9\times9}. \end{aligned}$$

We can do the same thing for $II_{\mathbf{C}}$,

$$\begin{aligned} \mathbf{g}_{II} &= 4\operatorname{vec}(\mathbf{FE}) \\ \mathbf{H}_{II} &= 4\left(\mathbf{I}_{3\times3} \otimes \mathbf{FF}^T + \mathbf{F}^T\mathbf{F} \otimes \mathbf{I}_{3\times3} + \mathbf{D}\right), \end{aligned}$$

where $\mathbf{D}$ was defined in Eqn. 4.27. Finally, here are the gradient and Hessian of $III_{\mathbf{C}}$,

$$\begin{aligned} \mathbf{g}_{III} &= 2\det J \cdot \mathbf{g}_J \\ \mathbf{H}_{III} &= 2\mathbf{g}_J\mathbf{g}_J^T + 2\det J \cdot \mathbf{H}_J \end{aligned}$$

where we saw $\mathbf{g}_J$ and $\mathbf{H}_J$ back in Eqns. 5.6 and 5.7:

$$\mathbf{g}_J = \operatorname{vec}\left(\frac{\partial J}{\partial \mathbf{F}}\right) = \operatorname{vec}\left(\left[\begin{array}{c|c|c} \mathbf{f}_1 \times \mathbf{f}_2 & \mathbf{f}_2 \times \mathbf{f}_0 & \mathbf{f}_0 \times \mathbf{f}_1 \end{array}\right]\right) \tag{5.6}$$

$$\mathbf{H}_J = \begin{bmatrix} \mathbf{0} & -\hat{\mathbf{f}}_2 & \hat{\mathbf{f}}_1 \\ \hat{\mathbf{f}}_2 & \mathbf{0} & -\hat{\mathbf{f}}_0 \\ -\hat{\mathbf{f}}_1 & \hat{\mathbf{f}}_0 & \mathbf{0} \end{bmatrix}. \tag{5.7}$$

5.2.2 Getting Any Hessian, the Cauchy-Green Way

With these in hand, we can now use the chain rule[4] to obtain a generic expression for the Hessian of some arbitrary energy, Ψ:

$$\begin{aligned} \operatorname{vec}\left(\frac{\partial^2 \Psi}{\partial \mathbf{F}^2}\right) = & \frac{\partial^2 \Psi}{\partial I_{\mathbf{C}}^2}\mathbf{g}_I\mathbf{g}_I^T + \frac{\partial \Psi}{\partial I_{\mathbf{C}}}\mathbf{H}_I + \\ & \frac{\partial^2 \Psi}{\partial II_{\mathbf{C}}^2}\mathbf{g}_{II}\mathbf{g}_{II}^T + \frac{\partial \Psi}{\partial II_{\mathbf{C}}}\mathbf{H}_{II} + \\ & \frac{\partial^2 \Psi}{\partial III_{\mathbf{C}}^2}\mathbf{g}_{III}\mathbf{g}_{III}^T + \frac{\partial \Psi}{\partial III_{\mathbf{C}}}\mathbf{H}_{III}. \end{aligned}$$

[4]Technically Faá di Bruno's formula, since it's a higher order derivative.

A compact, but slightly more obtuse, way of writing this is:

$$\operatorname{vec}\left(\frac{\partial^2\Psi}{\partial\mathbf{F}^2}\right) = \sum_{x\in\{I_\mathbf{C}, II_\mathbf{C}, III_\mathbf{C}\}} \frac{\partial^2\Psi}{\partial x^2}\mathbf{g}_x\mathbf{g}_x^T + \frac{\partial\Psi}{\partial x}\mathbf{H}_x. \tag{5.8}$$

We just got done listing all the $\mathbf{g}_*$ and $\mathbf{H}_*$ terms in §5.2.1, so there's no work left to do there. Like I said, it's the complete rogue's gallery. Instead, the only new work to be done is deriving all the $\frac{\partial\Psi}{\partial x}$ and $\frac{\partial^2\Psi}{\partial x^2}$ terms, such as $\frac{\partial\Psi}{\partial I_\mathbf{C}}$ and $\frac{\partial^2\Psi}{\partial I_\mathbf{C}^2}$.

Those are all scalar derivatives! You can grind through those with just a high-school level grasp of calculus.[5] If you're feeling really slovenly, you can even just pawn it off on Mathematica or Wolfram Alpha.

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

1. Re-write your energy Ψ in terms of $I_\mathbf{C}$, $II_\mathbf{C}$, and $III_\mathbf{C}$.
2. Derive the scalar derivatives, $\frac{\partial\Psi}{\partial I_\mathbf{C}}$, $\frac{\partial^2\Psi}{\partial I_\mathbf{C}^2}$, $\frac{\partial\Psi}{\partial II_\mathbf{C}}$, $\frac{\partial^2\Psi}{\partial II_\mathbf{C}^2}$, $\frac{\partial\Psi}{\partial III_\mathbf{C}}$ and $\frac{\partial^2\Psi}{\partial III_\mathbf{C}^2}$. Yes, there are six of them, but c'mon, it's like six homework problems! And it's not cheating to use the computer!
3. Plug the results into Eqn. 5.8. You're all done.

5.2.3 St. Venant Kirchhoff Stretching, the Cauchy-Green Way

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

$$\Psi_{\text{StVK, stretch}} = \|\mathbf{E}\|_F^2. \tag{5.9}$$

Step 1: Rewrite using invariants. Expanding out the $\mathbf{E} = \frac{1}{2}\left(\mathbf{F}^T\mathbf{F} - \mathbf{I}\right)$ and re-writing in terms of invariants, we get

$$\begin{aligned}
\Psi_{\text{StVK, stretch}} &= \|\mathbf{E}\|_F^2 && (5.10)\\
&= \frac{1}{4}\|\mathbf{F}^T\mathbf{F} - \mathbf{I}\|_F^2 && \text{(Substitute } \mathbf{E}\text{)}\\
&= \frac{1}{4}\left(\|\mathbf{F}^T\mathbf{F}\|_F^2 - 2\operatorname{tr}\left(\mathbf{F}^T\mathbf{F}\right) + \|\mathbf{I}\|_F^2\right) && \text{(Apply B.16)}\\
&= \frac{1}{4}II_\mathbf{C} - \frac{1}{2}I_\mathbf{C} + 3 && \text{(Substitute invariants)}
\end{aligned}$$

We also used $\|\mathbf{I}\|_F^2 = 3$ on the last line, which holds in 3D. In 2D, $\|\mathbf{I}\|_F^2 = 2$, but this usually isn't that important, because it works out to a constant either way, and the constant gets burned off when we take the derivative.

[5] If you *don't* have a high-school level grasp of calculus, but have gotten this far, then some Srinivasa Ramanujan-type situation is going on, and probably don't need to read any of this.

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

$$\frac{\partial \Psi_{\text{StVK, stretch}}}{\partial I_{\mathbf{C}}} = -\frac{1}{2} \qquad \frac{\partial^2 \Psi_{\text{StVK, stretch}}}{\partial I_{\mathbf{C}}^2} = 0 \tag{5.11}$$

$$\frac{\partial \Psi_{\text{StVK, stretch}}}{\partial II_{\mathbf{C}}} = \frac{1}{4} \qquad \frac{\partial^2 \Psi_{\text{StVK, stretch}}}{\partial II_{\mathbf{C}}^2} = 0 \tag{5.12}$$

$$\frac{\partial \Psi_{\text{StVK, stretch}}}{\partial III_{\mathbf{C}}} = 0 \qquad \frac{\partial^2 \Psi_{\text{StVK, stretch}}}{\partial III_{\mathbf{C}}^2} = 0 \tag{5.13}$$

Wait a minute ... most of these just work out to zero. Did the professor make the homework problem super-easy by accident?! No, the derivatives are usually pretty easy. Like I said, this part turns into a calculus homework problem; one of the easy ones near the beginning, not a hard one from the end.

Step 3: Plug into Eqn. 5.8. Now let's plug into that sum from before:

$$\begin{aligned}
\operatorname{vec}\left(\frac{\partial^2 \Psi_{\text{StVK, stretch}}}{\partial \mathbf{F}^2}\right) &= 0 \cdot \mathbf{g}_I \mathbf{g}_I^T - \frac{1}{2}\mathbf{H}_I + 0 \cdot \mathbf{g}_{II}\mathbf{g}_{II}^T + \frac{1}{4} \cdot \mathbf{H}_{II} + 0 \cdot \mathbf{g}_{III}\mathbf{g}_{III}^T + 0 \cdot \mathbf{H}_{III} \\
&= \frac{1}{4}\mathbf{H}_{II} - \frac{1}{2}\mathbf{H}_I \\
&= \frac{1}{4}\mathbf{H}_{II} - \mathbf{I}_{9\times 9} \qquad \text{(Plug in the definition of } \mathbf{H}_I\text{)}
\end{aligned}$$

It's exactly the same as Eqn. 4.29, but much easier to derive! The system works.

5.2.4 Neo-Hookean, the Cauchy-Green Way

Next let's try it out on Bonet and Wood (2008)-style Neo-Hookean:

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

Step 1: Rewrite using invariants. We can write this as:

$$\Psi_{\text{BW08}} = \frac{\mu}{2}(I_{\mathbf{C}} - 3) - \mu \log(\sqrt{III_{\mathbf{C}}}) + \frac{\lambda}{2}\left(\log(\sqrt{III_{\mathbf{C}}})\right)^2$$

At this point, I'm going to let you in on a secret: using $\sqrt{III_{\mathbf{C}}}$ is a *bad idea*. Remember that $III_{\mathbf{C}} = (\det \mathbf{F})^2$, and $J = \det \mathbf{F}$. When we compute $\sqrt{III_{\mathbf{C}}}$, all we're doing is taking the sign of J and throwing it away.

The sign of J is really useful! If it's negative, it means that our element has been non-physically poked inside out, a.k.a. *inverted*. If we're in this situation, we want to know about it so we can deal with it. We'll talk about this more later on in §6.1.2. For the time being, let's keep the Js in the energy intact:

$$\Psi_{\text{BW08}} = \frac{\mu}{2}(I_{\mathbf{C}} - 3) - \mu \log(J) + \frac{\lambda}{2}\left(\log(J)\right)^2$$

Step 2: Take the invariant derivatives. Let's take the derivatives, but instead of $\frac{\partial \Psi}{\partial III_{\mathbf{C}}}$ and $\frac{\partial^2 \Psi}{\partial III_{\mathbf{C}}^2}$, we'll use $\frac{\partial \Psi}{\partial J}$ and $\frac{\partial^2 \Psi}{\partial J^2}$[6]:

$$\frac{\partial \Psi_{\text{BW08}}}{\partial I_{\mathbf{C}}} = \frac{\mu}{2} \qquad \frac{\partial^2 \Psi_{\text{BW08}}}{\partial I_{\mathbf{C}}^2} = 0 \tag{5.14}$$

$$\frac{\partial \Psi_{\text{BW08}}}{\partial II_{\mathbf{C}}} = 0 \qquad \frac{\partial^2 \Psi_{\text{BW08}}}{\partial II_{\mathbf{C}}^2} = 0 \tag{5.15}$$

$$\frac{\partial \Psi_{\text{BW08}}}{\partial J} = \frac{\lambda \log J - \mu}{J} \qquad \frac{\partial^2 \Psi_{\text{BW08}}}{\partial J^2} = \frac{\lambda(1 - \log J) + \mu}{J^2}. \tag{5.16}$$

The scalar derivatives are a little more involved than last time, but again, you can just punch it into Mathematica or Wolfram Alpha and have it do all the heavy lifting for you. It's not cheating. It's what I just did.

Step 3: Plug into the chain rule. For the final expression, we're going to again replace all the $III_{\mathbf{C}}$ terms with J terms, as follows:

$$\begin{aligned}\operatorname{vec}\left(\frac{\partial^2 \Psi}{\partial \mathbf{F}^2}\right) =& \frac{\partial^2 \Psi}{\partial I_{\mathbf{C}}^2} \mathbf{g}_I \mathbf{g}_I^T + \frac{\partial \Psi}{\partial I_{\mathbf{C}}} \mathbf{H}_I + \\ & \frac{\partial^2 \Psi}{\partial II_{\mathbf{C}}^2} \mathbf{g}_{II} \mathbf{g}_{II}^T + \frac{\partial \Psi}{\partial II_{\mathbf{C}}} \mathbf{H}_{II} + \\ & \frac{\partial^2 \Psi}{\partial J^2} \mathbf{g}_J \mathbf{g}_J^T + \frac{\partial \Psi}{\partial J} \mathbf{H}_J.\end{aligned}$$

Fortunately, we have expressions for $\mathbf{g}_J$ and $\mathbf{H}_J$ in Eqns. 5.6 and 5.7[7], so we get:

$$\begin{aligned}\operatorname{vec}\left(\frac{\partial^2 \Psi_{\text{BW08}}}{\partial \mathbf{F}^2}\right) &= \frac{\mu}{2} \mathbf{H}_I + \frac{\lambda(1 - \log J) + \mu}{J^2} \mathbf{g}_J \mathbf{g}_J^T + \frac{\lambda \log J - \mu}{J} \mathbf{H}_J. \qquad (5.17) \\ &= \mu \mathbf{I}_{9\times 9} + \frac{\lambda(1 - \log J) + \mu}{J^2} \mathbf{g}_J \mathbf{g}_J^T + \frac{\lambda \log J - \mu}{J} \mathbf{H}_J. \\ &\qquad \text{(Plug in the definition of } \mathbf{H}_I\text{)}\end{aligned}$$

Again, it matches Eqn. 4.25 exactly! But, it was much easier to derive.

5.2.5 As-Rigid-As-Possible: Things Go Terribly Wrong (Again)

It sure seems like we now have an easy and generic way of deriving any Hessian. Let's try it out on ARAP:

$$\Psi_{\text{ARAP}} = \frac{\mu}{2} \|\mathbf{F} - \mathbf{R}\|_F^2 \tag{5.18}$$

[6] This is still kosher. You can verify it via the chain rule if it bothers you.

[7] Really, the $III_{\mathbf{C}}$ versions just built on top of them. The J versions are clearly the more atomic and fundamental expressions, which we'll see in more detail in §5.3.3.2.

Step 1: Rewrite using invariants. Things will go off the rails quickly:

$$\begin{aligned}\Psi_{\text{ARAP}} &= \frac{\mu}{2}\|\mathbf{F}-\mathbf{R}\|_F^2 && (5.19)\\ &= \|\mathbf{F}\|_F^2 - 2\operatorname{tr}\left(\mathbf{F}^T\mathbf{R}\right) + \|\mathbf{R}\|_F^2 && \text{(Apply B.16)}\\ &= I_{\mathbf{C}} - 2\operatorname{tr}\left(\mathbf{F}^T\mathbf{R}\right) + 3. && \text{(Substitute Invariants)}\end{aligned}$$

What the heck is $\operatorname{tr}\left(\mathbf{F}^T\mathbf{R}\right)$*?* It doesn't correspond to any of the Cauchy-Green invariants! Like we saw in 4.2.4, the $\mathbf{R}$ term comes from the polar decomposition $\mathbf{F} = \mathbf{RS}$, so it arises from the numerical factorization of $\mathbf{F}$.

But that messes everything up! The $\mathbf{R}$ makes it so that we can't fold *all* appearances of $\mathbf{F}$ into Cauchy-Green invariants, and that breaks our entire Hessian derivation strategy. Nice going, ARAP.

What went wrong? The ARAP energy has now ruined things twice. In §4.2.4, we tried to do things the hard way and slash our way through a thicket of tensors and derivatives, but came up against an impenetrable rotation gradient term, $\frac{\partial \mathbf{R}}{\partial \mathbf{F}}$. At that point, we were stuck. How do you take the symbolic derivative of something ($\mathbf{R}$) that is purely numerical?

Then, we tried the Cauchy-Green invariant approach, but immediately ran into a $\operatorname{tr}\left(\mathbf{F}^T\mathbf{R}\right)$ term that doesn't fit into our invariants, and again we got stuck. Why do things keep going wrong?

5.3 A Better Set of Invariants?

Our most recent problem, the $\operatorname{tr}\left(\mathbf{F}^T\mathbf{R}\right)$ term, provides a good clue. To see why, let's look again at all the forms that the $I_{\mathbf{C}}$ invariant can take:

$$I_{\mathbf{C}} = \operatorname{tr}\left(\mathbf{F}^T\mathbf{F}\right) = \operatorname{tr}(\mathbf{C}) = \|\mathbf{F}\|_F^2 = \mathbf{F}:\mathbf{F} = \sum_i\sum_j \mathbf{F}_{ij}^2.$$

I've highlighted the most relevant one in red. This sure looks similar to that $\operatorname{tr}\left(\mathbf{F}^T\mathbf{R}\right)$ term that was giving us trouble, doesn't it? Keeping in mind again that $\mathbf{R}$ arose from the polar decomposition $\mathbf{F} = \mathbf{RS}$, we can go ahead and write this as,

$$\operatorname{tr}\left(\mathbf{F}^T\mathbf{R}\right) = \operatorname{tr}\left(\mathbf{S}^T\right) = \operatorname{tr}(\mathbf{S}), \qquad (5.20)$$

where we can drop the transpose at the end because $\mathbf{S}$ is a symmetric matrix.

Can we just add this to our gallery of invariants? Is that even a sensible question to ask? I haven't actually mentioned yet why these are called *invariants*. Invariants with respect to what? Understanding these quantities intuitively is about to become important, so let's look at two different ways to think about them.

5.3.1 Invariants as Rotation Removers

The Cauchy-Green invariants are invariant to *rotation*, that irritating quantity from way back in §2.3.4 that we poured lots of effort into removing from $\mathbf{F}$. Only then could we get a clear picture of how much stretching and squashing was going on. No matter how you rotate some deformed triangle or tetrahedron in space, if you compute $I_{\mathbf{C}}$, $II_{\mathbf{C}}$, or $III_{\mathbf{C}}$, they should return the exact same number. In §2.3.4, we looked at two different approaches to removing the rotation.

5.3.1.1 The St. Venant-Kirchhoff Way

The first was in §2.3.4.3, where we used the StVK-style way,

$$\Psi_{\text{StVK, stretch}} = \frac{1}{2}\|\mathbf{F}^T\mathbf{F} - \mathbf{I}\|_F^2.$$

which uses $\mathbf{F}^T\mathbf{F}$ to remove rotation. If we write this in terms of the polar decomposition, we can explicitly see the moment the rotation gets burned off:

$$\begin{aligned}
\mathbf{F}^T\mathbf{F} &= (\mathbf{RS})^T\mathbf{RS} = \mathbf{S}^T\mathbf{R}^T\mathbf{RS} \\
&= \mathbf{S}^T\mathbf{S} && \text{(Since } \mathbf{R}^T\mathbf{R} = \mathbf{I}\text{)} \\
&= \mathbf{S}^2 && \text{(Since } \mathbf{S} \text{ is symmetric)}
\end{aligned}$$

So really, we *could* add yet another alias to the list of disguises that $I_{\mathbf{C}}$ can take on,

$$I_{\mathbf{C}} = \operatorname{tr}\left(\mathbf{F}^T\mathbf{F}\right) = \operatorname{tr}(\mathbf{S}^2),$$

and we *could* rewrite the StVK energy as:

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

5.3.1.2 The As-Rigid-As-Possible Way

We also looked at a *second* style of rotation removal in §2.3.4.4, the ARAP-style way:

$$\Psi_{\text{ARAP}} = \|\mathbf{F} - \mathbf{R}\|_F^2.$$

We can use the fact that the Frobenius norm does not change under rotation to burn off $\mathbf{R}$ in this case as well:

$$\begin{aligned}
\Psi_{\text{ARAP}} &= \|\mathbf{F} - \mathbf{R}\|_F^2 \\
&= \|\mathbf{R}^T(\mathbf{F} - \mathbf{R})\|_F^2 && \text{(Using rotation invariance of } \|\cdot\|_F\text{)} \\
&= \|\mathbf{R}^T(\mathbf{RS} - \mathbf{R})\|_F^2 && \text{(Inserting } \mathbf{F} = \mathbf{RS}\text{)} \\
&= \|\mathbf{S} - \mathbf{I}\|_F^2 && \text{(Using } \mathbf{R}^T\mathbf{R} = \mathbf{I}\text{)}
\end{aligned}$$

Looks pretty close to the StVK energy in this form, doesn't it? One squares $\mathbf{S}$ while the other one doesn't?

5.3.1.3 *Comparing the Two Ways*

We can do one last manipulation to get a better understanding of what is going on. The ultimate measure of how much stretching and squashing is *really* going on in $\mathbf{F}$ can be obtained using its SVD, which is $\mathbf{F} = \mathbf{U}\Sigma\mathbf{V}^T$. The diagonal matrix of singular values, Σ, shows *exactly* how much stretching is going on along each of the 3D directions. Can we write each energy in terms of that?

Using the rotation invariance of $\|\cdot\|_F^2$ again, and the definition $\mathbf{S} = \mathbf{V}^T\Sigma\mathbf{V}$, we get:

$$\begin{aligned}
\Psi_{\text{StVK, stretch}} &= \frac{1}{2}\|\mathbf{S}^2 - \mathbf{I}\|_F^2 \\
&= \frac{1}{2}\|\mathbf{V}\left(\mathbf{S}^2 - \mathbf{I}\right)\mathbf{V}^T\|_F^2 && \text{(Using rotation invariance of } \|\cdot\|_F\text{)} \\
&= \frac{1}{2}\|\mathbf{V}\left(\mathbf{V}^T\Sigma\mathbf{V}\mathbf{V}^T\Sigma\mathbf{V} - \mathbf{I}\right)\mathbf{V}^T\|_F^2 && \text{(Using } \mathbf{S} = \mathbf{V}^T\Sigma\mathbf{V}\text{)} \\
&= \frac{1}{2}\|\Sigma\Sigma - \mathbf{I}\|_F^2 && \text{(Using } \mathbf{V}^T\mathbf{V} = \mathbf{V}\mathbf{V}^T = \mathbf{I}\text{, a lot)} \\
&= \frac{1}{2}\|\Sigma^2 - \mathbf{I}\|_F^2 && \text{(Since } \Sigma \text{ is a diagonal matrix)} \\
\Psi_{\text{ARAP}} &= \|\mathbf{S} - \mathbf{I}\|_F^2 \\
&= \|\mathbf{V}\left(\mathbf{S} - \mathbf{I}\right)\mathbf{V}^T\|_F^2 \\
&= \|\mathbf{V}\left(\mathbf{V}^T\Sigma\mathbf{V} - \mathbf{I}\right)\mathbf{V}^T\|_F^2 \\
&= \|\Sigma - \mathbf{I}\|_F^2
\end{aligned}$$

In summary, we now have:

$$\Psi_{\text{StVK, stretch}} = \frac{1}{2}\|\Sigma^2 - \mathbf{I}\|_F^2 \qquad\qquad \Psi_{\text{ARAP}} = \|\Sigma - \mathbf{I}\|_F^2$$

Ignoring the $1/2$ in front of $\Psi_{\text{StVK, stretch}}$[8], the only real difference is that StVK squares all the singular values, while ARAP doesn't. What does that mean?

We can use the same $\mathbf{S} = \mathbf{V}^T\Sigma\mathbf{V}$ identity to obtain a new alias for $I_{\mathbf{C}}$:

$$I_{\mathbf{C}} = \operatorname{tr}\left(\mathbf{F}^T\mathbf{F}\right) = \operatorname{tr}(\mathbf{S}^2) = \operatorname{tr}(\Sigma^2).$$

Curiously, that irritating $\operatorname{tr}\left(\mathbf{F}^T\mathbf{R}\right)$ term from ARAP works out to something similar:

$$\operatorname{tr}\left(\mathbf{R}^T\mathbf{F}\right) = \operatorname{tr}(\mathbf{S}) = \operatorname{tr}(\Sigma).$$

When you boil things down to just the singular values Σ, it sure looks like $\operatorname{tr}\left(\mathbf{R}^T\mathbf{F}\right)$ measures something similar to $I_{\mathbf{C}}$. That squaring of Σ though, how important is it? Again, what does it even mean? To answer that, we'll need to look at the invariants from a different perspective.

[8] We're looking at the qualitative behavior of these energies as $\mathbf{F}$ changes, so a constant in front doesn't matter.

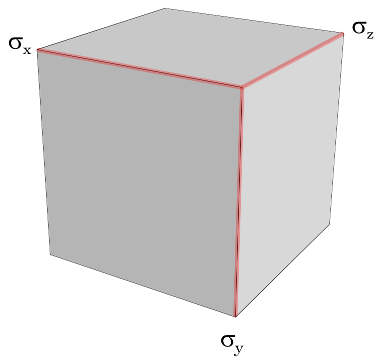


Figure 5.1.: The matrix **F** describes the rotation and scaling of an infinitesimal cube of material. The singular values of **F**, the $\sigma_{\{x,y,z\}}$ labels above, describe the lengths of each side of that cube.

5.3.2 Invariants as Geometric Measurements

In the last section, I said that to see what's *really* going on in **F**, you should look at its singular values, i.e. the Σ in $\mathbf{F} = \mathbf{U}\Sigma\mathbf{V}^T$. We can interpret the invariants in terms of these singular values.

Looking at the $\text{tr}(\Sigma^2)$ version of $I_\mathbf{C}$, this works out to:

$$I_\mathbf{C} = \sigma_x^2 + \sigma_y^2 + \sigma_z^2. \tag{5.21}$$

The $\sigma_{\{x,y,z\}}$ terms have a straightforward geometric interpretation. The **F** matrix characterizes rotation and scaling at a *quadrature point*, i.e. some infinitesimal volumetric piece of the solid. As described previously, the singular values are amount of stretching and squashing *along each axis*, as shown in Fig. 5.1.

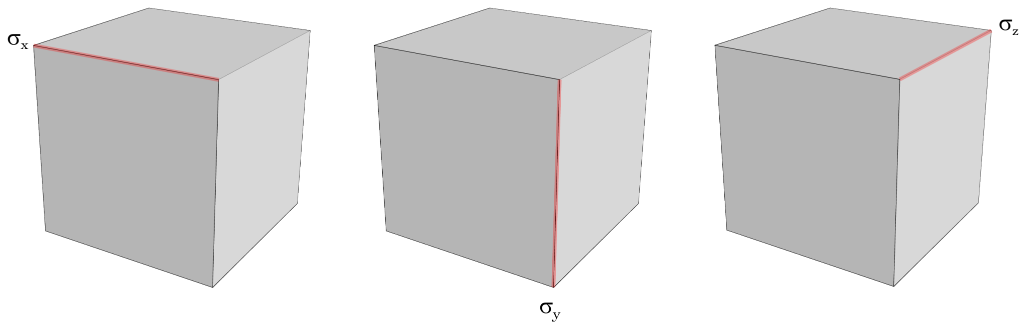


Figure 5.2.: $I_\mathbf{C}$ sums up the squared lengths of each side of the cube: $I_\mathbf{C} = \sigma_x^2 + \sigma_y^2 + \sigma_z^2$. I.e. it measures the overall edge lengths of the entire cube.

The $I_\mathbf{C}$ invariant is the sum of the *squared lengths of each side of the cube* (Fig. 5.2). We can work out similar terms for $II_\mathbf{C}$ and $III_\mathbf{C}$, and show that they have straightforward geometric meanings with respect to this cube.

The $II_{\mathbf{C}}$ invariant can be rewritten thusly:

$$\begin{aligned}
II_{\mathbf{C}} &= \|\mathbf{F}^T\mathbf{F}\|_F^2 \\
&= \operatorname{tr}\left(\left(\mathbf{F}^T\mathbf{F}\right)^T\mathbf{F}^T\mathbf{F}\right) \\
&= \operatorname{tr}\left(\mathbf{S}^2\mathbf{S}^2\right) && \text{(Using } \mathbf{F}^T\mathbf{F} = \mathbf{S}^2\text{, from §5.3.1.1)} \\
&= \operatorname{tr}\left(\mathbf{SSSS}\right) && \text{(Expanding the } \mathbf{S}^2\text{)} \\
&= \operatorname{tr}\left(\mathbf{V}^T\Sigma\mathbf{V}\mathbf{V}^T\Sigma\mathbf{V}\mathbf{V}^T\Sigma\mathbf{V}\mathbf{V}^T\Sigma\mathbf{V}\right) && \text{(Using } \mathbf{S} = \mathbf{V}^T\Sigma\mathbf{V}\text{)} \\
&= \operatorname{tr}\left(\mathbf{V}^T\Sigma\Sigma\Sigma\Sigma\mathbf{V}\right) && \text{(Using } \mathbf{V}\mathbf{V}^T = \mathbf{I}\text{)} \\
&= \operatorname{tr}\left(\mathbf{V}^T\Sigma^4\mathbf{V}\right) && \text{(Consolidating the } \Sigma\text{s)} \\
&= \operatorname{tr}\left(\mathbf{V}\mathbf{V}^T\Sigma^4\right) && \text{(Using the cyclic permutation property)} \\
&= \operatorname{tr}\left(\Sigma^4\right) && \text{(Using } \mathbf{V}\mathbf{V}^T = \mathbf{I}\text{)} \\
&= \sigma_x^4 + \sigma_y^4 + \sigma_z^4
\end{aligned}$$

The cyclic permutation property of traces is: $\operatorname{tr}(\mathbf{ABC}) = \operatorname{tr}(\mathbf{BCA}) = \operatorname{tr}(\mathbf{CAB})$.

Thus, the $II_{\mathbf{C}}$ invariant is the sum of the *lengths of each side of the cube, raised to the fourth power*. This seems ... not that different from $I_{\mathbf{C}}$. In fact, it seems to encapsulate the same basic information. $II_{\mathbf{C}}$ only becomes useful when it is folded into an alternate second invariant:

$$II_{\mathbf{C}}^* = \frac{1}{2}(I_{\mathbf{C}}^2 - II_{\mathbf{C}}). \tag{5.22}$$

Chugging through, we get:

$$\begin{aligned}
II_{\mathbf{C}}^* &= \frac{1}{2}\left(I_{\mathbf{C}}^2 - II_{\mathbf{C}}\right) \\
&= \frac{1}{2}\left((\sigma_x^2 + \sigma_y^2 + \sigma_z^2)^2 - II_{\mathbf{C}}\right) && \text{(Definition of } I_{\mathbf{C}}\text{)} \\
&= \frac{1}{2}\left(\sigma_x^4 + \sigma_y^4 + \sigma_z^4 + 2\left(\sigma_x\sigma_y\right)^2 + 2\left(\sigma_x\sigma_z\right)^2 + 2\left(\sigma_y\sigma_z\right)^2 - II_C\right) \\
& && \text{(Expanding the square)} \\
&= \frac{1}{2}\left(\sigma_x^4 + \sigma_y^4 + \sigma_z^4 + 2\left(\sigma_x\sigma_y\right)^2 + 2\left(\sigma_x\sigma_z\right)^2 + 2\left(\sigma_y\sigma_z\right)^2 - \left(\sigma_x^4 + \sigma_y^4 + \sigma_z^4\right)\right) \\
& && \text{(Definition of } II_{\mathbf{C}}\text{)} \\
&= \frac{1}{2}\left(2\left(\sigma_x\sigma_y\right)^2 + 2\left(\sigma_x\sigma_z\right)^2 + 2\left(\sigma_y\sigma_z\right)^2\right) && \text{(Fourth order terms cancel)} \\
&= \left(\sigma_x\sigma_y\right)^2 + \left(\sigma_x\sigma_z\right)^2 + \left(\sigma_y\sigma_z\right)^2
\end{aligned}$$

This expression is more interesting. Whereas $I_{\mathbf{C}}$ was the summed squared *lengths* of the cube, $II_{\mathbf{C}}^*$ is the sum of the squared *areas* of the cube faces (Fig.5.3).

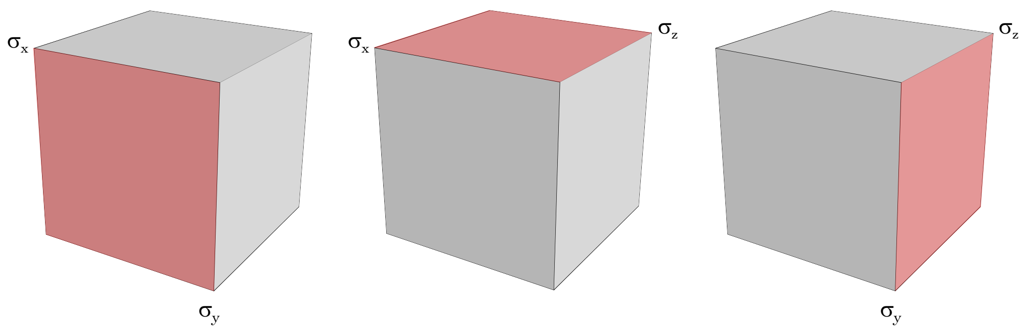


Figure 5.3.: $II^*_{\mathbf{C}}$ sums up the squared *areas* of each side of the cube: $II^*_{\mathbf{C}} = \left(\sigma_x\sigma_y\right)^2 + \left(\sigma_x\sigma_z\right)^2 + \left(\sigma_y\sigma_z\right)^2$. I.e. it measures the overall face areas of the entire cube.

Finally, let's look at $III_{\mathbf{C}}$. The determinant of a matrix is the product of its singular values, so this one is relatively easy:

$$\begin{aligned}
III_{\mathbf{C}} &= \det(\mathbf{F}^T\mathbf{F}) && (5.23)\\
&= \det\left(\mathbf{V}\Sigma\mathbf{U}^T\mathbf{U}\Sigma\mathbf{V}^T\right) && \text{(Using } \mathbf{F} = \mathbf{U}\Sigma\mathbf{V}^T\text{)}\\
&= \det\left(\mathbf{V}\Sigma\Sigma\mathbf{V}^T\right) && \text{(Using } \mathbf{U}^T\mathbf{U} = \mathbf{I}\text{)}\\
&= (\sigma_x\sigma_y\sigma_z)^2. && \text{(}\mathbf{V}\text{ is a rotation, so the singular values are } \Sigma^2\text{)}
\end{aligned}$$

Thus, $III_{\mathbf{C}}$ measures the square of the cube's *volume* (Fig. 5.4).

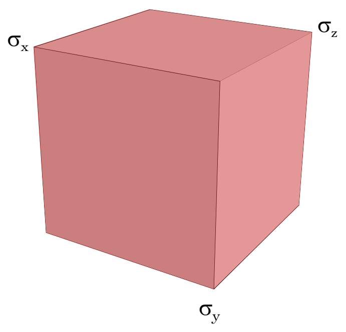


Figure 5.4.: $III^*_{\mathbf{C}}$ is the squared *volume* of the cube: $II^*_{\mathbf{C}} = (\sigma_x\sigma_y\sigma_z)^2$.

Taken together, the three invariants measure some intuitive and relevant geometric quantities:

$$\begin{aligned}
I_{\mathbf{C}} &= \sigma_x^2 + \sigma_y^2 + \sigma_z^2 && \rightarrow && \text{squared } \textit{edge lengths}\\
II^*_{\mathbf{C}} &= \left(\sigma_x\sigma_y\right)^2 + \left(\sigma_x\sigma_z\right)^2 + \left(\sigma_y\sigma_z\right)^2 && \rightarrow && \text{squared } \textit{face areas}\\
III_{\mathbf{C}} &= (\sigma_x\sigma_y\sigma_z)^2 && \rightarrow && \text{squared } \textit{volume}.
\end{aligned}$$

This looks like a *fairly complete* way to describe how some small cube of material can deform, so it makes sense that people have been using the Cauchy-Green invariants to describe deformation for the last 80 years (since Mooney (1940), at least).

So then what is the problem with ARAP? These three invariants look like a completely reasonable way to describe deformations, so why can't the ARAP energy be written down in terms of them?

5.3.3 A New Set of Invariants

5.3.3.1 *A New First Invariant*

Back in §5.3.1.3, we saw this alias for $I_{\mathbf{C}}$,

$$I_{\mathbf{C}} = \operatorname{tr}(\Sigma^2),$$

and this one for the irritatingly non-cooperative term in ARAP:

$$\operatorname{tr}\left(\mathbf{R}^T\mathbf{F}\right) = \operatorname{tr}(\mathbf{S}) = \operatorname{tr}(\Sigma).$$

Then in §5.3.2 we got a singular value version of $I_{\mathbf{C}}$,

$$I_{\mathbf{C}} = \sigma_x^2 + \sigma_y^2 + \sigma_z^2,$$

while the non-cooperative term trivially takes this form:

$$\operatorname{tr}(\mathbf{S}) = \sigma_x + \sigma_y + \sigma_z.$$

The non-cooperative term is an *unsquared* version of $I_{\mathbf{C}}$. Here, it finally becomes clear why ARAP can't be written in terms of the Cauchy-Green invariants. *You can't use a sum of squared values to express a sum of unsquared values.*

This *strongly* suggests that $\operatorname{tr}(\mathbf{S})$ should be its own invariant, because it describes something in $\mathbf{F}$ that can't be described with the Cauchy-Green invariants. Let's call it:

$$\boxed{I_1 = \operatorname{tr}(\mathbf{S}).} \tag{5.24}$$

5.3.3.2 *A New Second and Third Invariant*

We have essentially just performed the following substitution:

$$I_{\mathbf{C}} = \operatorname{tr}\left(\mathbf{F}^T\mathbf{F}\right) \qquad \mathbf{F}^T\mathbf{F} \to \mathbf{S} \qquad I_1 = \operatorname{tr}(\mathbf{S}) \tag{5.25}$$

Let's go ahead and perform the same operation on the second and third invariants to round out the whole set:

$$II_{\mathbf{C}} = \operatorname{tr}\left(\mathbf{F}^T\mathbf{F}\mathbf{F}^T\mathbf{F}\right) \qquad \mathbf{F}^T\mathbf{F} \to \mathbf{S} \qquad I_2 = \operatorname{tr}\left(\mathbf{S}^2\right) \tag{5.26}$$

$$III_{\mathbf{C}} = \det\left(\mathbf{F}^T\mathbf{F}\right) \qquad \mathbf{F}^T\mathbf{F} \to \mathbf{S} \qquad I_3 = \det(\mathbf{S}) \tag{5.27}$$

Is this a good set of invariants? One way to decide this is to see if we can express the original Cauchy-Green invariants using them. If we can, then we will know for sure that new invariants can describe a *super-set* of the phenomena that the Cauchy-Green ones could. That would be an excellent indicator that this new set is legitimate.

Well, $I_{\mathbf{C}}$ is equivalent to I_2. We saw that $I_{\mathbf{C}} = \operatorname{tr}\left(\mathbf{S}^2\right)$ back in §5.3.1.3, so that one is easy. Next up, $III_{\mathbf{C}} = I_3^2$. We had already gotten hints about something like this back in §5.2.4, when $J = \det \mathbf{F}$ sure looked a lot more fundamental than $III_{\mathbf{C}} = \det\left(\mathbf{F}^T\mathbf{F}\right)$. This suspicion is now confirmed, since $I_3 = \det \mathbf{S} = \det \mathbf{F} = J$. This determinant of $\mathbf{F}$ already captures all the essential information, so it should have an invariant all to itself.

This leaves $II_{\mathbf{C}}$. Showing the equivalence to our new invariants is straightforward in 2D:

$$\begin{aligned} II_{\mathbf{C}} &= I_2^2 - 2I_3^2 \\ &= (\sigma_x^2 + \sigma_y^2)^2 - 2(\sigma_x\sigma_y)^2 \\ &= \sigma_x^4 + \sigma_y^4 + 2(\sigma_x\sigma_y)^2 - 2(\sigma_x\sigma_y)^2 \\ &= \sigma_x^4 + \sigma_y^4. \end{aligned}$$

The 3D version takes more work, but is possible:

$$II_{\mathbf{C}} = \frac{1}{2}\left(I_2^2 - I_1^4\right) + I_1^2 I_2 + 4I_1I_3.$$

If you already believe me that this expression is legitimate, feel free to skip forward to §5.3.4. Otherwise, here's the proof in all its grisly glory. First some auxiliary variables:

$$\begin{aligned} \alpha &= (\sigma_x\sigma_y)^2 + (\sigma_x\sigma_z)^2 + (\sigma_y\sigma_z)^2 \\ \beta &= \sigma_x^2\sigma_y\sigma_z + \sigma_x\sigma_y^2\sigma_z + \sigma_x\sigma_y\sigma_z^2 \\ \gamma &= \sigma_x^3(\sigma_y + \sigma_z) + \sigma_y^3(\sigma_x + \sigma_z) + \sigma_z^3(\sigma_x + \sigma_y) \\ I_2^2 &= II_{\mathbf{C}} + 2\alpha \\ I_1^4 &= II_{\mathbf{C}} + 6\alpha + 12\beta + 4\gamma \\ I_1^2 I_2 &= II_{\mathbf{C}} + 2\alpha + 2\beta + 2\gamma \\ I_1 I_3 &= \beta \end{aligned}$$

Now let's plug them into our expression:

$$\begin{aligned} II_{\mathbf{C}} &= \frac{1}{2}\left(I_2^2 - I_1^4\right) + I_1^2 I_2 + 4I_1I_3 \\ II_{\mathbf{C}} &= \frac{1}{2}\left(II_{\mathbf{C}} + 2\alpha - II_{\mathbf{C}} - 6\alpha - 12\beta - 4\gamma\right) + I_1^2 I_2 + 4I_1I_3 && \text{(Expand } I_2^2 \text{ and } I_1^4\text{)} \\ II_{\mathbf{C}} &= -4\left(\alpha + 3\beta + \gamma\right) + I_1^2 I_2 + 4I_1I_3 && \text{(Simplify the expansion)} \\ II_{\mathbf{C}} &= -4\left(\alpha + 3\beta + \gamma\right) + II_{\mathbf{C}} + 2\alpha + 2\beta + 2\gamma + 4\beta && \text{(Expand } I_1^2 I_2 \text{ and } I_1 I_3\text{)} \\ II_{\mathbf{C}} &= -4\left(\alpha + 3\beta + \gamma\right) + II_{\mathbf{C}} + 4(\alpha + 3\beta + \gamma) && \text{(Simplify the expansion)} \\ II_{\mathbf{C}} &= II_{\mathbf{C}} \end{aligned}$$

Our new invariants look legitimate. They are a super-set of Cauchy-Green.

5.3.4 Does ARAP Work Now?

Now let's revisit §5.2.5 and try to write the ARAP energy in terms of our *new* invariants:

$$I_1 = \operatorname{tr} \mathbf{S} \qquad I_2 = \operatorname{tr}\left(\mathbf{S}^2\right) \qquad I_3 = \det \mathbf{S} \tag{5.28}$$

Can we do it?

$$\begin{aligned}
\Psi_{\text{ARAP}} &= \|\mathbf{F} - \mathbf{R}\|_F^2 && (5.29)\\
&= \|\mathbf{F}\|_F^2 - 2\operatorname{tr}\left(\mathbf{F}^T\mathbf{R}\right) + \|\mathbf{R}\|_F^2 && \text{(Apply B.16)}\\
&= I_2 - 2I_1 + 3 && \text{(Since } \|\mathbf{F}\|_F^2 = I_\mathbf{C} = I_2 \text{ and } \operatorname{tr}\mathbf{F}^T\mathbf{R} = \operatorname{tr}\mathbf{S} = I_1)
\end{aligned}$$

Hooray! Now let's see if we can get the PK1, since we will need it to compute a force:

$$\begin{aligned}
\frac{\partial \Psi_{\text{ARAP}}}{\partial \mathbf{F}} &= \frac{\partial}{\partial \mathbf{F}}\left(I_2 - 2I_1 + 3\right) && (5.30)\\
&= \frac{\partial I_2}{\partial \mathbf{F}} - 2\frac{\partial I_1}{\partial \mathbf{F}}. && (5.31)
\end{aligned}$$

Do we know these derivatives? Since $I_2 = I_\mathbf{C}$, we already have that one from §4.2.3:

$$\frac{\partial I_2}{\partial \mathbf{F}} = \frac{\partial I_\mathbf{C}}{\partial \mathbf{F}} = 2\mathbf{F}.$$

What about $\frac{\partial I_1}{\partial \mathbf{F}}$? Since I_1 is our brand-new invariant, we're going to need to establish a new expression. Fortunately, it turns out to be quite simple:

$$\frac{\partial I_1}{\partial \mathbf{F}} = \frac{\partial \operatorname{tr} \mathbf{S}}{\partial \mathbf{F}} = \frac{\partial \operatorname{tr}\left(\mathbf{R}\mathbf{F}^T\right)}{\partial \mathbf{F}} = \mathbf{R}. \tag{5.32}$$

It's just the rotation component of $\mathbf{F}$! Now we can write the full PK1:

$$\begin{aligned}
\frac{\partial \Psi_{\text{ARAP}}}{\partial \mathbf{F}} &= \frac{\partial I_2}{\partial \mathbf{F}} - 2\frac{\partial I_1}{\partial \mathbf{F}} && (5.33)\\
&= 2\mathbf{F} - 2\mathbf{R} && (5.34)\\
&= 2(\mathbf{F} - \mathbf{R}) && (5.35)
\end{aligned}$$

This matches what we got in §4.2.4, so everything checks out[9]. Now if we can just get an expression for the Hessian of ARAP, then we're finally all done:

$$\begin{aligned}
\frac{\partial^2 \Psi_{\text{ARAP}}}{\partial \mathbf{F}^2} &= 2\frac{\partial}{\partial \mathbf{F}}(\mathbf{F} - \mathbf{R}) && (5.36)\\
&= 2\left(\frac{\partial \mathbf{F}}{\partial \mathbf{F}} - \frac{\partial \mathbf{R}}{\partial \mathbf{F}}\right). && (5.37)
\end{aligned}$$

Oh ... right. There was that $\frac{\partial \mathbf{R}}{\partial \mathbf{F}}$ rotation gradient term from §4.2.4 that we didn't know how to deal with. Does our brand new I_1 invariant help us out with $\frac{\partial \mathbf{R}}{\partial \mathbf{F}}$? Not in any way that I can see. ARAP is making trouble again. ARAP, why can't you just behave?

[9] I dropped the $\frac{\mu}{2}$ term, just to make things a little cleaner.

5.4 The Eigenmatrices of the Rotation Gradient

We have a new set of promising invariants, but one key piece is missing. If we can't figure out a good way to write down $\frac{\partial^2 I_1}{\partial \mathbf{F}^2} = \frac{\partial \mathbf{R}}{\partial \mathbf{F}}$ then everything was for naught. Or rather, we can use some ugly numerical method to obtain $\frac{\partial \mathbf{R}}{\partial \mathbf{F}}$ (like the one in Papadopoulo and Lourakis (2000)), and our quest to find a clean and slick method for finding Hessians will come to a messy and vaguely unsatisfying end.

The core issue is that we don't know how to obtain a tidy symbolic derivative for the numerical quantity $\mathbf{R}$. However, as we will see, this derivative does indeed have a simple and clean structure, *if* you look at it from the right perspective.

To get to that perspective, we need to look along two counter-intuitive directions:

- The *eigendecomposition* of $\frac{\partial \mathbf{R}}{\partial \mathbf{F}}$ will have a simple structure, i.e. we will be able to form it directly $\mathbf{Q}$ and Λ from $\frac{\partial \mathbf{R}}{\partial \mathbf{F}} = \mathbf{Q}\Lambda\mathbf{Q}^T$. This is the *opposite* of how things usually go.

  Usually there's a simple way to write down some matrix $\mathbf{A}$, and then you compute the eigendecomposition, $\mathbf{A} = \mathbf{Q}\Lambda\mathbf{Q}^T$, using some byzantine numerical method. The entries of $\mathbf{Q}$ and Λ then form an impenetrable sea of numbers with no discernible structure. Only in a few special cases[10] do you ever get a simple, closed-form representation of $\mathbf{Q}$.

- We won't be able to see anything if we flatten $\frac{\partial \mathbf{R}}{\partial \mathbf{F}}$ out to a matrix and look at its eigenvectors. Instead, we have to recognize $\frac{\partial \mathbf{R}}{\partial \mathbf{F}}$ for what it really is: a 4th-order tensor (§4.1). We need to gaze at its eigenstructure in all its 4th-order glory.

  In the 4th-order universe, an eigen*vector* becomes an eigen*matrix*. The eigenmatrix view will reveal the fundamental structure of the rotation gradient.

Let's take the first point for granted: the eigendecomposition of the rotation gradient contains a simple structure that is waiting to be discovered. From there, you're probably wondering what the eigendecomposition of a 4th-order tensor looks like, and what the heck an eigenmatrix is.

5.4.1 What's an Eigenmatrix?

Let's recall the basic eigenvalue problem:

$$\mathbf{A}\mathbf{q}_0 = \lambda_0 \mathbf{q}_0.$$

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.

[10]Like Fourier series popping out of the heat equation. That's a pretty important one.

For some 4th-order tensor

$$\mathbb{A} = \begin{bmatrix} \begin{bmatrix} a_0 & a_2 \\ a_1 & a_3 \end{bmatrix} & \begin{bmatrix} a_8 & a_{10} \\ a_9 & a_{11} \end{bmatrix} \\ \\ \begin{bmatrix} a_4 & a_6 \\ a_5 & a_7 \end{bmatrix} & \begin{bmatrix} a_{12} & a_{14} \\ a_{13} & a_{15} \end{bmatrix} \end{bmatrix} \tag{5.38}$$

we can define a similar eigenpair, $(\lambda_0, \mathbf{Q}_0)$. The equivalent eigenvalue problem is,

$$\mathbb{A} : \mathbf{Q}_0 = \lambda_0 \mathbf{Q}_0,$$

where instead of the $\mathbf{A}\mathbf{q}_0$ matrix-vector multiply, we have a double-contraction. We can write down everything down in its grisly, fully-expanded verbosity like this:

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

Again, $\mathbf{Q}_0$ is a special matrix that, even after going through that baroque double-contraction operation, emerges essentially unscathed. Except, it was scaled by λ_0.

Now the thing is, the $\mathbf{A}\mathbf{q}_0 = \lambda_0\mathbf{q}_0$ and $\mathbb{A} : \mathbf{Q}_0 = \lambda_0\mathbf{Q}_0$ versions are exactly equivalent. If you find some eigenmatrix $\mathbf{Q}_0$ of the tensor $\mathbb{A}$, then it is also an eigenvector of the flattened out matrix $\mathbf{A} = \operatorname{vec}\mathbb{A}$. Again, spelling everything out verbosely:

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

This is good news! We don't have to come up with a whole new suite of numerical methods to find the eigenmatrices of some tensor $\mathbb{A}$. Instead, we can just flatten things out to $\mathbf{A} = \operatorname{vec} \mathbb{A}$, and run whatever eigenvalue routine you want from Matlab, Eigen, or LAPACK. The $(\lambda_i, \mathbf{q}_i)$ pairs that come back can then just be re-folded back up into the eigenmatrix-based pair, $(\lambda_i, \mathbf{Q}_i)$.

But if this is the case, why do we even care about eigenmatrices? If they contain the exact same entries as eigenvectors, just repackaged into matrix form, why not use the more familiar vector form?

5.4.2 Structures Lurk in the Decomposition of an Eigenmatrix

The reason we care about eigen*matrices* is that once you have a matrix, you can apply a variety of matrix-based tools that are not available for vectors. For example, you could take the eigendecomposition of an eigenmatrix. It would be weirdly meta-mathematical and fractal, but you could do it. Equivalently, you could take the QR decomposition or SVD of an eigenmatrix.

Or, we can apply the results of the SVD from related matrices, and see if those reveals any new structures. The most important matrix we've been dealing with throughout these chapters, and ever since I highlighted it in bright red back in §2.3.3, is the deformation gradient, $\mathbf{F}$. Let's say that we had the SVD of that deformation gradient,

$$\mathbf{F} = \mathbf{U}\Sigma\mathbf{V}^T.$$

Let's also say that we went ahead and did the horribly ugly thing by computing the rotation gradient, $\frac{\partial \mathbf{R}}{\partial \mathbf{F}}$, using the numerical approach of Papadopoulo and Lourakis (2000). I'll even give you the actual Matlab code I used to do this in Figs. 5.5 and 5.6.

To keep things simple, let's look at the 2D version of $\frac{\partial \mathbf{R}}{\partial \mathbf{F}}$. In that case, if we poke at the matrices coming out of `DRDF_Horrible` in Fig. 5.5, we will quickly discover that it's a rank-one matrix composed of a single eigenpair, $(\lambda_0, \mathbf{Q}_0)$. Hoping to discover some simple structure, we can stare at the entries of $\mathbf{Q}_0$, or even $\mathbf{q}_0 = \operatorname{vec} \mathbf{Q}_0$. As usual, it will just be an impenetrable wall of numbers. Not helpful.

We can poke at it more in Matlab and take its SVD, QR, and eigendecomposition to see if something interesting shows up. Alas, I can tell you from experience that this will not yield any insight either. Finally, while running a few errands over the weekend, and chewing this problem over in the backs of ours minds, we get the nagging feeling that everything we're dealing with really just boils down to $\mathbf{F}$. Why not pound on $\mathbf{Q}_0$ using the decomposition of $\mathbf{F}$ and see what happens?

We can try rotating $\mathbf{Q}_0$ into the same space as $\mathbf{F}$ using $\mathbf{U}\mathbf{Q}_0\mathbf{V}^T$. Nothing much to see here, just the usual sea of numbers. How about we rotate $\mathbf{Q}_0$ *out* of the space of $\mathbf{F}$:

$$\mathbf{U}^T\mathbf{Q}_0\mathbf{V} = \begin{bmatrix} 0 & -0.70711 \\ 0.70711 & 0 \end{bmatrix}? \tag{5.39}$$

```
function [final] = DRDF_Horrible(R,S)
  H = zeros(4,4);
  for index = 1:4
    i = mod(index - 1, 2);
    j = floor((index - 1) / 2);

    % promote it by a dimension
    i = i + 1;
    j = j + 1;
    index3 = i + (j - 1) * 3;

    R3 = eye(3,3);
    S3 = eye(3,3);
    R3(1:2, 1:2) = R;
    S3(1:2, 1:2) = S;

    [DR3, DS3] = DRDF_Column(R3,S3,index3);

    DR = DR3(1:2,1:2);
    DS = DS3(1:2,1:2);
    column = DR;

    H(:,index) = reshape(column, 4, 1);
  end
  final = H;
endfunction
```

Figure 5.5.: Matlab code to compute the rotation gradient using a horrible numerical method. We only use this temporarily on our way to finding a clean expression for $\frac{\partial \mathbf{R}}{\partial \mathbf{F}}$.

```
function [DR, DS] = DRDF_Column(R,S,index)
  i = mod(index - 1, 3);
  j = floor((index - 1) / 3);

  % Matlab indexing
  i = i + 1;
  j = j + 1;

  eiej = zeros(3,3);
  eiej(i,j) = 1;

  G = (eye(3,3) * trace(S) - S) * R';
  Rij = R' * eiej;

  % extract the skew vector
  Rijsym = (Rij - Rij') * 0.5;
  skew = zeros(3,1);

  skew(1) = -Rijsym(2,3);
  skew(2) =  Rijsym(1,3);
  skew(3) = -Rijsym(1,2);
  skew = 2 * skew;

  omega = (G^-1) * skew;

  cross = [        0 -omega(3)   omega(2);
            omega(3)         0 -omega(1);
           -omega(2)  omega(1)         0];
  DR = cross * R;
  DS = R' * (eiej - DR * S);
end
```

Figure 5.6.: Matlab code to compute one column of the 3D rotation gradient. Like in Fig. 5.5, we only use this as a waypoint on our journey to find a clean expression for $\frac{\partial \mathbf{R}}{\partial \mathbf{F}}$. in `DRDF_Horrible`, I pinned one dimension of this 3D version so that I could mess around with it in 2D.

Hang on a minute. A bunch of zeros appeared, which is usually a good sign. Also, that 0.70711 sure looks familiar from trigonometry. Wasn't it $\sqrt{2}/2 = 1/\sqrt{2}$, or something like that? If we pull that constant out, we see that the structure of $\mathbf{Q}_0$ is actually:

$$\mathbf{Q}_0 = \frac{1}{\sqrt{2}}\mathbf{U}\begin{bmatrix}0 & -1\\ 1 & 0\end{bmatrix}\mathbf{V}^T. \tag{5.40}$$

That's ... pretty simple! It's similar to $\mathbf{F}$, but instead of the singular value matrix Σ in the middle, it's just the constant matrix $\begin{bmatrix}0 & -1\\ 1 & 0\end{bmatrix}$. This matrix corresponds to an infinitesimal rotation in the 2D plane, so we call the whole thing a *twist* matrix. The $1/\sqrt{2}$ 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 center matrix. Since $\sqrt{1^2 + (-1)^2} = \sqrt{2}$, the normalization factor is $1/\sqrt{2}$.

If this entire process sounds improbably informal, and worryingly outside the hypothesis-experiment-conclusion loop of scientific discovery you were taught in high school, just know that real-world research doesn't always conform to that model. I know I'm not the only researcher who stumbles across new things by just messing around in Matlab.

5.4.3 What About the Eigenvalue?

Let's look again at our horribly ugly, numerically-based rotation gradient code, `DRDF_Horrible`. Just to see how the λ_0 eigenvalue evolves, we can start plugging easy-looking integers into the singular values, like $\sigma_x = 2$ and $\sigma_y = 3$, $\sigma_x = 2$ and $\sigma_y = 5$, and so on. You can literally try this out yourself – the code to run these experiments is in Fig. 5.7.

From there, we can see λ_0 take on values that look suspiciously like some tidy fractions. For example, $(\sigma_0, \sigma_1) = (2, 3)$ yielded 0.4, and $(\sigma_0, \sigma_1) = (2, 5)$ yields 0.28571. Just to confirm our suspicion that these are fractions, we can push them through Matlab's `rats` function, and see that `rats`$(0.4) \rightarrow 2/5$ and `rats`$(0.28571) \rightarrow 2/7$ (last line, Fig. 5.7).

Running a variety of experiments with different $\mathbf{U}$, Σ and $\mathbf{V}$, it will quickly become clear that the eigenvalue takes the general form:

$$\lambda_0 = \frac{2}{\sigma_x + \sigma_y}. \tag{5.41}$$

If the informality and empiricism of this process bothers you, think of it like running experiments to build up some numerical intuition. Once you see a pattern, you can posit a testable theory.

5.4.4 Building the Rotation Gradient (Finally)

We now have the single eigenpair that composes the 2D rotation gradient:

$$\lambda_0 = \frac{2}{\sigma_x + \sigma_y} \qquad \mathbf{Q}_0 = \frac{1}{\sqrt{2}}\mathbf{U}\begin{bmatrix}0 & -1\\ 1 & 0\end{bmatrix}\mathbf{V}^T. \tag{5.42}$$

```matlab
% Pick your sigmas
Sigma = [2 0,
         0 5];

% Pick some rotations
angleU = 0;
U = [cos(angleU) -sin(angleU);
     sin(angleU)  cos(angleU)];

angleV = 0;
V = [cos(angleV) -sin(angleV);
     sin(angleV)  cos(angleV)];

% compose your F
F = U * Sigma * V';

% compose your polar decomposition
R = U * V';
S = V * Sigma * V';

% get the rotation gradient
H = DRDF_Horrible(R,S)

% what's the eigendecomposition look like?
[Q Lambda] = eig(H)

% is the eigenvalue a rational number?
rats(Lambda(4,4))
```

Figure 5.7.: Matlab code to experiment with the single eigenvalue of the rotation gradient. Plug different integers into the diagonal of `Sigma` and see suspiciously rational-looking quantities pop out at the end.

```
function [final] = DRDF_Clean_2D(U, Sigma, V)
  Q0 = (1 / sqrt(2)) * U * [0 -1; 1 0] * V';
  q0 = vec(Q0);
  lambda0 = 2 / (Sigma(1,1) + Sigma(2,2));
  final = lambda0 * (q0 * q0');
endfunction
```

Figure 5.8.: Fast and compact Matlab code for building the 2D rotation gradient. This implements Eqn. 5.43. Looks nicer than Figs. 5.5 and 5.6, don't you think?

We can then use an outer product to arrive at a very simple way to compute the flattened rotation gradient:[11]

$$\operatorname{vec}\left(\frac{\partial \mathbf{R}}{\partial \mathbf{F}}\right) = \frac{2}{\sigma_x + \sigma_y} \operatorname{vec}\left(\mathbf{Q}_0\right) \operatorname{vec}\left(\mathbf{Q}_0\right)^T. \tag{5.43}$$

This also maps onto some very simple Matlab code, given in Fig. 5.8. We can chug through the entire experimental process again to find a similar expression for the 3D rotation gradient. I will spare you all the intermediate Matlab muddling, and instead tell you that the 3D rotation gradient is rank-3, and the three eigenpairs follow a pattern that is similar to 2D:

$$\lambda_0 = \frac{2}{\sigma_x + \sigma_y} \qquad \mathbf{Q}_0 = \frac{1}{\sqrt{2}}\mathbf{U}\begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}\mathbf{V}^T \tag{5.44}$$

$$\lambda_1 = \frac{2}{\sigma_y + \sigma_z} \qquad \mathbf{Q}_1 = \frac{1}{\sqrt{2}}\mathbf{U}\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & -1 & 0 \end{bmatrix}\mathbf{V}^T \tag{5.45}$$

$$\lambda_2 = \frac{2}{\sigma_x + \sigma_z} \qquad \mathbf{Q}_2 = \frac{1}{\sqrt{2}}\mathbf{U}\begin{bmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \\ -1 & 0 & 0 \end{bmatrix}\mathbf{V}^T. \tag{5.46}$$

We can then use the same outer product form as the 2D case to compute the final $\operatorname{vec}\left(\frac{\partial \mathbf{R}}{\partial \mathbf{F}}\right)$ matrix,

$$\operatorname{vec}\left(\frac{\partial \mathbf{R}}{\partial \mathbf{F}}\right) = \sum_{i=0}^{2} \lambda_i \operatorname{vec}\left(\mathbf{Q}_i\right) \operatorname{vec}\left(\mathbf{Q}_i\right)^T, \tag{5.47}$$

where λ_i and $\mathbf{Q}_i$ correspond to Eqns. 5.44 to 5.46. The corresponding Matlab code is given in Fig. 5.9.

[11] Oh hey look it's that counter-intuitive eigendecomposition I warned you would happen in the first bullet point back at the beginning of §5.4.

```matlab
function [H] = DRDF_Clean_3D(U, Sigma, V)
  % get the twist modes
  T0 = [0 -1  0;
        1  0  0;
        0  0  0];
  T0 = (1 / sqrt(2)) * U * T0 * V';

  T1 = [0  0  0;
        0  0  1;
        0 -1  0];
  T1 = (1 / sqrt(2)) * U * T1 * V';

  T2 = [ 0  0  1;
         0  0  0;
        -1  0  0];
  T2 = (1 / sqrt(2)) * U * T2 * V';

  % get the flattened versions
  t0 = vec(T0);
  t1 = vec(T1);
  t2 = vec(T2);

  % get the singular values
  sx = Sigma(1,1);
  sy = Sigma(2,2);
  sz = Sigma(3,3);

  H = (2 / (sx + sy)) * (t0 * t0');
  H = H + (2 / (sy + sz)) * (t1 * t1');
  H = H + (2 / (sx + sz)) * (t2 * t2');
end
```

Figure 5.9.: Fast and compact Matlab code for building the 3D rotation gradient. This is the outer product of the eigenmatrices from Eqns. 5.44 - 5.46.

5.5 Building a Generic Hessian (Finally)

We now have everything in hand needed to compute the Hessian for an isotropic energy. Remember what happened back in §5.2.2: we wrote things in terms of the Cauchy-Green invariants, and it looked like we were on easy street for building the final expression. Then we tried to approach on ARAP, which didn't fit into the Cauchy-Green formalism, and blew it all to smithereens.

Now we *do* have a set of invariants that encompass ARAP, so let's use them to get a similarly generic approach. Starting from our new Smith et al. (2019) invariants,

$$I_1 = \mathrm{tr}\left(\mathbf{S}\right) \qquad I_2 = \mathrm{tr}\left(\mathbf{F}^T\mathbf{F}\right) \qquad I_3 = \det \mathbf{F}, \tag{5.48}$$

we have the gradient ($\mathbf{g}_*$) and Hessian ($\mathbf{H}_*$) of each:

$$\begin{aligned}
\mathbf{g}_1 &= \mathrm{vec}\left(\mathbf{R}\right) & \mathbf{H}_1 &= \sum_{i=0}^{2} \lambda_i \,\mathrm{vec}\left(\mathbf{Q}_i\right)\mathrm{vec}\left(\mathbf{Q}_i\right)^T \text{ (Eqn. 5.47)}\\
\mathbf{g}_2 &= \mathrm{vec}\left(2\mathbf{F}\right) & \mathbf{H}_2 &= 2\mathbf{I}_{9\times 9}\\
\mathbf{g}_J &= \mathrm{vec}\left(\left[\begin{array}{c|c|c} \mathbf{f}_1\times\mathbf{f}_2 & \mathbf{f}_2\times\mathbf{f}_0 & \mathbf{f}_0\times\mathbf{f}_1 \end{array}\right]\right) & \mathbf{H}_J &= \begin{bmatrix} \mathbf{0} & -\hat{\mathbf{f}}_2 & \hat{\mathbf{f}}_1\\ \hat{\mathbf{f}}_2 & \mathbf{0} & -\hat{\mathbf{f}}_0\\ -\hat{\mathbf{f}}_1 & \hat{\mathbf{f}}_0 & \mathbf{0}\end{bmatrix}.
\end{aligned}$$

Now we can write an alternative expression for the energy Hessian:

$$\mathrm{vec}\left(\frac{\partial^2\Psi}{\partial\mathbf{F}^2}\right) = \frac{\partial^2\Psi}{\partial I_1^2}\mathbf{g}_1\mathbf{g}_1^T + \frac{\partial\Psi}{\partial I_1}\mathbf{H}_1 + \frac{\partial^2\Psi}{\partial I_2^2}\mathbf{g}_2\mathbf{g}_2^T + \frac{\partial\Psi}{\partial I_2}\mathbf{H}_2 + \frac{\partial^2\Psi}{\partial I_3^2}\mathbf{g}_3\mathbf{g}_3^T + \frac{\partial\Psi}{\partial I_3}\mathbf{H}_3 \tag{5.49}$$

$$= \sum_{i=1}^{3} \frac{\partial^2\Psi}{\partial I_i^2}\mathbf{g}_i\mathbf{g}_i^T + \frac{\partial\Psi}{\partial I_i}\mathbf{H}_i. \tag{5.50}$$

Finally, we follow the same three-step process as before, but using the Smith et al. (2019) invariants:

1. Re-write your energy Ψ using I_1, I_2 and I_3.
2. Derive the scalar derivatives, $\frac{\partial\Psi}{\partial I_1}$, $\frac{\partial^2\Psi}{\partial I_1^2}$, $\frac{\partial\Psi}{\partial I_2}$, $\frac{\partial^2\Psi}{\partial 1_2^2}$, $\frac{\partial\Psi}{\partial I_3}$ and $\frac{\partial^2\Psi}{\partial I_3^2}$.
3. Plug the results into Eqn. 5.50.

Let's put this algorithm through its paces. First, let's confirm it does the same thing as Cauchy-Green by confirming that we still get the same results for Neo-Hookean.

5.5.1 Neo-Hookean, the Smith et al. (2019) Way

Once again, here's Bonet and Wood (2008)-style Neo-Hookean:

$$\Psi_{\mathrm{BW08}} = \frac{\mu}{2}(\|\mathbf{F}\|_F^2 - 3) - \mu\log(J) + \frac{\lambda}{2}\left(\log(J)\right)^2$$

Step 1: Rewrite using invariants. This takes the form:

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

We're now using I_3 instead of J, or the even more ridiculous-looking $\sqrt{III_{\mathbf{C}}}$. So, things are already looking cleaner.

Step 2: Take the invariant derivatives. These are similar to before, which is good sign:

$$\frac{\partial \Psi_{\text{BW08}}}{\partial I_1} = 0 \qquad \frac{\partial^2 \Psi_{\text{BW08}}}{\partial I_1^2} = 0 \tag{5.51}$$

$$\frac{\partial \Psi_{\text{BW08}}}{\partial I_2} = \frac{\mu}{2} \qquad \frac{\partial^2 \Psi_{\text{BW08}}}{\partial I_2} = 0 \tag{5.52}$$

$$\frac{\partial \Psi_{\text{BW08}}}{\partial I_3} = \frac{\lambda \log I_3 - \mu}{I_3} \qquad \frac{\partial^2 \Psi_{\text{BW08}}}{\partial I_3^2} = \frac{\lambda(1 - \log I_3) + \mu}{I_3^2}. \tag{5.53}$$

Step 3: Plug into the chain rule. Plugging into Eqn. 5.50, but dropping everything with a zero coefficient, we get:

$$\operatorname{vec}\left(\frac{\partial^2 \Psi_{\text{BW08}}}{\partial \mathbf{F}^2}\right) = \mu \mathbf{I}_{9\times 9} + \frac{\lambda(1 - \log I_3) + \mu}{I_3^2}\mathbf{g}_3\mathbf{g}_3^T + \frac{\lambda \log I_3 - \mu}{I_3}\mathbf{H}_3.$$

It matches both Eqn. 4.25 and Eqn. 5.17: mission accomplished! The Smith et al. (2019) Way is looking legit.

5.5.2 ARAP, the Smith et al. (2019) Way

Now let's look again at our troublemaking friend, ARAP:

$$\Psi_{\text{ARAP}} = \|\mathbf{F} - \mathbf{R}\|_F^2.$$

Step 1: Rewrite using invariants. Unlike the last time we tried this in §5.2.5, we now have the I_1 invariant at our disposal. We actually already did this rewrite in §5.3.4, and it worked out to:

$$\Psi_{\text{ARAP}} = I_2 - 2I_1 + 3.$$

Step 2: Take the invariant derivatives. Here we go:

$$\frac{\partial \Psi_{\text{ARAP}}}{\partial I_1} = -2 \qquad \frac{\partial^2 \Psi_{\text{ARAP}}}{\partial I_1^2} = 0 \tag{5.54}$$

$$\frac{\partial \Psi_{\text{ARAP}}}{\partial I_2} = 1 \qquad \frac{\partial^2 \Psi_{\text{ARAP}}}{\partial I_2} = 0 \tag{5.55}$$

$$\frac{\partial \Psi_{\text{ARAP}}}{\partial I_3} = 0 \qquad \frac{\partial^2 \Psi_{\text{ARAP}}}{\partial I_3^2} = 0. \tag{5.56}$$

```
function [H] = ARAP_Hessian(F)
  [U Sigma V] = svd_rv(F);

  % get the twist modes
  T0 = [0 -1 0; 1 0 0; 0 0 0];
  T0 = (1 / sqrt(2)) * U * T0 * V';

  T1 = [0 0 0; 0 0 1; 0 -1 0];
  T1 = (1 / sqrt(2)) * U * T1 * V';

  T2 = [0 0 1; 0 0 0; -1 0 0];
  T2 = (1 / sqrt(2)) * U * T2 * V';

  % get the flattened versions
  t0 = vec(T0);
  t1 = vec(T1);
  t2 = vec(T2);

  % get the singular values in an order that is consistent
  % with the numbering in [Smith et al. 2019]
  s0 = Sigma(1,1);
  s1 = Sigma(2,2);
  s2 = Sigma(3,3);

  H = 2 * eye(9,9);
  H = H - (4 / (s0 + s1)) * (t0 * t0');
  H = H - (4 / (s1 + s2)) * (t1 * t1');
  H = H - (4 / (s0 + s2)) * (t2 * t2');
end
```

Figure 5.10.: Matlab code to compute the exact Hessian of the ARAP energy in 3D. The code for `svd_rv` is given in Fig. F.1.

This time it was easy. I didn't even have to go running to Mathematica for this one.

Step 3: Plug into the chain rule. Again but dropping everything with a zero coefficient, we get:

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

FINALLY, A SIMPLE EXPRESSION. I ALWAYS KNEW YOU HAD IT IN YOU, ARAP. The Matlab code to compute this is shown in Fig. 5.10.

5.5.3 Symmetric Dirichlet, the Smith et al. (2019) Way

Just to build some confidence that this approach is indeed generic and we're not fooling ourselves, let's try it out on an energy we haven't seen before, the Symmetric Dirichlet energy from Smith and Schaefer (2015a):

$$\Psi_{\text{Symmetric}} = \|\mathbf{F}\|_F^2 + \|\mathbf{F}^{-1}\|_F^2. \tag{5.57}$$

Step 1: Rewrite using invariants. We're not quite on easy street with this energy. Is it possible to write $\|\mathbf{F}^{-1}\|^2$ in terms of the invariants? Indeed it is, and just to keep things moving along, I'm going to pull it in as an identity from Eqn. B.36. Then we get:

$$\Psi_{\text{Symmetric}} = I_2 + \frac{1}{4}\left(\frac{I_1^2 - I_2}{I_3}\right) - 2\frac{I_1}{I_3}. \tag{5.58}$$

Step 2: Take the invariant derivatives. This part is tedious but straightforward. As always, you can have Mathematica or Matlab do this part for you:

$$\frac{\partial \Psi_{\text{Symmetric}}}{\partial I_1} = \frac{1}{I_3}\left(\frac{I_1}{2} - 2\right) \qquad \frac{\partial^2 \Psi_{\text{Symmetric}}}{\partial I_1^2} = \frac{1}{2I_3} \tag{5.59}$$

$$\frac{\partial \Psi_{\text{Symmetric}}}{\partial I_2} = 1 - \frac{1}{4I_3} \qquad \frac{\partial^2 \Psi_{\text{Symmetric}}}{\partial I_2} = 0 \tag{5.60}$$

$$\frac{\partial \Psi_{\text{Symmetric}}}{\partial I_3} = \frac{1}{(I_3)^2}\left(2I_1 - \frac{I_1^2 - I2}{4}\right) \qquad \frac{\partial^2 \Psi_{\text{Symmetric}}}{\partial I_3^2} = \frac{1}{(I_3)^3}\left(-4I_1 + \frac{I_1^2 - I2}{2}\right). \tag{5.61}$$

Once again, I used Mathematica for this, just a minute ago. It's not cheating.

Step 3: Plug into the chain rule. It's not *quite* as clean and pretty as the other energies:

$$\begin{aligned}\text{vec}\left(\frac{\partial^2 \Psi_{\text{Symmetric}}}{\partial \mathbf{F}^2}\right) =& \frac{1}{2I_3}\mathbf{g}_1\mathbf{g}_1^T + \frac{1}{I_3}\left(\frac{I_1}{2} - 2\right)\mathbf{H}_1 + \left(1 - \frac{1}{4I_3}\right)\mathbf{H}_2 + \\ & \frac{1}{(I_3)^3}\left(-4I_1 + \frac{I_1^2 - I2}{2}\right)\mathbf{g}_3\mathbf{g}_3^T + \frac{1}{(I_3)^2}\left(2I_1 - \frac{I_1^2 - I2}{4}\right)\mathbf{H}_3.\end{aligned}$$

But, it worked, once we got past the $\|\mathbf{F}^{-1}\|^2$ hurdle. The system looks legit! You now have a mechanical method for computing the Hessian of any isotropic energy. You need to take some high-school-level derivatives, and maybe dig out a new identity during **Step 1**. Otherwise, the process is close to automatic.

Chapter 6

A Friendlier Neo-Hookean Energy

This chapter is loose retelling of §3 from Smith et al. (2018). If you want to cut to the chase and see the final energy, it's in §6.3.3. It's *not* the final energy listed in Smith et al. (2018), but it *is* the one that is used internally at Pixar. More details appear in a footnote[9].

6.1 Cauchy-Green vs. Smith et al. (2019)

In the previous chapter, we found that the Cauchy-Green invariants, which have been used extensively for the last 80 years, were not up to the task of generating energy Hessians. ARAP's membership application to the Cauchy-Green Club was roundly rejected, causing all sorts of problems. Instead, we built a new, more inclusive club for ARAP: the invariants from Smith et al. (2019). Once ARAP stepped inside this new clubhouse, the simple and elegant structure that had been living inside it all along was finally revealed.

This raises a question though: where have these more inclusive invariants been for the last 80 years? Why did Mooney (1940) choose to use the unnecessarily restrictive Cauchy-Green invariants? Unfortunately, we can't ask Melvin Mooney directly because he passed away over 52 years ago. George Green has been gone for 179 years, and Augustin-Louis Cauchy for 163, so they can't help either. Thus, we are left to speculate. Having mulled this over for a while, I can think of two reasons.

6.1.1 Maybe Mooney Didn't Know About the Polar Decomposition

The Smith et al. (2019) invariants are based on the polar decomposition, which has been known in math circles for more than a century. For example, Higham (1986) cites Autonne (1902). However, it may not have been widely known in Mooney's immediate physics and rheology communities in the 1940s.

Just 15 years earlier in 1925, Werner Heisenberg had not known about basic matrix operations . While investigating quantum mechanics, he unknowingly re-derived basic linear algebraic identities, and did not make the connection until Max Born pointed it out to him (Gribbin (2011), Chapter 6). At the time, Heisenberg was in Göttingen, Germany, the epicenter of all math and science research, not the jerkwater nowheresville of Passaic, New Jersey where Mooney was sitting.[1]

Matrix notation had not even been standardized by 1940, and numerical methods were still an extremely niche discipline. The hardware to run them did not exist yet, though this would change a few years later when the Allies built massive machines to break Nazi codes. Even if these machines had existed, standard methods for computing the polar decomposition, such as those involving the SVD, would need techniques like Golub-Kahan bidiagonalization (Golub and Kahan (1965)). These were still 25 years in the future.

Thus, it would make sense that Mooney (1940) did not have the the polar decomposition at his fingertips. His paper does not list a single matrix, and is firmly a work of analysis.

6.1.2 Maybe Mooney Didn't Care About Inversion

Alternatively, it is possible that Mooney *did* know about the polar decomposition, or some analytic equivalent, and decided to ignore it. To see why, we should ask which physical phenomena are captured by Smith et al. (2019) that are missing from Cauchy-Green, and whether Mooney would have cared about them.

We saw a piece of this in §5.2.4 when I let you in on the secret that nobody actually uses that weird-looking $\sqrt{III_{\mathbf{C}}}$ term. Instead, everybody uses J. Later on in §5.3.3.2, we saw J getting the recognition it deserves with a promotion to first-class invariant, $I_3 = J = \det \mathbf{F}$.

As previously mentioned, this I_3 version preserves the *sign* of the determinant:

$$I_3 = \sigma_x \sigma_y \sigma_z \tag{6.1}$$

$$\sqrt{III_{\mathbf{C}}} = \sqrt{\sigma_x^2 \sigma_y^2 \sigma_z^2} = |\sigma_x \sigma_y \sigma_z|. \tag{6.2}$$

But then why does the Cauchy-Green version wrap the $|\cdot|$ around the whole thing? Why is it so eager to throw away the sign?

A negative determinant signals that an element has *inverted*[2]. As shown in Fig. 6.1, this happens if you squash a tetrahedron so hard that it pokes *inside out*. This sort of configuration is *not possible* in nature, but due to the fact that we can numerically represent phenomena that don't occur in nature[3], it can happen in the simulation.

[1]Down the road in Murray Hill, NJ, Bell Labs would become the epicenter soon enough.

[2]As far as I can tell, this use of the term *inversion* comes from Irving et al. (2004), though they acknowledge that the concept was observed earlier (Espinosa et al. (1998)). This creates an unfortunate nomenclature collision with *matrix inversion*, which is a related but distinct phenomenon. Thinking of it instead as *element reversal* or a *backwards element*, or the anatomically graphic *prolapsed element* may help here.

[3]Simulation could be viewed as coercing the computer into obeying the physics of our current universe, instead of drifting into one of the myriad other ones that the math permits, but our physical laws do not.

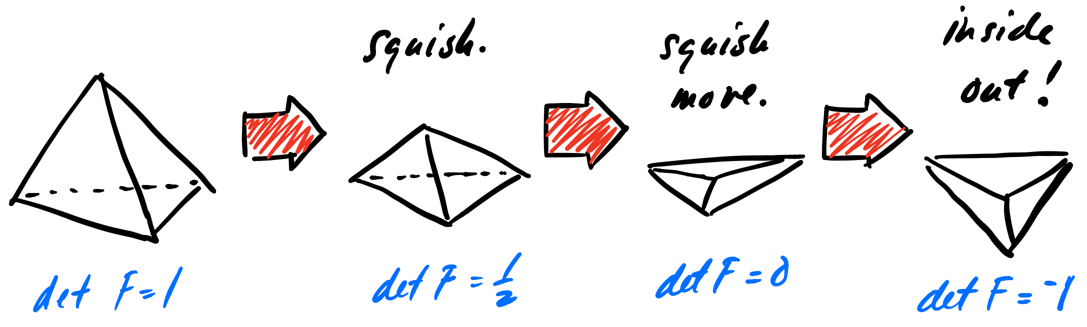


Figure 6.1.: A tetrahedron is sitting on a table (left). We squish its top vertex down (center, left) until it has been smashed into a pancake (center, right). We *keep pushing*, and the tetrahedron flips inside out! This is not physically possible, but it can *happen numerically*.

In animation, inversions happen *all the time*. We frequently squash things down to Wile-E-Coyote-style pancakes, and if we then keep squashing, we get an inversion. Thus, the energies we use need to be able to support this phenomenon.

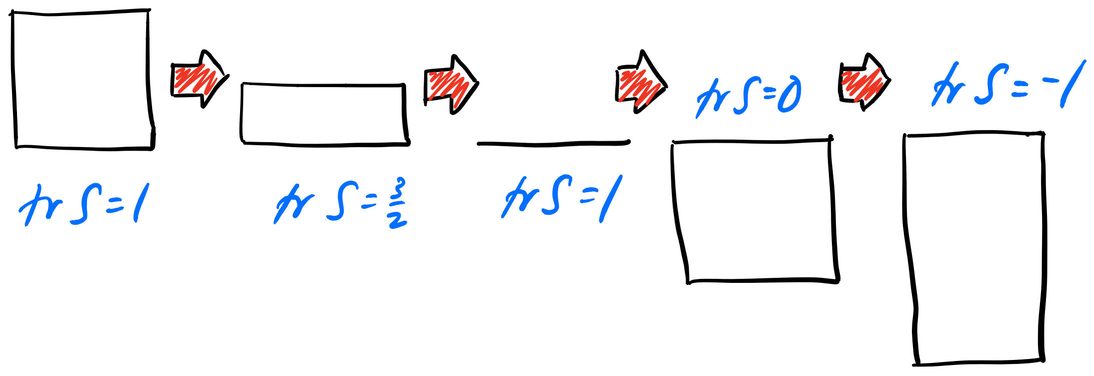


Figure 6.2.: A 2D square is sitting on a table (left). We squish its top edge until it has been smashed into a pancake (center), and then keep pushing, non-physically, into the table. A prolapsed square whose area is equal to the original (center, right) get a score of $\operatorname{tr}\mathbf{S} = 0$, and past that, the score becomes negative.

Apparently this often happens in geometry processing as well, because one of the main strengths of the ARAP model is that it is *also* able to see inversion. If an element has been squished inside out with sufficient violence, the $I_1 = \operatorname{tr}\mathbf{S}$ term that is so important in ARAP also becomes negative (Fig. 6.2). In contrast, the original $I_C = \operatorname{tr}\left(\mathbf{F}^T\mathbf{F}\right)$ invariant squares all of its singular values, so it can never become negative.

From Mooney's perspective, it may have made sense to use the Cauchy-Green invariants because the idea of an inversion was absurdly non-physical, so the math should preclude its very existence. Square everything! Nature abhors a negative invariant! He wasn't performing any numerical computations or animating any movies, so Cauchy-Green was the way to go.[4]

6.2 ARAP (And Others) Don't Do Great

Having poured all this effort into understanding ARAP, how do the final results look when we push them into a solid mechanics simulation? To answer that, we should first establish what we *want* out of a solid mechanics simulation. Our target application is computer animation, and expressively realistic *squashing and stretching* are some of the most important visual features, according to the Disney animators who wrote Johnston and Thomas (1981) and Blair (2003).

Let's look at what we expect under squashing in stretching. In Fig. 6.3, we have a square of bubble gum, held between two metal plates. Under squashing, we expect that stuff will pooch out of the top and bottom, and under stretching, we expect it to form a graceful curve that flattens out to a line along the middle.

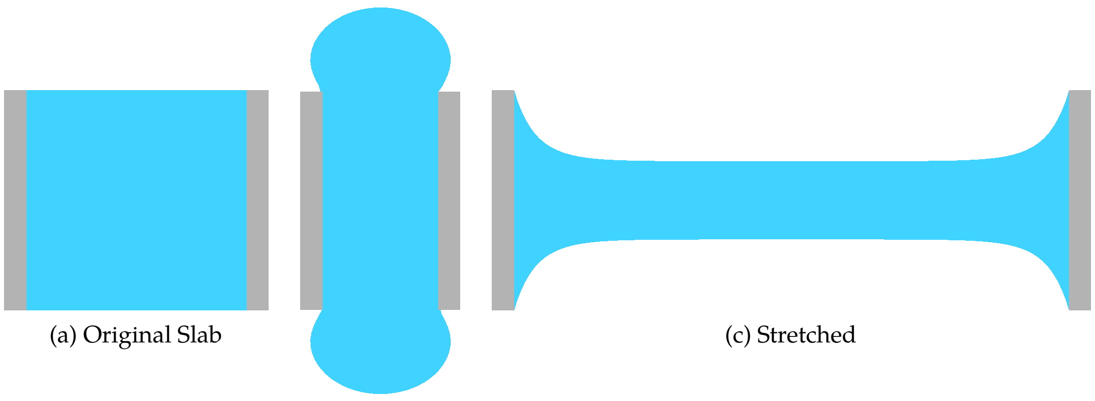

(a) Original Slab (b) Squashed (c) Stretched

Figure 6.3.: What we want from a 2D square of rubber (left). Under squashing, stuff squeezes out the top and bottom (center). Under stretching, it swoops down to a nice clean line along the middle (right).

What happens when we try to subject ARAP to these conditions? The results shown in Fig. 6.4 aren't great. This partially explains why ARAP is popular in *geometry processing*, but less so in simulation. While it's relatively robust[5], it's not very realistic, and never pretended to be. If we look at the energy closely, $\Psi_{\text{ARAP}} = I_2 - 2I_1 - 3$, we see that it

[4]If we want to go further down this rabbit hole, Mooney never actually invokes the Cauchy-Green invariants, the follow-on analysis of Rivlin (1948) does 8 years later in Eqn. 3.4, which further muddies the provenance. I think we've travelled down this hole far enough for the moment though.

[5]Provided you compute the exact rotation gradient!

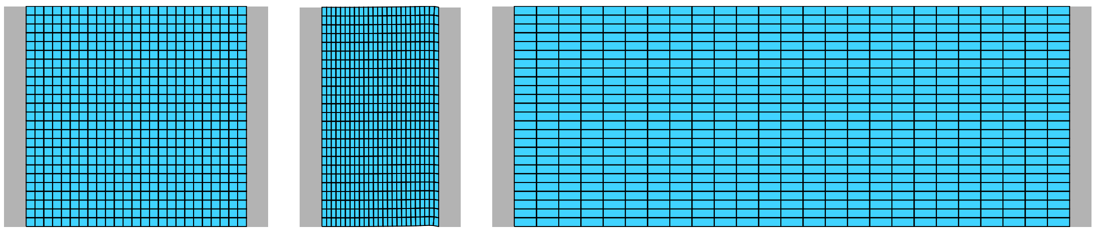

(a) Original Slab (b) Squashed, ARAP (c) Stretched, ARAP

Figure 6.4.: ARAP, under squashing (left) and stretching (right). Nowhere close to Fig. 6.3.

only involves the first and second invariants. However, we saw back in §5.3.2 and Fig. 5.4, that the third invariant is the one that deals with *volume*.

The ARAP energy does not explicitly perform any volume preservation, because it doesn't even bother to compute the volume of each element. Instead, it only cares about preserving relative lengths. That's why Fig. 6.4 looks like a length of chickenwire being stretched and squashed.

6.2.1 A Brief Aside: The Lamé Parameters

The ARAP model only preserves lengths, not volumes. Thus, we didn't need parameters describing how much relative length and volume preservation we wanted. But, we're going to need such parameters in a moment, so let's describe them now.

These are usually specified using the *Lamé parameters*, μ and λ. The μ parameter controls length preservation, and is also sometimes called the *shear modulus* or *Lamé's second parameter*. The λ parameter controls volume preservation, and is sometimes called *Lamé's first parameter*.[6] If you want lots of volume preservation, you set λ to be much larger than μ. Conversely, if you want more length preservation, you push up the value of μ.

Just to make your life more difficult, many simulators, including *Fizt*, don't take in values for μ and λ directly. Instead, they accept an equivalent set of parameters: the Young's modulus E, and Poisson's ratio, ν. The conversion from (E, ν) and (μ, λ) is:

$$\mu = \frac{E}{2(1+\nu)} \tag{6.3}$$

$$\lambda = \frac{E\nu}{(1+\nu)(1-2\nu)}. \tag{6.4}$$

Hold on. When $\nu = \frac{1}{2}$, the value of λ creates a divide-by-zero that blasts off to infinity? Is this a mistake?

[6] I know. The λ sets up a naming collision with the eigenvalues λ_i. I don't make the rules here, but it should be clear from context which one I mean.

It's correct: specifying $\nu = \frac{1}{2}$ means that volume must be preserved at all costs, no matter how much it ruins the stability of the simulation. Biological tissues like muscle and fat are known to have $\nu \approx \frac{1}{2}$ (Greaves et al. (2011)), but they're not exactly $\frac{1}{2}$, so we can still chug ahead.

6.2.2 St. Venant Kirchhoff Doesn't Do Better

With the Lamé preliminaries out of the way, let's look at St. Venant Kirchhoff to see how it compares to ARAP. We're moving past the stretching-only version and finally adding the "volume preservation" term to the right to arrive at the full model:

$$\Psi_{\text{StVK}} = \mu \|\mathbf{E}\|_F^2 + \frac{\lambda}{2} \operatorname{tr}^2(\mathbf{E}). \tag{6.5}$$

Since we care how fleshy our cartoon characters look, lets try this out with $\nu = 0.49$ in Fig. 6.5.[7] First the good news: the stretching looks better than ARAP. At least it bows a little in the middle, even though it doesn't give us the clean line we want along the middle. But what is going on under squashing?! It poofs up slightly along the top and bottom, but then some bizarre overlap and collapse happens along the sides!

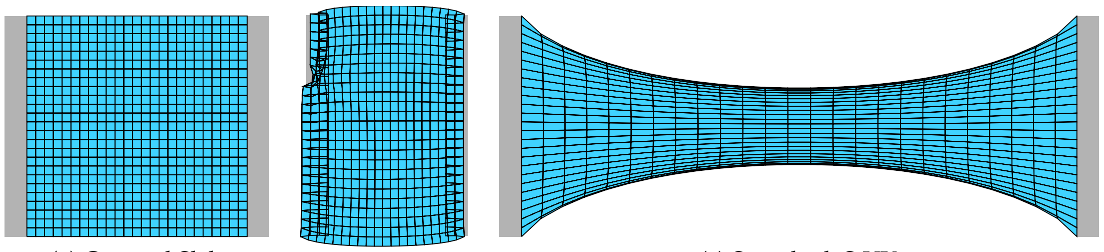

(a) Original Slab (b) Squashed, StVK (c) Stretched, StVK

Figure 6.5.: StVK, under squashing (middle) and stretching (right). What is going on with squashing?!

[7]If you're a stickler, the Lamé conversion formulas should be slightly different for StVK because it's a quartic model. We're going to abandon StVK as a whole in a minute though, so I'm not going to spend time on that.

What we're seeing is that StVK is a card-carrying member of the Cauchy-Green Club:

$$\begin{aligned}
\Psi_{\text{StVK}} &= \mu\|\mathbf{E}\|_F^2 + \frac{\lambda}{2}\operatorname{tr}^2(\mathbf{E}) \\
&= \frac{\mu}{4}\|\mathbf{E}\|_F^2 + \frac{\lambda}{8}\operatorname{tr}^2(\mathbf{F}^T\mathbf{F} - \mathbf{I}) && \text{(Definition of } \mathbf{E}\text{, and pull out the } \tfrac{1}{2}) \\
&= \frac{\mu}{4}\|\mathbf{E}\|_F^2 + \frac{\lambda}{8}\operatorname{tr}(\mathbf{F}^T\mathbf{F} - \mathbf{I})\operatorname{tr}(\mathbf{F}^T\mathbf{F} - \mathbf{I}) && \text{(Make the squaring explicit)} \\
&= \frac{\mu}{4}\|\mathbf{E}\|_F^2 + \frac{\lambda}{8}\left(\operatorname{tr}(\mathbf{F}^T\mathbf{F})^2 - 6\operatorname{tr}(\mathbf{F}^T\mathbf{F}) + 9\right) && \text{(Eqn. B.13 and } \operatorname{tr}\mathbf{I} = 3) \\
&= \frac{\mu}{4}\|\mathbf{E}\|_F^2 + \frac{\lambda}{8}\left(I_{\mathbf{C}}^2 - 6I_{\mathbf{C}} + 9\right) && \text{(Definition of } I_{\mathbf{C}}) \\
&= \frac{\mu}{4}(II_{\mathbf{C}} - 2I_{\mathbf{C}} + 3) + \frac{\lambda}{8}\left(I_{\mathbf{C}}^2 - 6I_{\mathbf{C}} + 9\right). && \text{(From Eqn. 5.10)}
\end{aligned}$$

As we just saw in §6.1.2, the Cauchy-Green invariants don't know about inversion. In fact, they deliberately put on blinders by squaring it away. Thus, the bizarrely collapsed and overlapping elements in Fig. 6.5 are elements that have inverted. StVK *thinks* they're not inverted, due to its Cauchy-Green myopia, so it does not exert any forces that try to correct things.

Thus, StVK makes some progress under stretching, but this behavior under squashing simply won't do.

6.2.3 Co-Rotational Doesn't Do Better

Next up, let's try that co-rotational model that everybody (Müller et al. (2002); Etzmuss et al. (2003); Irving et al. (2004)) simultaneously re-discovered back in the mid-2000s:

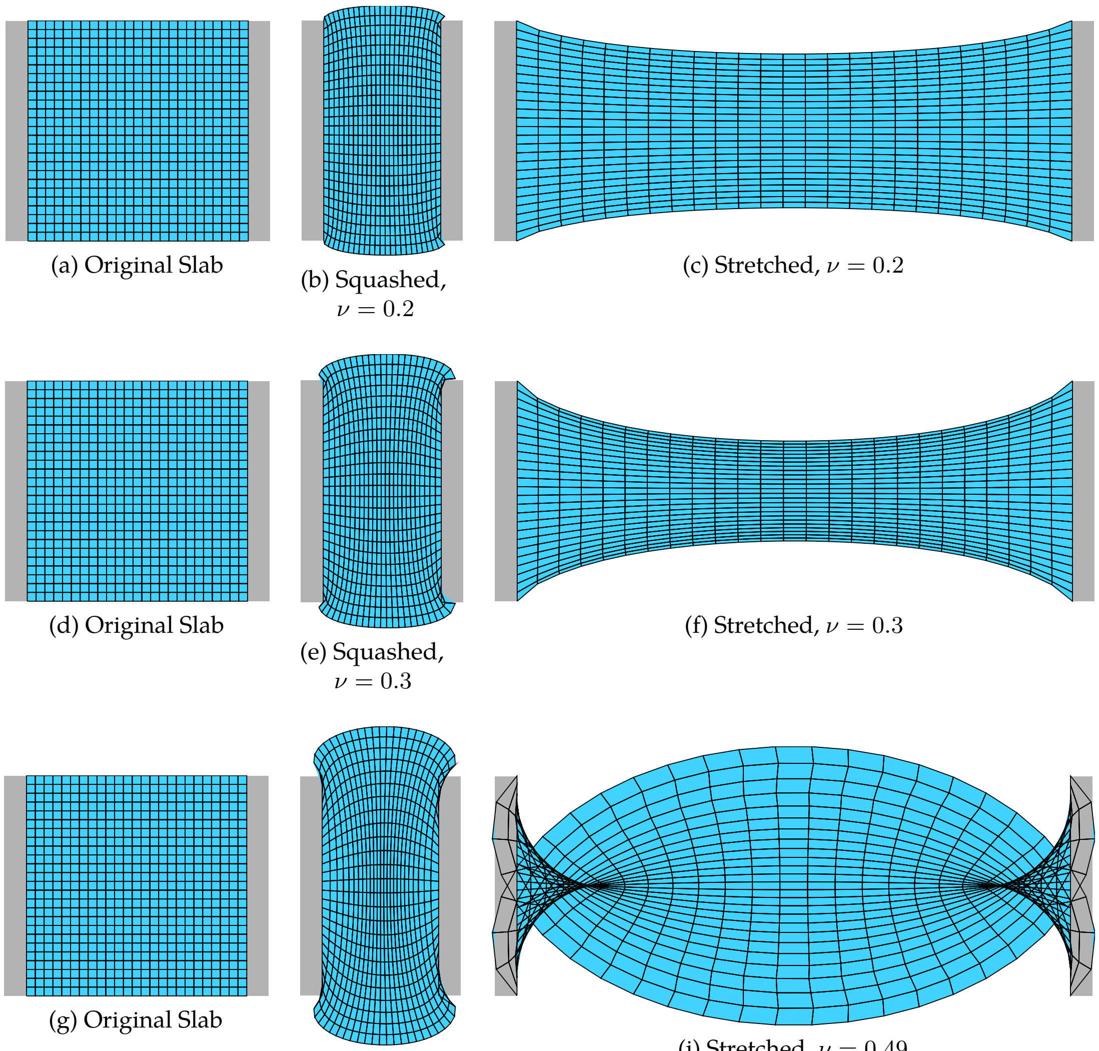
$$\Psi_{\text{CoRot}} = \frac{\mu}{2}\|\mathbf{F} - \mathbf{R}\|_F^2 + \frac{\lambda}{2}\operatorname{tr}^2(\mathbf{S} - \mathbf{I}). \tag{6.6}$$

As you can see in Fig. 6.6, things get really weird. The freakiness around squashing we saw with StVK is gone, but now it's replaced with bigger freakishness under stretching! Why did the slab turn into a prolapsed trampoline!?

Let's unpack that so-called "volume preserving" term. Following the same steps as StVK, we get:

$$\Psi_{\text{CoRot}} = \frac{\mu}{2}\|\mathbf{F} - \mathbf{R}\|_F^2 + \frac{\lambda}{2}\left(I_1^2 - 6I_1 + 9\right). \tag{6.7}$$

What a minute ... where's I_3? What kind of a shady volume term doesn't even compute the volume? A *linearized* volume term, that's what kind. That won't do here.



(a) Original Slab (b) Squashed, $\nu = 0.2$ (c) Stretched, $\nu = 0.2$

(d) Original Slab (e) Squashed, $\nu = 0.3$ (f) Stretched, $\nu = 0.3$

(g) Original Slab (h) Squashed, $\nu = 0.49$ (i) Stretched, $\nu = 0.49$

Figure 6.6.: Co-rotational, with increasing ν. As $\nu \to 1/2$, stretching gets really strange.

6.2.4 Neo-Hookean Is Okay Unless It Burns The House Down

Finally, let's try the Neo-Hookean material, which we can re-write in terms of the Smith et al. (2019) invariants:

$$\begin{aligned}
\Psi_{\text{BW08}} &= \frac{\mu}{2}(\|\mathbf{F}\|_F^2 - 3) - \mu \log(J) + \frac{\lambda}{2}\left(\log(J)\right)^2 \\
&= \frac{\mu}{2}(I_2 - 3) - \mu \log(I_3) + \frac{\lambda}{2}\left(\log(I_3)\right)^2 .
\end{aligned}$$

The I_3 invariant shows up for sure, so at least we're not getting suckered into using some sham volume preservation term that doesn't even know the definition of "volume".

The results of the simulation are in Fig. 6.7. First the good news: under stretching, Neo-Hookean finally gives us that nice, stretched-out bubble gum look that we wanted. Then then bad: under squashing, Neo-Hookean produces a `NaN` that burns down the rest of the simulation.

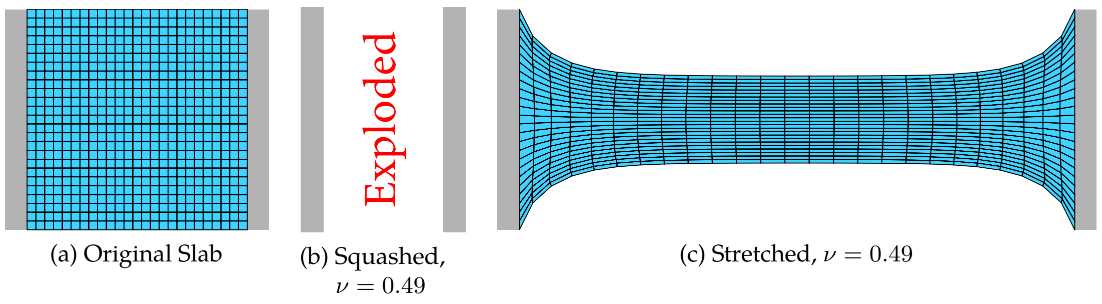


Figure 6.7.: Neo-Hookean material looks great under stretching (right), but produces a `NaN` and explodes under squashing.

The culprit is that $\log(I_3)$ term, since as $I_3 \rightarrow 0$, we get the singularity $\log(I_3) \rightarrow -\infty$. It's actually slightly worse, since there's a squared $\left(\log(J)\right)^2$ term that blasts off even more rapidly towards infinity.

Now our speculative motivations from §6.1.2 about Mooney using the squared invariants rears its ugly head. Under this interpretation, nature abhors a negative invariant, so a $\log(I_3)$ term has been inserted as a barrier function to ensure that we never cross the $I_3 = 0$ threshold. In this world view, we abhor a negative invariant so much that *we give the energy permission to burn down the entire simulation* if it looks like a negative is about to appear.

Computer animation does not ascribe to this view. Again, Wile-E-Coyote-style pancakes happen *all the time*. They don't persist very long; after the initial impact, the Road Runner's anvil usually falls away, and the coyote blows into his thumb to re-inflate himself. A good energy in this scenario should allow pancakes and inversions to occur, but once the anvil forces are removed, sproing the coyote back into his original shape.

6.3 A Better Neo-Hookean Energy?

6.3.1 So Many Neo-Hookeans

We looked at one Neo-Hookean energy in §6.2.4, but there are a whole lot of energies that call themselves "Neo-Hookean". We even saw one way back in Eqn. 2.6 which was just the Dirichlet energy in disguise. If Fig. 6.7, we saw that we can get good-looking stretching shapes out of Neo-Hookean, but it explodes under squashing. Maybe one of these other energies that calls itself "Neo-Hookean" can do better?

Here's a few I found in the literature:

$$\Psi_{\text{A}} = \frac{\mu}{2}(I_2 - 3) - \mu \log I_3 + \frac{\lambda}{2}(I_3 - 1)^2 \qquad \text{(Ogden (1997))}$$

$$\Psi_{\text{B}} = \frac{\mu}{2}\left(\frac{I_2}{I_3^{\frac{2}{3}}} - 3\right) + \frac{\lambda}{2}(I_3 - 1)^2 \qquad \text{(Bower (2009))}$$

$$\Psi_{\text{C}} = \frac{\mu}{2}\left(\frac{I_2}{I_3^{\frac{2}{3}}} - 3\right) + \frac{\lambda}{2}(I_3 - 1). \qquad \text{(Wang and Yang (2016))}$$

We saw before that any $\log(I_3)$ will create an infinity under pancake states, so Ψ_{A} is already out of the running. The Ψ_{B} and Ψ_{C} energies don't contain a $\log(I_3)$, but they instead have $\frac{I_2}{I_3^{\frac{2}{3}}}$ terms. These terms have the same problem: as $I_3 \rightarrow 0$, the $\frac{I_2}{I_3^{\frac{2}{3}}}$ is standing there with a match and a can of gasoline, ready to burn everything down with an ∞. That's not helpful either.

6.3.2 Let's Mix-And-Match Our Own

Still, if we look at Ψ_{A}, Ψ_{B}, and Ψ_{C}, we can see some glimmers of hope. That $\frac{\mu}{2}(I_2 - 3)$ in front of Ψ_{A}, which also appears in front of Ψ_{BW08}, is a well-behaved Dirichlet-like term. It doesn't give us the behavior we want, but at least it will never generate a simulation-destroying singularity.

All three of the energies also incorporate a $\frac{\lambda}{2}(I_3 - 1)^2$ term. This is well-behaved as well! It can never create a divide-by-zero error, and unlike the co-rotational model, it uses the 100% certified, honest-to-goodness volume-measuring, I_3 invariant. Maybe we can just mix-and-match these two components to get our own Neo-Hookean flavor:

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

This looks quite reasonable, doesn't it? Until you try to simulate it. In Fig. 6.8, we see what happens at the first step of the simulation, even when *no forces have been applied*. The mesh shrinks! The artifact becomes less pronounced as $\nu \rightarrow \frac{1}{2}$, but is still clearly visible. Other biomechanics works have observed similar phenomena (e.g. Blemker et al. (2005)), though not in the context of authoring an inversion-friendly Neo-Hookean energy. The solution is usually to set $\lambda \approx 1000\mu$, which corresponds to ignoring the artifact by always keeping ν really close to $\frac{1}{2}$. Let's try to do better.

6.3.3 A Stable Neo-Hookean Energy

Why does the mesh shrink to begin with? First, it has a Dirichlet-like term, $\frac{\mu}{2}(I_2 - 3)$, which we first saw in §2.3.4 when it tried to collapse our entire mesh down to a point.

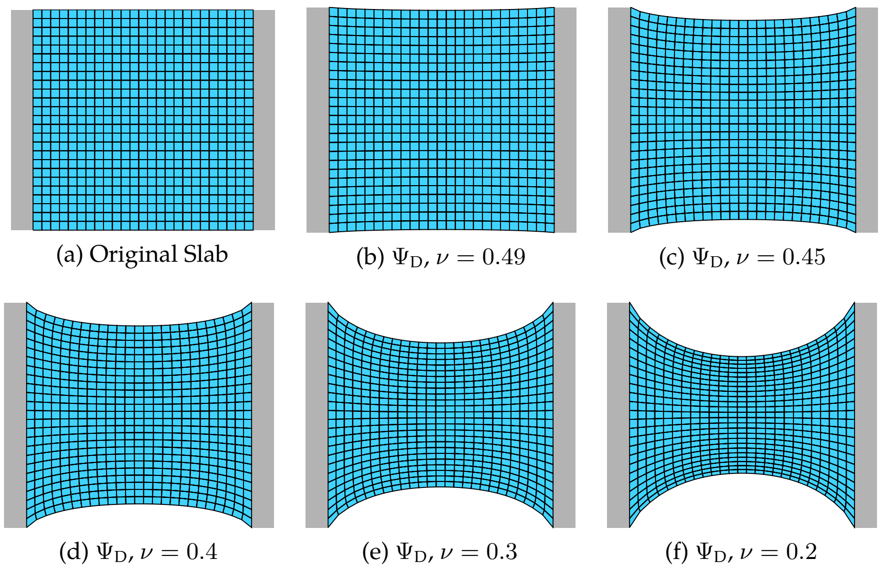

(a) Original Slab (b) Ψ_{D}, $\nu = 0.49$ (c) Ψ_{D}, $\nu = 0.45$ (d) Ψ_{D}, $\nu = 0.4$ (e) Ψ_{D}, $\nu = 0.3$ (f) Ψ_{D}, $\nu = 0.2$

Figure 6.8.: Ψ_{D}, our first attempt at a mix-and-match Neo-Hookean energy, freakishly causes the mesh to shrink, even if no forces are being applied. As ν decreases, the problem gets worse, but it visibly persists even as $\nu \approx \frac{1}{2}$.

Second, it has a volume preservation term, $\frac{\lambda}{2}(I_3 - 1)^2$, which tries to keep I_3 as close to the original volume (which corresponds to 1) as possible. Even in the absence of any external forces, these two terms are pulling in opposition to each other, and the true "rest-state" that a mesh will have is the middle ground where these two forces cancel out.

This cancellation point will *not* be when $I_3 = 1$. It will instead be at some compromise point with the dueling Dirichlet force, which wants to pull everything down into the zero-volume $I_2 = 0$ state. In this duel, if we increase λ, then $\frac{\lambda}{2}(I_3 - 1)^2$ gains the upper hand and the artifact become smaller.

Here's an idea: why don't we give the $\frac{\lambda}{2}(I_3 - 1)^2$ upper hand by making the element fatter? Instead of trying to get back to the $I_3 = 1$ state, why don't we tell the energy to return to some $I_3 = \alpha$ state, where $\alpha > 1$? Then when it has to compromise with the Dirichlet term, the slightly shrunken version that they balance out to will be the $I_3 = 1$ state we originally wanted?

We can write this out in more detail by taking the PK1 of Ψ_{D}:

$$\begin{aligned}\frac{\partial \Psi_{\text{D}}}{\partial \mathbf{F}} &= \mu \mathbf{F} - \lambda(I_3 - 1)\frac{\partial I_3}{\partial \mathbf{F}} \\ &= \mu \mathbf{F} - \lambda(I_3 - 1)\begin{bmatrix} f_{11} & -f_{10} \\ -f_{01} & f_{00} \end{bmatrix}. \qquad \text{(In 2D)}\end{aligned}$$

If everything was working as it should the PK1 would be all zeros under zero deformation,[8] $\mathbf{F} = \mathbf{I}$, but we see that it doesn't work out that way:

$$\begin{aligned}\frac{\partial \Psi_{\mathrm{D}}(\mathbf{I})}{\partial \mathbf{F}} &= \mu \mathbf{I} - \lambda(1-1)\mathbf{I} \\ &= \mu \mathbf{I}.\end{aligned}$$

That's not great, but we knew it was going to happen. Now let's introduce our α factor, and see how much chubbier we need to make each element:

$$\begin{aligned}\frac{\partial \Psi_\alpha}{\partial \mathbf{F}} &= \mu \mathbf{F} - \lambda(I_3 - \alpha)\frac{\partial I_3}{\partial \mathbf{F}} \\ &= \mu \mathbf{F} - \lambda(I_3 - \alpha)\begin{bmatrix} f_{11} & -f_{10} \\ -f_{01} & f_{00}\end{bmatrix} \\ \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} &= \mu \mathbf{I} - \lambda(1-\alpha)\mathbf{I}. && \text{(Plugging in } \mathbf{F} = \mathbf{I}, \tfrac{\partial \Psi_\alpha}{\partial \mathbf{F}} = 0)\end{aligned}$$

The last line encodes the constraint $0 = \mu - \lambda(1-\alpha)$. Solving for α yields:

$$\alpha = 1 - \frac{\mu}{\lambda}. \tag{6.8}$$

We have not seen the $\frac{\partial \Psi}{\partial \mathbf{F}} = 0$ explicitly named anywhere, so in Smith et al. (2019) we decided to call it the "rest-stability" criterion. Our rest-stable Neo-Hookean energy is then:

$$\begin{aligned}\Psi_{\mathrm{SNH}} &= \frac{\mu}{2}(I_2 - 3) + \frac{\lambda}{2}(I_3 - \alpha)^2 \\ &= \frac{\mu}{2}(I_2 - 3) + \frac{\lambda}{2}\left(I_3 - 1 - \frac{\mu}{\lambda}\right)^2.\end{aligned}$$

Expanding the square, we get:

$$\Psi_{\mathrm{SNH}} = \frac{\mu}{2}(I_2 - 3) - \mu(I_3 - 1) + \frac{\lambda}{2}(I_3 - 1)^2 + \left(\frac{\mu}{\lambda}\right)^2.$$

Constant factors only translates the overall energy and get burned off under differentiation, so this is equivalent to:

$$\Psi_{\mathrm{SNH}} = \frac{\mu}{2}(I_2 - 3) - \mu(I_3 - 1) + \frac{\lambda}{2}(I_3 - 1)^2. \tag{6.9}$$

This is our final Stable Neo-Hookean (SNH) energy.[9]

[8] If no deformation is happening, the element shouldn't need to push back on anything, right? If the PK1 is all zeros, then the force will be all zeros after $\frac{\partial \mathbf{F}}{\partial \mathbf{x}}$ gets applied.

[9] The Smith et al. (2019) paper also adds an *origin barrier* term that further complicates things. That version is *not the one* that is used internally at Pixar. The Ψ_{SNH} energy is. The origin barrier was added because the paper's referees insisted that a spurious but extremely-difficult-to-reach root at $\mathbf{F} = 0$ would destabilize the energy. Extensive production experience has shown that it does not.

6.4 A Bunch of Other Stable Energies

We can apply this approach more generally. Just to drive you crazy, a bunch of other energies are usually written down without any volume preservation term. The assumption is that you will somehow add your own. For example, the two-term Mooney Rivlin model can be written:

$$\Psi_{\text{MR}} = \mu_0(I_{\mathbf{C}} - 3) + \frac{\mu_1}{2}(I_{\mathbf{C}}^2 - II_{\mathbf{C}} - 3).$$

If you try to simulate this, you get the collapsed mesh on the left of Fig. 6.9. Clearly, a volume term is needed. We could be stubborn, insist that we learned nothing from the previous section, and just patch on a $(I_3 - 1)^2$ to obtain:

$$\Psi_{\text{MR+}} = \mu_0(I_{\mathbf{C}} - 3) + \frac{\mu_1}{2}(I_{\mathbf{C}}^2 - II_{\mathbf{C}} - 3) + \frac{\lambda}{2}(I_3 - 1)^2.$$

Predictably, we then encounter the same mesh contraction problems, as seen on the right side of Fig. 6.9. We can do better.

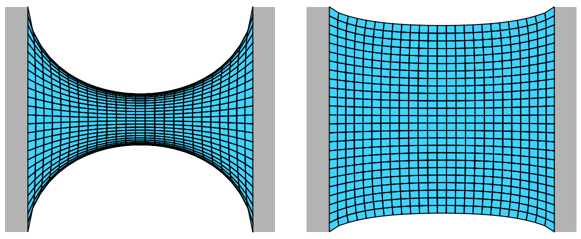

(a) Ψ_{MR}, no volume term (b) $\Psi_{\text{MR+}}$, $\nu = 0.45$

Figure 6.9.: Mooney-Rivlin, without a volume term (left), and with a volume term, but without rest stability (right). Same contraction problems as before: the mesh shrinks, even though no forces are being applied.

6.4.1 Stable Mooney-Rivlin

I'm being slightly abusive here and mixing the Cauchy-Green and Smith et al. (2019) invariants. But, as we saw in §5.3.3.2, the $II_{\mathbf{C}}$ term gets a lot bigger when written in terms of the new invariants. Here's the Stable Mooney-Rivlin energy,

$$\Psi_{\text{SMR}} = \mu_0(I_{\mathbf{C}} - 3) + \frac{\mu_1}{2}(I_{\mathbf{C}}^2 - II_{\mathbf{C}} - 3) + \frac{\lambda}{2}(I_3 - \alpha)^2,$$

and the task is now to solve for α in the PK1 when $\mathbf{F} = \mathbf{I}$. The PK1 is:

$$\frac{\partial \Psi_{\text{SMR}}}{\partial \mathbf{F}} = 2\mu_0\mathbf{F} + \mu_1 I_{\mathbf{C}}\mathbf{F} - 2\mu_1\mathbf{F}\mathbf{F}^T\mathbf{F} + \lambda(I_3 - \alpha)\begin{bmatrix} f_{11} & -f_{10} \\ -f_{01} & f_{00} \end{bmatrix}.$$

Pushing through, the alpha works out to:

$$\begin{aligned}
0 &= 2\mu_0 \mathbf{I} + 3\mu_1 \mathbf{I} - 2\mu_1 \mathbf{I} + \lambda(1-\alpha)\mathbf{I} \\
&2\mu_0 \mathbf{I} + \mu_1 \mathbf{I} + \lambda(1-\alpha)\mathbf{I} \\
\alpha &= \frac{2\mu_0 + \mu_1}{\lambda} + 1
\end{aligned}$$

The final Stable Mooney-Rivlin energy takes on a form similar to Stable Neo-Hookean:

$$\begin{aligned}
\Psi_{\text{SMR}} &= \mu_0(I_{\mathbf{C}} - 3) + \frac{\mu_1}{2}(I_{\mathbf{C}}^2 - II_{\mathbf{C}} - 3) + \frac{\lambda}{2}\left(I_3 - \frac{2\mu_0 + \mu_1}{\lambda} - 1\right)^2 && (6.10) \\
&= \mu_0(I_{\mathbf{C}} - 3) + \frac{\mu_1}{2}(I_{\mathbf{C}}^2 - II_{\mathbf{C}} - 3) + \frac{\lambda}{2}(I_3 - 1)^2 - (2\mu_0 + \mu_1)(I_3 - 1). && (6.11)
\end{aligned}$$

The new terms are shown in red.

6.4.2 Stable Arruda-Boyce

Let's do it again for the energy of Arruda and Boyce (1993). A three-term version of this is:

$$\Psi_{\text{AB}} = \frac{\mu}{2}(I_2 - 3) + \frac{\mu\beta_1}{4}(I_2^2 - 9) + \frac{\mu\beta_2}{6}(I_2^3 - 27).$$

Our stable version is,

$$\Psi_{\text{SAB}} = \frac{\mu}{2}(I_2 - 3) + \frac{\mu\beta_1}{4}(I_2^2 - 9) + \frac{\mu\beta_2}{6}(I_2^3 - 27) + \frac{\lambda}{2}(I_3 - \alpha)^2,$$

and after obtaining $\alpha = 1 + \frac{\mu}{\lambda}(1 + 3\beta_1 + 9\beta_2)$, the final version is

$$\begin{aligned}
\Psi_{\text{SAB}} &= \frac{\mu}{2}(I_2 - 3) + \frac{\mu\beta_1}{4}(I_2^2 - 9) + \frac{\mu\beta_2}{6}(I_2^3 - 27) + \frac{\lambda}{2}\left(I_3 - 1 - \frac{\mu}{\lambda}(1 + 3\beta_1 + 9\beta_2)\right)^2 && (6.12) \\
&= \frac{\mu}{2}(I_2 - 3) + \frac{\mu\beta_1}{4}(I_2^2 - 9) + \frac{\mu\beta_2}{6}(I_2^3 - 27) + \frac{\lambda}{2}(I_3 - 1)^2 - \mu(1 + 3\beta_1 + 9\beta_2)(I_3 - 1). && (6.13)
\end{aligned}$$

Again, the new terms are in red.

6.4.3 Stable Fung Hardening

Finally, let's look at the hardening model of Fung (2013). This energy usually doesn't appear all by itself, but is added so that a material becomes dramatically stiffer (hardens) under large stretching. Here we append it to our Stable Neo-Hookean model,

$$\Psi_{\text{SNH+Fung}} = \frac{\mu_0}{2}(I_2 - 3) + \frac{\lambda}{2}(I_3 - \alpha)^2 + \frac{\gamma}{2}\left(e^{\frac{\mu_1}{2}(I_2 - 3)} - 1\right),$$

and after solving for $\alpha = 1 + \frac{\mu_0 + \gamma\mu_1}{\lambda}$, we obtain

$$\Psi_{\text{SNH+Fung}} = \frac{\mu_0}{2}(I_2 - 3) + \frac{\lambda}{2}\left(I_3 - 1 - \frac{\mu_0 + \gamma\mu_1}{\lambda}\right)^2 + \frac{\gamma}{2}\left(e^{\frac{\mu_1}{2}(I_2-3)} - 1\right) \tag{6.14}$$

$$= \frac{\mu_0}{2}(I_2 - 3) + \frac{\lambda}{2}(I_3 - 1)^2 + \frac{\gamma}{2}\left(e^{\frac{\mu_1}{2}(I_2-3)} - 1\right) - (\mu_0 + \gamma\mu_1)(I_3 - 1). \tag{6.15}$$

In general: When obtaining the rest stabilization term α, the expressions tend to take the form $\alpha = 1 + \frac{\beta}{\lambda}$. When inserted into $(I_3 - \alpha)^2$, these then separate (up to constant) into the original $\frac{\lambda}{2}(I_3 - 1)^2$ volume term, and an energy-specific stabilization term $-\beta(I_3 - 1)$.

Go forth and design your own stable energy!

Chapter 7

The Analytic Eigensystems of Isotropic Energies

This chapter is a loose retelling of §4 from Smith et al. (2018) and §4 and §5 in Smith et al. (2019). If you want to cut to the chase, Matlab code that implements the final algorithm is given in Fig. 7.7. Plug your favorite isotropic energy into line 11, and it will spit out the analytic eigenvalues.

7.1 Keeping Everything Semi-Positive-Definite

The core of *Fizt* and Baraff and Witkin (1998)-style integration is the use of a preconditioned conjugate gradients (PCG) solver. PCG requires that the underlying matrix $\mathbf{A}$ is semi-positive-definite, i.e. that all of its eigenvalues are greater than or equal to zero. This is usually written somewhat obtusely in linear algebra textbooks as:

$$\mathbf{x}^T\mathbf{A}\mathbf{x} \geq 0 \qquad \forall \mathbf{x}.$$

Another way to look at this is through the product $\mathbf{A}\mathbf{x} = \mathbf{y}$. Positive-definiteness implies that the *sign* of each entry in $\mathbf{y}$ 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. If it does, when we compute the dot product $\mathbf{x}^T\mathbf{y}$, the negatives of $\mathbf{x}$ and $\mathbf{y}$ will not mutually annihilate, and you can end up with a result less than zero.

We care about this property because if the system *isn't* semi-positive-definite, it implies the existence of multiple, physically valid, energetically equivalent solutions. In that case, the optimization will not know which solution to pick, and can oscillate indecisively between them forever, or explode spectacularly to infinity. This ambiguity *can and does* happen in the physical systems we are examining, but let's put that fact aside for the moment. In the moment, we want to be able to hand to our existing *Fizt* solver a semi-positive-definite system. How can we know for sure that all the Hessians that came out of all of our $\frac{\partial \Psi}{\partial \mathbf{F}}$ and $\frac{\partial \Psi}{\partial \mathbf{x}}$ manipulations produced all-positive eigenvalues?

One way would be to compute the eigendecomposition of the global system matrix you assembled using the $\frac{\partial^2 \Psi}{\partial \mathbf{F}^2}$ from every element in your simulation (Nocedal and Wright (2006)). If any of those eigenvalues are less than zero, you can perform a *projection* to snap those back to zero, reassemble a modified matrix, and then perform PCG using this modified matrix. We would prefer not to do this, because PCG runs in roughly $O(N^{\frac{3}{2}})$ time, but an eigendecomposition of the global matrix takes $O(N^3)$. The cure is worse than the disease. We can instead try a *per-element* projection in the style of Teran et al. (2005), where we compute a eigendecomposition for each element's $\frac{\partial^2 \Psi}{\partial \mathbf{F}^2}$, which only involves taking the eigendecomposition of a bunch of 9×9 matrices, not the global $N \times N$ matrix. The sum of semi-positive-definite matrices is known to also be semi-positive definite, so the strategy is sound.

7.2 Can ARAP Go Indefinite?

Maybe we're worrying about nothing, and the energies we're interested in can't even go indefinite. To see if this is true, let's look at the ARAP energy.

Fortunately, the slick representation we found for the rotation gradient $\frac{\partial \mathbf{R}}{\partial \mathbf{F}}$ already gives analytic expressions for its own eigendecomposition. From there, it's only a short skip and jump to obtain the analytic eigenvalues of the entire ARAP energy. From §5.5.2, the flattened Hessian for ARAP is:

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

We know what the eigendecomposition of $2\mathbf{I}_{9\times9}$ is. All the eigenvalues are $\lambda_{0\ldots8} = 2$, and since the eigenvalues are all the same, the eigenvectors are not unique. Any basis than spans a rank-9 subspace will do.

We *also* know what the eigensystem of $\mathbf{H}_1$ from §5.4.4. Here they are again:

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

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

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

$$\lambda_{3\ldots8} = 0 \qquad \mathbf{Q}_{3\ldots8} = \text{subspace orthogonal to } \mathbf{Q}_{0,1,2}.$$

Multiplying a matrix by a constant only scales its eigenvalues, so $-2\mathbf{H}_1$ has the eigenvalues

$$\lambda_0 = \frac{-4}{\sigma_x + \sigma_y} \qquad \lambda_1 = \frac{-4}{\sigma_y + \sigma_z} \qquad \lambda_2 = \frac{-4}{\sigma_x + \sigma_z} \qquad \lambda_{3\dots 8} = 0, \tag{7.1}$$

and the eigenvectors remain the same.

So, what's the eigensystem of their sum, $\mathbf{I}_{9\times 9} - 2\mathbf{H}_1$? 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 $\mathbf{I}_{9\times 9}$! 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:

$$\lambda_0 = 2 - \frac{4}{\sigma_x + \sigma_y} \qquad \mathbf{Q}_0 = \frac{1}{\sqrt{2}}\mathbf{U}\begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}\mathbf{V}^T \tag{7.2}$$

$$\lambda_1 = 2 - \frac{4}{\sigma_y + \sigma_z} \qquad \mathbf{Q}_1 = \frac{1}{\sqrt{2}}\mathbf{U}\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & -1 & 0 \end{bmatrix}\mathbf{V}^T \tag{7.3}$$

$$\lambda_2 = 2 - \frac{4}{\sigma_x + \sigma_z} \qquad \mathbf{Q}_2 = \frac{1}{\sqrt{2}}\mathbf{U}\begin{bmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \\ -1 & 0 & 0 \end{bmatrix}\mathbf{V}^T \tag{7.4}$$

$$\lambda_{3\dots 8} = 2 \qquad \mathbf{Q}_{3\dots 8} = \text{subspace orthogonal to } \mathbf{Q}_{0,1,2}. \tag{7.5}$$

First of all, who knew that ARAP's eigendecomposition had such crisp expressions? I'm used to seeing the Hessian as a chaotic churn of numbers, and taking its eigendecomposition just plunges you further into the primordial soup. But, this crystalline eigenstructure was sitting there at the bottom this whole time! A pleasant surprise.

Second, and returning to the task at hand, can the energy go indefinite? Certainly $\lambda_{3\dots 8}$ are all safely positive for all time, but when do $\lambda_{0,1,2}$ go negative? Let's look at λ_0 as an example:

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

Similar expressions appear for the other two:

$$\lambda_1 \leq 0 \iff \sigma_y + \sigma_z \leq 2 \tag{7.6}$$

$$\lambda_2 \leq 0 \iff \sigma_x + \sigma_z \leq 2. \tag{7.7}$$

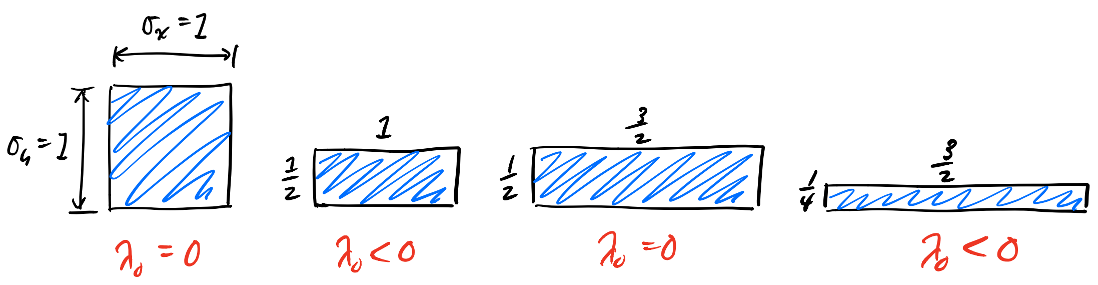


Figure 7.1.: **From left to right:** An undeformed element is already semi-definite. Squashing along y makes it indefinite. Stretching enough in x makes it semi-definite again. Squashing y even further pushes us back into indefiniteness.

Uh oh. Don't these conditions get tripped all the time? Let's put a single element through its paces in Fig. 7.1. If an element is undergoing zero deformation, $\sigma_x = \sigma_y = 1$, it is already flirting with disaster, as $\lambda_0 = 0$. If any of these directions gets squashed, e.g. $\lambda_0 = \frac{1}{2}$ while $\lambda_1 = 1$, then this eigenvalue will be negative. The other direction could pick up the slack, i.e. $\lambda_0 = \frac{1}{2}$ while $\lambda_1 = \frac{3}{2}$, but if any direction is *more squashed than the other direction is stretched*, then the eigenvalue will become negative. The only situation where λ_0 is safely positive is if both directions are being stretched.

The appearance of negative eigenvalues under squashing is well-known in mechanical engineering, as it corresponds to the onset of *buckling*, as shown in Fig. 7.2. If you squash a bar of material, will it buckle up or down? From a deformation energy standpoint, both configurations will give the *exact same score* for Ψ_{ARAP}. Therefore, we are currently stuck between multiple, equally enticing solutions, which corresponds to a saddle point in the energy. Saddle points possess both positive and negative curvature, a.k.a. indefinite Hessians, and so the negative eigenvalues make sense.

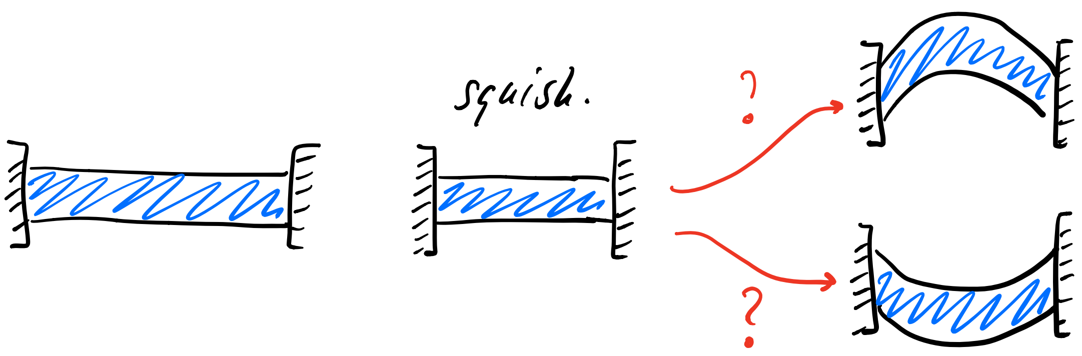


Figure 7.2.: **From left to right:** An undeformed bar. We squash it. Which direction should it buckle in? Up or down? As far as the deformation energy goes, both have exactly the same score!

So yes: ARAP can go indefinite. However, we now also know its eigensystem in closed form, so we can devise a *projected* Hessian quite easily. If we detect that any of the eigenvalues are less than zero, we can efficiently snap it back to zero. It works out to a few extra lines on top of `ARAP_Hessian` from Fig. 5.10, which we'll call `ARAP_Hessian_Filtered` and give the code for in Fig. 7.3.

We *didn't* have to numerically compute a 9×9 eigendecomposition anywhere, which is both faster and cleaner. Additionally, we got better intuition of when and why elements go indefinite, and some fast, compact code for projected Hessians. Wouldn't it be nice if we could do this for every energy?

7.3 The Eigendecompositions of Arbitrary Energies

An analytic decomposition is not only available for the ARAP energy, it can be obtained for *any* isotropic energy. It will take a little bit of work to find a generic expression, but once we do, I'll bake everything into a Matlab script for you (Fig. 7.5)

7.3.1 The General Eigensystem of I_3

We're actually $^2\!/_3$ of the way to a generic approach already. We have the analytic eigensystem for $\frac{\partial^2 I_1}{\partial \mathbf{F}^2}$, which was given in §5.4.4 and §7.2. We also have the eigensystem of $\frac{\partial^2 I_2}{\partial \mathbf{F}^2}$, which was easy to get, because it's a diagonal matrix.

If we can get the analytic eigensystem for $\frac{\partial^2 I_3}{\partial \mathbf{F}^2}$, then maybe, just maybe, we can combine the eigendecompositions for all three in some way to arrive at a generic approach. If you're an analysis rockstar, you can probably derive the eigensystem by taking the variational derivative of I_3 or using some related principle.

I am not an analysis rockstar, so this is not what I did. Just like in §5.4.2, I messed around in Matlab with some easy cases where $\mathbf{U} = \mathbf{V} = \mathbf{I}$ and Σ was loaded up with easy integers, until I found something that looked like a pattern, and then verified their correctness symbolically using Mathematica. Not the most straight-line route to finding analytic expressions, but there's more than one way to skin a cat.

```
function [H] = ARAP_Hessian_Filtered(F)
  [U Sigma V] = svd_rv(F);

  % get the twist modes
  T0 = [0 -1 0; 1 0 0; 0 0 0];
  T0 = (1 / sqrt(2)) * U * T0 * V';

  T1 = [0 0 0; 0 0 1; 0 -1 0];
  T1 = (1 / sqrt(2)) * U * T1 * V';

  T2 = [0 0 1; 0 0 0; -1 0 0];
  T2 = (1 / sqrt(2)) * U * T2 * V';

  % get the flattened versions
  t0 = vec(T0);
  t1 = vec(T1);
  t2 = vec(T2);

  % get the singular values in an order that is consistent
  % with the numbering in the paper
  s0 = Sigma(1,1);
  s1 = Sigma(2,2);
  s2 = Sigma(3,3);

  % build the filter the non-trivial eigenvalues
  lambda0 = 2 / (s0 + s1);
  lambda1 = 2 / (s1 + s2);
  lambda2 = 2 / (s0 + s2);
  if (s0 + s1 < 2)
    lambda0 = 1;
  end
  if (s1 + s2 < 2)
    lambda1 = 1;
  end
  if (s0 + s2 < 2)
    lambda2 = 1;
  end

  H = eye(9,9);
  H = H - lambda0 * (t0 * t0');
  H = H - lambda1 * (t1 * t1');
  H = H - lambda2 * (t2 * t2');

  H = 2 * H;
end
```

Figure 7.3.: Matlab code to compute the projected Hessian of the ARAP energy in 3D. Any eigenvalue that is less than zero is snapped back to zero. The `lambda` terms are not set to zero directly, because they still need to cancel off the entries of `eye(9,9)` when they are added to `H`.

Weirdly enough, three of the eigenvectors of $\frac{\partial^2 I_3}{\partial \mathbf{F}^2}$ are *exactly the same* as $\frac{\partial^2 I_1}{\partial \mathbf{F}^2}$,

$$\lambda_3 = \sigma_z \qquad \mathbf{Q}_3 = \frac{1}{\sqrt{2}}\mathbf{U}\begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}\mathbf{V}^T$$

$$\lambda_4 = \sigma_x \qquad \mathbf{Q}_4 = \frac{1}{\sqrt{2}}\mathbf{U}\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & -1 & 0 \end{bmatrix}\mathbf{V}^T$$

$$\lambda_5 = \sigma_y \qquad \mathbf{Q}_5 = \frac{1}{\sqrt{2}}\mathbf{U}\begin{bmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \\ -1 & 0 & 0 \end{bmatrix}\mathbf{V}^T,$$

and the eigenvalues are even simpler: just the singular values $\sigma_{\{x,y,z\}}$. Just like we saw with Eqn. 5.40, these correspond to infinitesimal rotations, so we call these the *twist* matrices. The next three eigenpairs are so similar to the first three that you're going to think I typed them in wrong:

$$\lambda_6 = -\sigma_z \qquad \mathbf{Q}_6 = \frac{1}{\sqrt{2}}\mathbf{U}\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}\mathbf{V}^T$$

$$\lambda_7 = -\sigma_x \qquad \mathbf{Q}_7 = \frac{1}{\sqrt{2}}\mathbf{U}\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}\mathbf{V}^T$$

$$\lambda_8 = -\sigma_y \qquad \mathbf{Q}_8 = \frac{1}{\sqrt{2}}\mathbf{U}\begin{bmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix}\mathbf{V}^T.$$

They're the same as the twist matrices, except the eigenmatrices are missing a -1, and the eigenvalues are negated. Instead of a rotation, these encode both a rotation and a reflection, so we call these the *flip* matrices.

Since $\operatorname{vec}\left(\frac{\partial^2 I_3}{\partial \mathbf{F}^2}\right) \in \Re^{9\times 9}$, the Hessian contains a total of nine eigenpairs. We've already found six, so what are the other three? So far, all the eigenmatrices we've seen contain 1s and -1s somewhere along the off-diagonal. For last three eigenmatrices, it would make sense if they corresponded to the on-diagonal scaling modes:

$$\mathbf{D}_0 = \mathbf{U}\begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}\mathbf{V}^T \quad \mathbf{D}_1 = \mathbf{U}\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}\mathbf{V}^T \quad \mathbf{D}_2 = \mathbf{U}\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}\mathbf{V}^T. \tag{7.8}$$

As we will see, this happens *sometimes*, but not in general. The general expressions for the last three eigenpairs are a little messier-looking than the other six. The last three eigenvalues are the roots of the depressed cubic equation,

$$\lambda_i = 2\sqrt{\frac{I_2}{3}} \cos\left[\frac{1}{3}\left(\arccos\left(\frac{3I_3}{I_2}\sqrt{\frac{3}{I_2}}\right) + 2\pi(i-1)\right)\right], \tag{7.9}$$

where $i \in \{0, 1, 2\}$. The eigenvectors are then a *linear combination* of the scaling modes,

$$\mathbf{Q}_i = \sum_{j=0}^{2} z_j \mathbf{D}_j \quad \text{where} \quad \begin{cases} z_0 = \sigma_x\sigma_z + \sigma_y\lambda_i \\ z_1 = \sigma_y\sigma_z + \sigma_x\lambda_i \\ z_2 = \lambda_i^2 - \sigma_z^2 \end{cases},$$

and the exact weights in the combination varies based on the current deformation.[1]

7.3.2 All Isotropic Energies Have the Exact Same Eigenvectors

The nine eigenmatrices that we have just seen arise from I_3 are in fact the eigenvectors of *all isotropic energies*. No matter what isotropic energy you write down, these will always be the eigenvectors. That primordial soup we saw when taking the numerical eigendecomposition of $\Re^{9\times9}$ Hessians? It wasn't a soup after all, it was a stack of the same nine Lego blocks all along.

Given their importance, let's list them again here, once and for all:

$$\mathbf{Q}_{i\in\{0,1,2\}} = \sum_{j=0}^{2} z_j \mathbf{D}_j \quad \text{where} \quad \begin{cases} z_0 = \sigma_x\sigma_z + \sigma_y\lambda_i \\ z_1 = \sigma_y\sigma_z + \sigma_x\lambda_i \\ z_2 = \lambda_i^2 - \sigma_z^2 \end{cases}, \tag{7.10}$$

$$\mathbf{Q}_3 = \frac{1}{\sqrt{2}}\mathbf{U}\begin{bmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}\mathbf{V}^T \quad \mathbf{Q}_4 = \frac{1}{\sqrt{2}}\mathbf{U}\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & -1 & 0 \end{bmatrix}\mathbf{V}^T \quad \mathbf{Q}_5 = \frac{1}{\sqrt{2}}\mathbf{U}\begin{bmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \\ -1 & 0 & 0 \end{bmatrix}\mathbf{V}^T \tag{7.11}$$

$$\mathbf{Q}_6 = \frac{1}{\sqrt{2}}\mathbf{U}\begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}\mathbf{V}^T \quad \mathbf{Q}_7 = \frac{1}{\sqrt{2}}\mathbf{U}\begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}\mathbf{V}^T \quad \mathbf{Q}_8 = \frac{1}{\sqrt{2}}\mathbf{U}\begin{bmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix}\mathbf{V}^T. \tag{7.12}$$

How do we know that these eigenvectors are so universal? We had that generic expression for the Hessians from Eqn. 5.50 back in §5.5:

$$\operatorname{vec}\left(\frac{\partial^2\Psi}{\partial\mathbf{F}^2}\right) = \sum_{i=1}^{3} \frac{\partial^2\Psi}{\partial I_i^2}\mathbf{g}_i\mathbf{g}_i^T + \frac{\partial\Psi}{\partial I_i}\mathbf{H}_i. \tag{7.13}$$

[1]Ever since we found these equations, it has bugged me that these three eigenpairs are much less parsimonious than the other six. I suspect that there's a slicker way to write these down, but it clearly involves a trick I haven't thought of yet.

There are only two places that eigenvectors can come from: the $\mathbf{H}_i$ matrices and $\mathbf{g}_i\mathbf{g}_i^T$ outer products. We already have complete knowledge of the eigenvectors that arise in the $\mathbf{H}_i$ terms:

- The Hessian of I_1 has the twist eigenvectors (Eqn. 7.11). Past those, it doesn't care – the rest spans an arbitrary subspace.
- The Hessian of I_2 is diagonal, so it is one big arbitrary, "don't care" subspace.
- The Hessian of I_3 has the twists eigenvectors, just like I_1, so those will already align. Then it has the flip and the scaling eigenvectors, but since I_1 and I_2 both had arbitrary subspaces along those spans, they will just get pinned to flip and scaling.[2]

That leaves $\mathbf{g}_i\mathbf{g}_i^T$ as the only term that could ruin our claim that we've found the nine eigenvectors that can exist. Since it is an outer-product, each $\mathbf{g}_i\mathbf{g}_i^T$ term is its own eigenvector, i.e. the eigenvector is the normalized version of $\mathbf{g}_i$.

All three gradients, $\mathbf{g}_1 = \operatorname{vec}\mathbf{R}$, $\mathbf{g}_2 = \operatorname{vec} 2\mathbf{F}$, $\mathbf{g}_3 =$ Eqn. B.23 are orthogonal to the twist and flip eigenvectors.[3] Therefore, these get chunked in with the $\mathbf{Q}_{i\in\{0,1,2\}}$ eigenvectors, so nothing new can appear from them either.

If you have an isotropic energy, these are the eigenvectors. It doesn't matter how gnarly the energy looks, these are the nine.[4]

7.3.3 Cranking Out Analytic Eigenvalues

If the eigenvectors are exactly the same for all isotropic energies, the only place where energies stand apart *must* be the eigenvalues. We saw a generic (albeit ugly) way to compute the analytic eigenvalues of I_3, and we can generalize this by performing a *deflation* (see e.g. Bunch et al. (1978)). We will explicitly project off the twist and flip eigenmodes, so that what's left over is a $\Re^{3\times3}$ system of scaling modes.

[2] A bunch of linear algebra identities are getting thrown around here that I haven't mentioned before. Don't feel bad if you can't follow it entirely.

[3] This appears as Lemma B.1 in Smith et al. (2019). Take $\mathbf{g}_2 = \operatorname{vec} 2\mathbf{F}$ and $\mathbf{Q}_3$ as an example. First, observe that the dot product $\operatorname{vec}(2\mathbf{F})^T \operatorname{vec}(\mathbf{Q}_3) = 0$ can be written as the trace $\operatorname{vec}(2\mathbf{F})^T \operatorname{vec}(\mathbf{Q}_3) = 2\mathbf{F} : \mathbf{Q}_3 = \operatorname{tr}\left(2\mathbf{F}^T\mathbf{Q}_3\right)$. Then $\operatorname{tr}\left(2\mathbf{F}^T\mathbf{Q}_3\right)$ can be expanded using $\mathbf{F} = \mathbf{U}\boldsymbol{\Sigma}\mathbf{V}^T$ and the definition of $\mathbf{Q}_3$ to get $\operatorname{tr}\left(2\mathbf{V}\boldsymbol{\Sigma}\mathbf{U}^T\mathbf{U}\begin{bmatrix}0 & -1 & 0\\ 1 & 0 & 0\\ 0 & 0 & 0\end{bmatrix}\mathbf{V}^T\right)$ and simplified to $\operatorname{tr}\left(2\begin{bmatrix}\sigma_x & 0 & 0\\ 0 & \sigma_y & 0\\ 0 & 0 & \sigma_z\end{bmatrix}\begin{bmatrix}0 & -1 & 0\\ 1 & 0 & 0\\ 0 & 0 & 0\end{bmatrix}\right)$. The product of the two matrices produces all zeros along the diagonal, so the trace is zero. Moreover, as long as the middle matrix in $\mathbf{Q}_i$ has all zeros along the diagonal, i.e. is a *hollow* matrix, this will always hold. Since the middle matrix in all of $\mathbf{Q}_{3\ldots8}$ are all hollow, they are all orthogonal.

[4] The form of Eqn. 5.50 needs to be slightly more general to encompass *all* isotropic energies; some mixed $\mathbf{g}_i\mathbf{g}_j^T$ where $i \neq j$ terms can appear too. This doesn't make a difference; no new eigenvectors appear.

The entries of this scaling-only, projected matrix $\mathbf{A}$ are:

$$a_{ij} = \sigma_k \frac{\partial \Psi}{\partial I_3} + \frac{\partial^2 \Psi}{\partial I_1^2} + 4\frac{I_3}{\sigma_k}\frac{\partial^2 \Psi}{\partial I_2^2} + \sigma_k I_3 \frac{\partial^2 \Psi}{\partial I_3^2} + 2\sigma_k\left(I_2 - \sigma_k^2\right)\frac{\partial^2 \Psi}{\partial I_2 \partial I_3} + (I_1 - \sigma_k)\left(\sigma_k \frac{\partial^2 \Psi}{\partial I_3 \partial I_1} + 2\frac{\partial^2 \Psi}{\partial I_1 \partial I_2}\right) \tag{7.14}$$

$$a_{ii} = 2\frac{\partial \Psi}{\partial I_2} + \frac{\partial^2 \Psi}{\partial I_1^2} + 4\sigma_i^2 \frac{\partial^2 \Psi}{\partial I_2^2} + \frac{I_3^2}{\sigma_i^2}\frac{\partial^2 \Psi}{\partial I_3^2} + 4\sigma_i \frac{\partial^2 \Psi}{\partial I_1 \partial I_2} + 4I_3 \frac{\partial^2 \Psi}{\partial I_2 \partial I_3} + 2\frac{I_3}{\sigma_i}\frac{\partial^2 \Psi}{\partial I_3 \partial I_1}. \tag{7.15}$$

In these equations, I have used a slightly irritating $\{i, j, k\}$ notation. The symbol a_{ii} denotes a diagonal entry, where if i lines up with the x direction, the σ_i on the right-hand-side is replaced with σ_x. The symbol a_{ij} denotes an off-diagonal entry, and if this corresponds to the (x, y) off-diagonal entry, then σ_k on the right-hand-side becomes σ_z, i.e. whichever coordinate does *not* appear as i or j. If you're still confused, the Matlab code for this is given in Fig. 7.4

The first three eigenvalues of the larger system are now the eigenvalues of the matrix $\mathbf{A}$. These can be solved for any number of ways, both numerical and analytic, using the cubic equation, the method of Jenkins and Traub (1970), the method of Smith (1961), a cornucopia of methods from Golub and Van Loan (2013), or plain old Eigen (Guennebaud et al. (2010)).

The other six eigenvalues, on the other hand, have a much more compact general form:

$$\lambda_3 = \frac{2}{\sigma_x + \sigma_y}\frac{\partial \Psi}{\partial I_1} + 2\frac{\partial \Psi}{\partial I_2} + \sigma_z \frac{\partial \Psi}{\partial I_3} \tag{7.16}$$

$$\lambda_4 = \frac{2}{\sigma_y + \sigma_z}\frac{\partial \Psi}{\partial I_1} + 2\frac{\partial \Psi}{\partial I_2} + \sigma_x \frac{\partial \Psi}{\partial I_3} \tag{7.17}$$

$$\lambda_5 = \frac{2}{\sigma_x + \sigma_z}\frac{\partial \Psi}{\partial I_1} + 2\frac{\partial \Psi}{\partial I_2} + \sigma_y \frac{\partial \Psi}{\partial I_3} \tag{7.18}$$

$$\lambda_6 = 2\frac{\partial \Psi}{\partial I_2} - \sigma_z \frac{\partial \Psi}{\partial I_3} \tag{7.19}$$

$$\lambda_7 = 2\frac{\partial \Psi}{\partial I_2} - \sigma_x \frac{\partial \Psi}{\partial I_3} \tag{7.20}$$

$$\lambda_8 = 2\frac{\partial \Psi}{\partial I_2} - \sigma_y \frac{\partial \Psi}{\partial I_3}. \tag{7.21}$$

7.3.3.1 St. Venant-Kirchhoff

Let's push through one instance of these eigenvalues, using our old friend the StVK energy:

$$\Psi_{\text{StVK}} = \mu \|\mathbf{E}\|_F^2 + \frac{\lambda}{2}\operatorname{tr}^2(\mathbf{E}). \tag{7.22}$$

```matlab
function [A] = Get_Stretching_System(Psi)
  syms I1 I2 I3 sigma0 sigma1 sigma2 sigmai sigmak real;
  A = sym(zeros(3,3));

  % first derivatives
  firstI1 = diff(Psi, I1);
  firstI2 = diff(Psi, I2);
  firstI3 = diff(Psi, I3);

  % second derivatives
  secondI1 = diff(firstI1, I1);
  secondI2 = diff(firstI2, I2);
  secondI3 = diff(firstI3, I3);

  % mixed derivatives
  secondI1I2 = diff(firstI1, I2);
  secondI1I3 = diff(firstI1, I3);
  secondI2I3 = diff(firstI2, I3);

  % get the diagonal entry
  aii = 2 * firstI2 + secondI1 + 4 * sigmai^2 * secondI2 + ...
        (I3 / sigmai)^2 * secondI3 + 4 * sigmai * secondI1I2 + ...
        4 * I3 * secondI2I3 + 2 * (I3 / sigmai) * secondI1I3;
  A(1,1) = subs(aii, sigmai, sigma0);
  A(2,2) = subs(aii, sigmai, sigma1);
  A(3,3) = subs(aii, sigmai, sigma2);

  % get the off-diagonal entry
  aij = sigmak * firstI3 + secondI1 + 4 * (I3 / sigmak) * secondI2 + ...
        sigmak * I3 * secondI3 + ...
        2 * sigmak * (I2 - sigmak^2) * secondI2I3 + ...
        (I1 - sigmak) * (sigmak * secondI1I3 + 2 * secondI1I2);
  A(1,2) = subs(aij, sigmak, sigma2);
  A(1,3) = subs(aij, sigmak, sigma1);
  A(2,3) = subs(aij, sigmak, sigma0);

  % symmetrize and simplify
  A(2,1) = A(1,2);
  A(3,1) = A(1,3);
  A(3,2) = A(2,3);
  A = Simplify_Invariants(A);
end
```

Figure 7.4.: Matlab/Octave code to get an analytic expressions for stretching subspace of any isotropic energy `Psi`.

The entries of the $\mathbf{A}$ matrix evaluate to something not so bad-looking:

$$\begin{aligned} a_{ii} &= -\mu + \frac{\lambda}{2}(I_2 - 3) + (\lambda + 3\mu)\,\sigma_i^2 \\ a_{ij} &= \lambda \sigma_i \sigma_j. \end{aligned}$$

The last six eigenvalues are available directly:

$$\lambda_3 = -\mu + \frac{\lambda}{2}(I_2 - 3) + \mu\left(\sigma_y^2 + \sigma_z^2 - \sigma_y\sigma_z\right) \tag{7.23}$$

$$\lambda_4 = -\mu + \frac{\lambda}{2}(I_2 - 3) + \mu\left(\sigma_x^2 + \sigma_z^2 - \sigma_x\sigma_z\right) \tag{7.24}$$

$$\lambda_5 = -\mu + \frac{\lambda}{2}(I_2 - 3) + \mu\left(\sigma_x^2 + \sigma_y^2 - \sigma_x\sigma_y\right) \tag{7.25}$$

$$\lambda_6 = -\mu + \frac{\lambda}{2}(I_2 - 3) + \mu\left(\sigma_y^2 + \sigma_z^2 + \sigma_y\sigma_z\right) \tag{7.26}$$

$$\lambda_7 = -\mu + \frac{\lambda}{2}(I_2 - 3) + \mu\left(\sigma_x^2 + \sigma_z^2 + \sigma_x\sigma_z\right) \tag{7.27}$$

$$\lambda_8 = -\mu + \frac{\lambda}{2}(I_2 - 3) + \mu\left(\sigma_x^2 + \sigma_y^2 + \sigma_x\sigma_y\right) \tag{7.28}$$

A little verbose, but not too bad.

7.3.4 If You're Lucky, Things Get Simpler

If we grind Ψ_{ARAP} through Eqns. 7.14 - 7.21, we will get exactly the same eigenvalue expressions that we obtained in Eqns. 7.2 - 7.5. But hang on a minute ... where did all the ugly stuff from Eqns. 7.14 and 7.15 go?

In some cases, if you're lucky, the first three eigenvectors work out to the scaling modes in Eqn. 7.8. In these cases, the eigenvalues also take on the following closed forms:

$$\lambda_0 = 2\frac{\partial \Psi}{\partial I_2} + \frac{\partial^2 \Psi}{\partial I_1^2} + 4\sigma_x^2\frac{\partial^2 \Psi}{\partial I_2^2} + \sigma_y^2\sigma_z^2\frac{\partial^2 \Psi}{\partial I_3^2} + 4\sigma_x\frac{\partial^2 \Psi}{\partial I_1 \partial I_2} + 4I_3\frac{\partial^2 \Psi}{\partial I_2 \partial I_3} + 2\sigma_y\sigma_z\frac{\partial^2 \Psi}{\partial I_3 \partial I_1} \tag{7.29}$$

$$\lambda_1 = 2\frac{\partial \Psi_q}{\partial I_2} + \frac{\partial^2 \Psi_q}{\partial I_1^2} + 4\sigma_y^2\frac{\partial^2 \Psi_q}{\partial I_2^2} + \frac{I_3^2}{\sigma_y^2}\frac{\partial^2 \Psi_q}{\partial I_3^2} + 4\sigma_y\frac{\partial^2 \Psi_q}{\partial I_1 \partial I_2} + 4I_3\frac{\partial^2 \Psi_q}{\partial I_2 \partial I_3} + 2\frac{I_3}{\sigma_y}\frac{\partial^2 \Psi_q}{\partial I_3 \partial I_1} \tag{7.30}$$

$$\lambda_2 = 2\frac{\partial \Psi_q}{\partial I_2} + \frac{\partial^2 \Psi_q}{\partial I_1^2} + 4\sigma_z^2\frac{\partial^2 \Psi_q}{\partial I_2^2} + \frac{I_3^2}{\sigma_z^2}\frac{\partial^2 \Psi_q}{\partial I_3^2} + 4\sigma_z\frac{\partial^2 \Psi_q}{\partial I_1 \partial I_2} + 4I_3\frac{\partial^2 \Psi_q}{\partial I_2 \partial I_3} + 2\frac{I_3}{\sigma_z}\frac{\partial^2 \Psi_q}{\partial I_3 \partial I_1} \tag{7.31}$$

They're still not that pretty, but the expression you get out at the end might be, [5] and the need to call any additional solver has been removed entirely.

[5]Also, remember I promised I'll bake all this out into a Matlab script for you.

How do you know if you've hit the **Eigenvalue Jackpot?** If *all* the off-diagonal entries a_{ij} work out to zero, then the scaling modes become decoupled, and you've hit the jackpot. You can use Eqns. 7.29 - 7.31 instead of the building out an unwieldy matrix.

A bunch of energies hit the jackpot.

7.3.4.1 Symmetric Dirichlet

The Symmetric Dirichlet energy from Smith and Schaefer (2015b),

$$\Psi_{\text{SD}} = \frac{\left(\|\mathbf{F}\|_F^2 + \|\mathbf{F}^{-1}\|_F^2\right)}{2}, \tag{7.32}$$

has the following eigenvalues:

$$\begin{aligned}
\lambda_0 &= 1 + \frac{3}{\sigma_x^4} & \lambda_1 &= 1 + \frac{3}{\sigma_y^4} & \lambda_2 &= 1 + \frac{3}{\sigma_z^4} \\
\lambda_3 &= 1 + \frac{\sigma_x^2}{I_3^2} - \frac{(I_2 - \sigma_x^2)\sigma_x^3}{I_3^3} & \lambda_4 &= 1 + \frac{\sigma_y^2}{I_3^2} - \frac{(I_2 - \sigma_y^2)\sigma_y^3}{I_3^3} & \lambda_5 &= 1 + \frac{\sigma_z^2}{I_3^2} - \frac{(I_2 - \sigma_z^2)\sigma_z^3}{I_3^3} \\
\lambda_6 &= 1 + \frac{\sigma_x^2}{I_3^2} + \frac{(I_2 - \sigma_x^2)\sigma_x^3}{I_3^3} & \lambda_7 &= 1 + \frac{\sigma_y^2}{I_3^2} + \frac{(I_2 - \sigma_y^2)\sigma_y^3}{I_3^3} & \lambda_8 &= 1 + \frac{\sigma_z^2}{I_3^2} + \frac{(I_2 - \sigma_z^2)\sigma_z^3}{I_3^3}.
\end{aligned}$$

7.3.4.2 Symmetric ARAP

The Symmetric ARAP energy from Shtengel et al. (2017),

$$\Psi_{\text{SARAP}} = \frac{\mu}{2}\left(\|\mathbf{F} - \mathbf{R}\|^2 + \|\mathbf{F}^{-1} - \mathbf{R}^{-1}\|^2\right), \tag{7.33}$$

also hits the jackpot:

$$\lambda_0 = \mu\left(1 - \frac{2}{\sigma_x^3} + \frac{3}{\sigma_x^4}\right) \qquad \lambda_1 = \mu\left(1 - \frac{2}{\sigma_y^3} + \frac{3}{\sigma_y^4}\right) \qquad \lambda_2 = \mu\left(1 - \frac{2}{\sigma_z^3} + \frac{3}{\sigma_z^4}\right)$$

$$\begin{aligned}
\lambda_3 &= \mu + \frac{\mu}{\sigma_y + \sigma_z}\left[\frac{1}{\sigma_y^2} + \frac{1}{\sigma_z^2} - \frac{1}{\sigma_y^3} - \frac{1}{\sigma_z^3} - 2\right] \\
\lambda_4 &= \mu + \frac{\mu}{\sigma_x + \sigma_z}\left[\frac{1}{\sigma_x^2} + \frac{1}{\sigma_z^2} - \frac{1}{\sigma_x^3} - \frac{1}{\sigma_z^3} - 2\right] \\
\lambda_5 &= \mu + \frac{\mu}{\sigma_x + \sigma_y}\left[\frac{1}{\sigma_x^2} + \frac{1}{\sigma_y^2} - \frac{1}{\sigma_x^3} - \frac{1}{\sigma_y^3} - 2\right]
\end{aligned}$$

$$\lambda_6 = \mu\left[1 + \frac{1}{(\sigma_y\sigma_z)^2} + \frac{\sigma_y^2+\sigma_z^2}{(\sigma_y\sigma_z)^3} - \frac{1}{\sigma_y^2\sigma_z} - \frac{1}{\sigma_y\sigma_z^2}\right]$$
$$\lambda_7 = \mu\left[1 + \frac{1}{(\sigma_x\sigma_z)^2} + \frac{\sigma_x^2+\sigma_z^2}{(\sigma_x\sigma_z)^3} - \frac{1}{\sigma_x^2\sigma_z} - \frac{1}{\sigma_x\sigma_z^2}\right]$$
$$\lambda_8 = \mu\left[1 + \frac{1}{(\sigma_x\sigma_y)^2} + \frac{\sigma_x^2+\sigma_y^2}{(\sigma_x\sigma_y)^3} - \frac{1}{\sigma_x^2\sigma_y} - \frac{1}{\sigma_x\sigma_y^2}\right]$$

7.3.4.3 *Co-rotational (sort of)*

The Co-rotational energy hits two out of three cherries on the slot-machine,

$$\Psi_{\text{CR}} = \mu\|\mathbf{F} - \mathbf{R}\|^2 + \frac{\lambda}{2}\operatorname{tr}^2(\mathbf{S} - \mathbf{I})$$

and has the scaling modes weakly couple in a single rotation mode:

$$\lambda_0 = 2\mu + 3\lambda \qquad \mathbf{Q}_0 = \frac{1}{\sqrt{3}}\mathbf{R}$$
$$\lambda_{1,2} = 2\mu \qquad \mathbf{Q}_{1,2} = \text{2D subspace orthogonal to } \mathbf{R}.$$

$$\lambda_3 = 2\mu + 2\lambda\frac{(I_1 - 3 - 2\mu)}{(\sigma_y + \sigma_z)} \qquad \lambda_4 = 2\mu + 2\lambda\frac{(I_1 - 3 - 2\mu)}{(\sigma_z + \sigma_x)} \qquad \lambda_5 = 2\mu + 2\lambda\frac{(I_1 - 3 - 2\mu)}{(\sigma_x + \sigma_y)}$$
$$\lambda_6 = 2\mu \qquad \lambda_7 = 2\mu \qquad \lambda_8 = 2\mu.$$

For all of these energies, projecting to semi-positive-definiteness is now straightforward: look at the eigenvalues, and if they're below zero, snap them to zero. As we did with ARAP, we can also derive the exact conditions under which they become indefinite in order to better understand their behavior.

None of these energies use the I_3 invariant, and that's not a coincidence. The invariant $I_3 = \sigma_x\sigma_y\sigma_z$ couples the scaling modes; the exact thing that all these jackpot winners *don't* do. If your energy has I_3 lurking in it somewhere, then it's inevitable that the first three eigenvalues will become non-linearly coupled.

7.3.4.4 *Finally, Some Code, To Save You From Error-Riddled Typing*

As promised, the code to produce these analytic eigensystems is provided in Fig. 7.5. These implement the Simple Eigenvalue Jackpot case. If I_3 appears in your energy, you'll still need to build out your $\mathbf{A}$ matrix using Eqns. 7.14 and 7.15. We will see an example of that next.

```
function [lambdas] = Get_Analytic_Eigenvalues(Psi)
  syms I1 I2 I3 real;
  syms sigma0 sigma1 sigma2 sigmai sigmaj sigmak real;
  lambdas = sym(zeros(9,1));

  % x-twist, y-twist, and z-twist
  lambdas(4) = (2 / (sigma1 + sigma2)) * diff(Psi, I1) + ...
               2 * diff(Psi, I2) + sigma0 * diff(Psi, I3);
  lambdas(5) = (2 / (sigma0 + sigma2)) * diff(Psi, I1) + ...
               2 * diff(Psi, I2) + sigma1 * diff(Psi, I3);
  lambdas(6) = (2 / (sigma0 + sigma1)) * diff(Psi, I1) + ...
               2 * diff(Psi, I2) + sigma2 * diff(Psi, I3);

  % x-flip, y-flip, and z-flip
  lambdas(7) = 2 * diff(Psi, I2) - sigma0 * diff(Psi, I3);
  lambdas(8) = 2 * diff(Psi, I2) - sigma1 * diff(Psi, I3);
  lambdas(9) = 2 * diff(Psi, I2) - sigma2 * diff(Psi, I3);

  % x-scale, y-scale and z-scale
  lambdaScale = 2 * diff(Psi, I2) + diff(Psi, I1, 2) + ...
                4 * sigmai^2 * diff(Psi, I2, 2) + ...
                sigmaj^2 * sigmak^2 * diff(Psi, I3, 2) + ...
                4 * sigmai * diff(diff(Psi, I1), I2) + ...
                4 * I3 * diff(diff(Psi, I2), I3) + ...
                2 * sigmaj * sigmak * diff(diff(Psi, I3), I1);
  lambdas(1) = subs(lambdaScale, {sigmai, sigmaj, sigmak}, ...
                                 {sigma0, sigma1, sigma2});
  lambdas(2) = subs(lambdaScale, {sigmai, sigmaj, sigmak}, ...
                                 {sigma1, sigma0, sigma2});
  lambdas(3) = subs(lambdaScale, {sigmai, sigmaj, sigmak}, ...
                                 {sigma2, sigma0, sigma1});

  % try to get the simplest expression
  lambdas = Simplify_Invariants(lambdas);
end
```

Figure 7.5.: Matlab/Octave code to get analytic expressions for the eigenvalues of any isotropic energy `Psi`. The routine for `Simplify_Invariants` is defined in Fig. 7.6.

```
function [out] = Simplify_Invariants(in)
  syms I1 I2 I3 sigma0 sigma1 sigma2 real;
  out = subs(in, I1, sigma0 + sigma1 + sigma2);
  out = subs(out, I2, sigma0^2 + sigma1^2 + sigma2^2);
  out = subs(out, I3, sigma0 * sigma1 * sigma2);
  out = simplify(out);
  out = subs(out, sigma0 + sigma1 + sigma2, I1);
  out = subs(out, sigma0^2 + sigma1^2 + sigma2^2, I2);
  out = subs(out, sigma0 * sigma1 * sigma2, I3);
  out = simplify(out);
end
```

Figure 7.6.: Matlab/Octave code for simplifying expressions using the definitions of the invariants. This is called from `Get_Analytic_Eigenvalues` in Fig. 7.5.

7.4 The Stable Neo-Hookean Eigensystem

Let's take a closer look at the Stable Neo-Hookean eigensystem, because it exhibits some qualitatively different behaviors from all the "jackpot" energies we just saw.

We can obtain the complete eigenmodes by punching the Ψ_{SNH} symbolically into Matlab/Octave code and running the code from Figs. 7.4 and 7.5. An example of the calls is given in Fig. 7.7.

The resulting scaling system is:

$$\begin{aligned} a_{ii} &= \mu + \lambda \frac{I_3^2}{\sigma_i^2} \\ a_{ij} &= \sigma_k \left(\lambda(2I_3 - 1) - \mu\right). \end{aligned}$$

The twist and flip eigenvalues are:

$$\lambda_3 = \mu + \sigma_z \left(\lambda(I_3 - 1) - \mu\right) \tag{7.34}$$
$$\lambda_4 = \mu + \sigma_x \left(\lambda(I_3 - 1) - \mu\right) \tag{7.35}$$
$$\lambda_5 = \mu + \sigma_y \left(\lambda(I_3 - 1) - \mu\right) \tag{7.36}$$
$$\lambda_6 = \mu - \sigma_z \left(\lambda(I_3 - 1) - \mu\right) \tag{7.37}$$
$$\lambda_7 = \mu - \sigma_x \left(\lambda(I_3 - 1) - \mu\right) \tag{7.38}$$
$$\lambda_8 = \mu - \sigma_y \left(\lambda(I_3 - 1) - \mu\right). \tag{7.39}$$

7.4.1 When Does It Go Indefinite?

When do these eigenvalues go indefinite? These are slightly more difficult to analyze, because the relations are still non-linear. When we were obtaining the eigensystem of I_3, the flip modes corresponding to $\lambda_{6\ldots8}$ were *always* negative, so let's take a look at what happens after they've been plugged into the Stable Neo-Hookean energy.

```
 1 % Boilerplate to load up the symbolic package in Octave
 2 isOctave = exist('OCTAVE_VERSION', 'builtin') ~= 0;
 3 if (isOctave)
 4     pkg load symbolic;
 5 end
 6
 7 % Declare the invariants and constants
 8 syms I1 I2 I3 mu lambda real;
 9
10 % Build the SNH energy
11 Psi = (mu / 2) * (I2 - 3) - mu * (I3 - 1) + (lambda / 2) * (I3 - 1)^2;
12 % Build the ARAP energy
13 % Psi = I2 - 2 * I1 + 3
14
15 % Get the analytic twist and flip eigenvalues, and
16 % maybe the stretching values too, if you hti the jackpot
17 [lambdas] = Get_Analytic_Eigenvalues(Psi);
18
19 % Get the stretching matrix
20 A = Get_Stretching_System(Psi);
21
22 % Did we hit the Simple Eigenvalue Jackpot?
23 jackpot = A(1,2) + A(1,3) + A(2,3);
24 if (jackpot == sym(0))
25   printf('You HIT the Jackpot!\n');
26   printf('No stretching matrix is needed.\n');
27   printf('Here are your eigenvalues:\n');
28   lambdas
29 else
30   printf('You MISSED the Jackpot!\n');
31   printf('The stretching eigenvalues are the three\n');
32   printf('eigenvalues from this matrix:\n');
33   A
34   printf('The last six are these:\n');
35   lambdas(4:9)
36 end
```

Figure 7.7.: Matlab/Octave code for obtaining the analytic eigenvalues and stretching system for the Stable Neo-Hookean energy, Ψ_{SNH}. If you want to see what it feels like to win the Simple Eigenvalue Jackpot, uncomment line 13 and run everything on Ψ_{ARAP}.

In particular, we can focus on λ_6, and examine the conditions under which it becomes zero, i.e. when it is on the cusp of becoming indefinite. Solving for the σ_x roots of the quadratic

$$\mu - \sigma_z \left(\lambda(\sigma_x\sigma_y\sigma_z - 1) - \mu\right) = 0 \tag{7.40}$$

yields the two roots:

$$\sigma_x = \frac{\lambda + \mu \pm \sqrt{4\sigma_y\sigma_z\lambda\mu + (\lambda+\mu)^2}}{2\lambda\sigma_y\sigma_z}. \tag{7.41}$$

Let's examine a specific instance of λ and μ. As $\nu \to \frac{1}{2}$, which corresponds to biological tissue like muscle and skin, $\lambda \to \infty$ and overwhelms the μ term.

Another way to phrase this domination of λ over μ is to set $\lambda = 1$ and $\mu = 0$:

$$\sigma_x = \frac{1 \pm 1}{2\sigma_y\sigma_z} = \left\{0, \frac{1}{\sigma_y\sigma_z}\right\}. \tag{7.42}$$

The $\sigma_x = 0$ corresponds to the total pancake case where the elements has been squashed to zero volume. Indefiniteness make sense there; there are myriad ways to un-pancake yourself.

The $\sigma_x = \frac{1}{\sigma_y\sigma_z}$ condition is interesting if we view $\sigma_y\sigma_z$ as the cross-sectional area orthogonal to the σ_x direction:

$$\sigma_x = \frac{1}{\text{area}_\perp}.$$

If the x direction has been squashed so much that its length is *less than the inverse of* $\text{area}_\perp$, then the material will buckle. These cases are illustrated in Fig. 7.8. If the middle of your element does not bulge out far enough to stabilize your current amount of squashing, the material has no choice but to buckle.

The twist eigenvalues produce the exact same roots under $\lambda = 1$, $\mu = 0$, so those modes are not safe from indefiniteness either. These conditions only apply to $\nu = \frac{1}{2}$, but nonetheless, our analytic eigenanalysis gives a glimpse into the qualitative behavior of the Stable Neo-Hookean energy that was not possible before.

I wonder what running this eigenanalysis on *your* favorite energy will reveal?

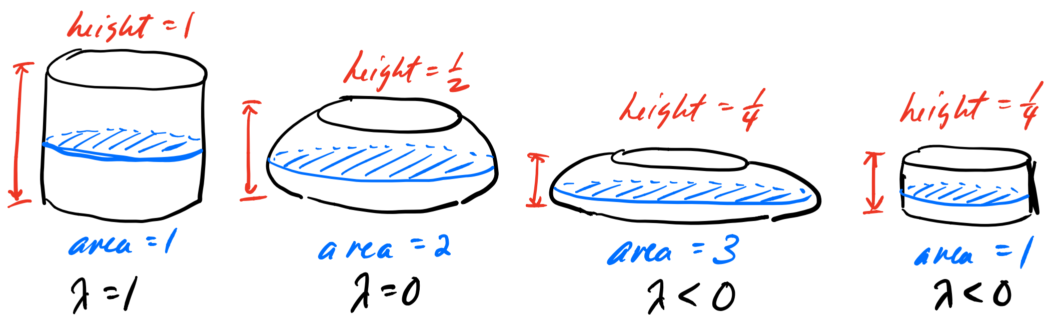


Figure 7.8.: **From left to right:** An undeformed volume is safely positive definite. As it is squashed by $\frac{1}{2}$, the cross-sectional area has to be at least 2 to prevent it from buckling. If this condition isn't met, the volume does not have base wide enough to hold up, and must buckle.

Chapter 8

A Better Way For Anisotropic Solids

This chapter is a loose retelling of §4 and §5 from Kim et al. (2019). If you want to cut to the chase, the Matlab code that spits out the analytic eigenvalues of an anisotropic energy is given in §8.2.5.

8.1 What's Anisotropy?

So far, we have looked at *isotropic* solids. If you squish them down, they bulge out the exactly the same amount in all directions. If you stretch them out, they resist the same amount, no matter which direction you're pulling in. Most real-world solids don't do this exactly; they can be stiffer in some directions and softer in others.

Let's compare a length of rubber band and a length of hemp rope sagging under gravity (Fig. 8.1). The rubber band sags a whole bunch. While the rope sags too, it doesn't dip

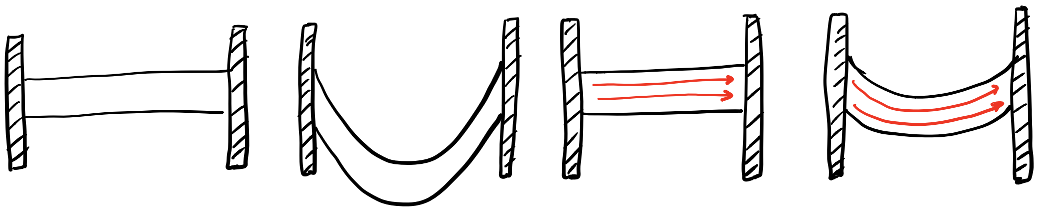


Figure 8.1.: **Left pair:** A length of rubber band sags a lot under gravity. **Right pair:** A length of hemp rope sags less, because it contains stiff fibers that all run in a preferred direction.

as far down, because it is composed of bundles of stiff fibers that all run in a single, preferred direction. In the extreme, if we ran a steel reinforcement rod down the length of the rope, it wouldn't sag at all.

The material is *anisotropic*. It's stiffer in some directions than it is in others.

The same thing can happen in a volumetric solid. When we squash down an isotropic material (Fig. 8.2), it bulges out equally in all directions. If the solid contains fibers that

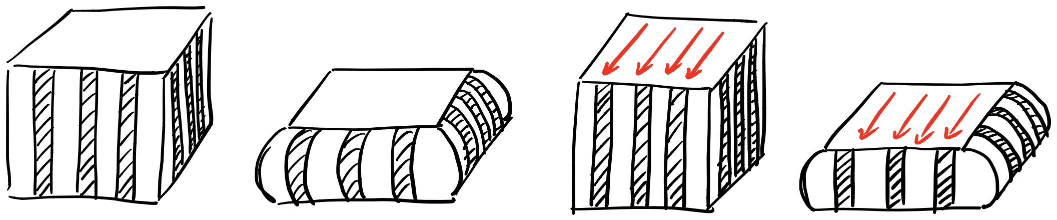


Figure 8.2.: **Left pair:** An isotropic material bulges out equally in all directions **Right pair:** Fiber reinforcement has been inserted, so it bulges less in that direction. The lines along the front remain vertical.

all run in a specific direction, then it will be reluctant to pooch out in that direction.[1] One everyday material that does this is *biological muscle*. All the muscles in your body contain fibers, and when you flex a muscle, your brain is telling those aligned fibers to contract, i.e. *become shorter* (Fig. 8.3). When these fibers become shorter, they pull on the bones on

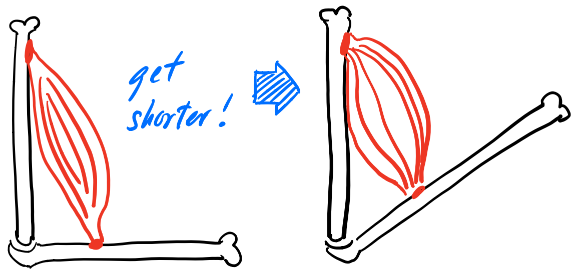


Figure 8.3.: Muscles are an anisotropic material. When fibers contract, they pull on the bones on either side of the muscle.

either ends of the muscle, and that's why your body moves. End of anatomy lesson.

[1]Avi Goyal suggested another (totally separate) example: trying to squash a block of Post-It Notes. Pound on top with your fist, and basically nothing happens. But pinch it along the sides with your fingers, and it pooches out easily.

8.2 The (Wrong) Cauchy-Green Invariants, Again

8.2.1 The $IV_\mathbf{C}$ and $V_\mathbf{C}$ Invariants

How do we tell our energies that there's a preferred anisotropy direction that we want to be stiffer? Even if we have some anisotropy direction $\mathbf{a} \in \Re^3$ in mind, there's nowhere to plug it into our existing invariants.

In biomechanics (see e.g. the survey of Chagnon et al. (2015)), this is solved by introducing a new set of invariants that explicitly take into account some vector $\mathbf{a}$. These new invariants gets stacked onto the end of our existing sequence of Cauchy-Green invariants:[2]

$$IV_\mathbf{C} = \mathbf{a}^T\mathbf{C}\mathbf{a} \qquad\qquad V_\mathbf{C} = \mathbf{a}^T\mathbf{C}^T\mathbf{C}\mathbf{a}, \tag{8.1}$$

where, as before, $\mathbf{C} = \mathbf{F}^T\mathbf{F}$. The anisotropy direction only needs to be defined *at the rest*

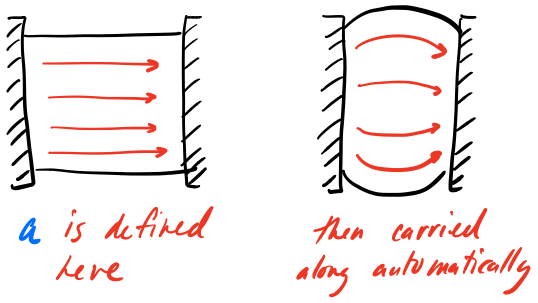


Figure 8.4.: The anisotropy direction $\mathbf{a}$ only needs to be defined *at the rest shape.* It gets automatically carried along as the object deforms.

shape (Fig. 8.4). To see why, we can unpack $IV_\mathbf{C}$ into:

$$IV_\mathbf{C} = \mathbf{a}^T\mathbf{F}^T\mathbf{F}\mathbf{a} = (\mathbf{F}\mathbf{a})^T\,\mathbf{F}\mathbf{a} = \left(\mathbf{D}_s\mathbf{D}_m^{-1}\mathbf{a}\right)^T\mathbf{D}_s\mathbf{D}_m^{-1}\mathbf{a}. \tag{8.2}$$

On the right, we can see that the invariant is just two applications of $\mathbf{D}_s\mathbf{D}_m^{-1}\mathbf{a}$. The first multiply, $\mathbf{D}_m^{-1}\mathbf{a}$, rotates the fiber direction out of the original rest shape. Then it gets rotated into the current deformation by the second multiply, $\mathbf{D}_s$. Thus, the invariant will automatically transform the original $\mathbf{a}$ into the appropriate orientation for the current deformation.

8.2.2 Gradients and Hessians, Again

Let's look at how to take the gradient and Hessian of an anisotropic energy.

[2]If you're a real invariant nerd, it may bother you that these invariants don't actually arise from the characteristic polynomial of $\mathbf{F}$. Where's their generating equation? I don't know. They're definitely translation and rotation invariant though, so they at least fit the definition of "invariant" in that sense.

8.2.2.1 *Anisotropic St. Venant Kirchhoff*

There's a bunch of different anisotropic energies appearing in the literature that we can pick from, but for some reason people seem reluctant to name them. This one appears in anonymous form in both Liu et al. (2017) and Holzapfel (2005):

$$\Psi_{\text{AStVK}} = \frac{\mu}{2}\left(IV_{\mathbf{C}} - 1\right)^2. \tag{8.3}$$

Since it follows the StVK approach of squaring a $\mathbf{C}$-like term, let's call it *Anisotropic StVK*.

The gradient and Hessian are,

$$\frac{\partial \Psi_{\text{AStVK}}}{\partial \mathbf{F}} = \mu\left(IV_{\mathbf{C}} - 1\right)\frac{\partial IV_{\mathbf{C}}}{\partial \mathbf{F}} \tag{8.4}$$

$$\operatorname{vec}\left(\frac{\partial^2 \Psi_{\text{AStVK}}}{\partial \mathbf{F}^2}\right) = \mu\left((IV_{\mathbf{C}} - 1)\mathbf{H}_{IV} + \mathbf{g}_{IV}\mathbf{g}_{IV}^T\right), \tag{8.5}$$

where

$$\frac{\partial IV_{\mathbf{C}}}{\partial \mathbf{F}} = 2\mathbf{F}\mathbf{a}\mathbf{a}^T \tag{8.6}$$

$$\mathbf{g}_{\text{IV}} = \operatorname{vec}\left(\frac{\partial IV_{\mathbf{C}}}{\partial \mathbf{F}}\right) = \operatorname{vec}\left(2\mathbf{F}\mathbf{a}\mathbf{a}^T\right) \tag{8.7}$$

$$\mathbf{H}_{IV} = \operatorname{vec}\left(\frac{\partial^2 IV_{\mathbf{C}}}{\partial \mathbf{F}^2}\right) = 2\mathbf{a}\mathbf{a}^T \otimes \mathbf{I}. \tag{8.8}$$

Rather than slugging through these derivatives every time, just like in the isotropic case (Eqns. 5.8 and 5.50), there is a generic, oh-thank-my-lucky-stars-I-only-need-scalar-derivatives way of deriving the Hessian:

$$\operatorname{vec}\left(\frac{\partial^2 \Psi}{\partial \mathbf{F}^2}\right) = \frac{\partial \Psi}{\partial IV_{\mathbf{C}}}\mathbf{H}_{IV} + \frac{\partial^2 \Psi}{\partial IV_{\mathbf{C}}^2}\mathbf{g}_{IV}\mathbf{g}_{IV}^T. \tag{8.9}$$

Once you have the relatively easy $\frac{\partial \Psi}{\partial IV_{\mathbf{C}}}$ and $\frac{\partial^2 \Psi}{\partial IV_{\mathbf{C}}^2}$ terms worked out[3], you're all done.

8.2.2.2 *Anisotropic Square Root*

Let's try the generic approach on another (usually unnamed) energy that appears in biomechanics (Alastrué et al. (2008)):

$$\Psi_{\text{ASqrt}} = \frac{\mu}{2}\left(\sqrt{IV_{\mathbf{C}}} - 1\right)^2. \tag{8.10}$$

[3]Again, don't be ashamed to use Mathematica or Matlab. Really.

It's almost exactly the same as the Anisotropic StVK energy, except for the square root. For that reason, let's call it the *Anisotropic Square Root* model. The gradient is:

$$\frac{\partial \Psi_{\text{ASqrt}}}{\partial \mathbf{F}} = \mu \left(1 - \frac{1}{\sqrt{IV_{\mathbf{C}}}}\right) \mathbf{F}\mathbf{a}\mathbf{a}^T. \tag{8.11}$$

We can get the Hessian by taking the scalar derivatives,

$$\frac{\partial \Psi_{\text{ASqrt}}}{\partial IV_{\mathbf{C}}} = \frac{\mu}{2}\left(1 - \frac{1}{\sqrt{IV_{\mathbf{C}}}}\right) \qquad \frac{\partial^2 \Psi_{\text{ASqrt}}}{\partial IV_{\mathbf{C}}^2} = \frac{\mu}{4 IV_{\mathbf{C}}^{\frac{3}{2}}},$$

and plugging them in to get our final Hessian[4]:

$$\text{vec}\left(\frac{\partial^2 \Psi_{\text{ASqrt}}}{\partial \mathbf{F}^2}\right) = \frac{\mu}{2}\left(1 - \frac{1}{\sqrt{IV_{\mathbf{C}}}}\right) \mathbf{H}_{IV} + \frac{\mu}{4 IV_{\mathbf{C}}^{\frac{3}{2}}} \mathbf{g}_{IV} \mathbf{g}_{IV}^T. \tag{8.12}$$

All done. Except that the Hessian might be indefinite.

8.2.3 The Eigensystem of $IV_{\mathbf{C}}$

With Eqn. 8.9 in hand, we can apply the same approach as §7.3. If we can get an analytic expression for the eigendecomposition of $\mathbf{H}_{IV}$, then we can get the analytic eigenstructure of *any* energy based on $IV_{\mathbf{C}}$.

Fortunately, Matlab hack-and-slashing quickly reveals that the eigensystem of $\mathbf{H}_{IV}$ has a rank-six null-space, so there are only three non-trivial eigenpairs:

$$\lambda_{0,1,2} = 2 \tag{8.13}$$

$$\mathbf{Q}_0 = \begin{bmatrix} & \mathbf{a}^T & \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \qquad \mathbf{Q}_1 = \begin{bmatrix} 0 & 0 & 0 \\ & \mathbf{a}^T & \\ 0 & 0 & 0 \end{bmatrix} \qquad \mathbf{Q}_2 = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ & \mathbf{a}^T & \end{bmatrix}. \tag{8.14}$$

First off, it *is* an arbitrary subspace, so the forms given for $\mathbf{Q}_{0,1,2}$ are not unique. They're convenient to write in this form though. Second, the most general form of the eigenvalues is $\lambda_{0,1,2} = 2\|\mathbf{a}\|_2^2$, but usually $\mathbf{a}$ is specified as a unit vector. The purpose of $\mathbf{a}$ is to convey that you want things stiffer in *this* direction, but the amount of stiffness you want would be communicated by some sort of μ constant. Baking it into the magnitude of $\mathbf{a}$ would just cause confusion.

[4]This expression looks slightly different from the one I gave in Kim et al. (2019) because I pushed some factors of 2 around in a different way. However, I assure you that if you work through all the terms, they are indeed equivalent.

8.2.4 The Eigensystems of Arbitrary $IV_{\mathbf{C}}$ Energies

With the eigensystem of $\mathbf{H}_{IV}$ in hand, we can now build the eigensystems of arbitrary $IV_{\mathbf{C}}$-based energies. Looking again at the generic form of these Hessians,

$$\operatorname{vec}\left(\frac{\partial^2 \Psi}{\partial \mathbf{F}^2}\right) = \frac{\partial \Psi}{\partial IV_{\mathbf{C}}} \mathbf{H}_{IV} + \frac{\partial^2 \Psi}{\partial IV_{\mathbf{C}}^2} \mathbf{g}_{IV} \mathbf{g}_{IV}^T,$$

we already know the eigensystem of the first term on the right, $\frac{\partial \Psi}{\partial IV_{\mathbf{C}}} \mathbf{H}_{IV}$. The eigenvectors are still those from Eqn. 8.14, and the eigenvalues all get scaled by $\frac{\partial \Psi}{\partial IV_{\mathbf{C}}}$.

Adding the $\frac{\partial^2 \Psi}{\partial IV_{\mathbf{C}}^2} \mathbf{g}_{IV} \mathbf{g}_{IV}^T$ term complicates things, because (again) there's no general theorem that helps you get the eigensystem of a sum of matrices, even if you already know the eigensystems of the summands. But, while we're not adding to a diagonal matrix like we did in §7.2 and §7.3, we're doing something similar. We're adding a rank-one update, $\mathbf{g}_{IV} \mathbf{g}_{IV}^T$, to a matrix with an arbitrary subspace, $\mathbf{H}_{IV}$.

In our case, what happens is that *one* of the directions in the arbitrary subspace gets pinned to $\mathbf{g}_{IV}$, i.e. the direction of the rank-one update. The other two directions are still arbitrary, but they then get pinned to the subspace that is orthogonal to $\mathbf{g}_{IV}$.

More concretely, the first eigenpair is:

$$\lambda_0 = 2\left(\frac{\partial \Psi}{\partial IV_{\mathbf{C}}} + 2 IV_{\mathbf{C}} \frac{\partial^2 \Psi}{\partial IV_{\mathbf{C}}^2}\right) \qquad \mathbf{Q}_0 = \frac{1}{\sqrt{IV_{\mathbf{C}}}} \mathbf{F}\mathbf{a}\mathbf{a}^T. \tag{8.15}$$

Keeping in mind that $\mathbf{g}_{IV} = 2\operatorname{vec}\left(\mathbf{F}\mathbf{a}\mathbf{a}^T\right)$, the eigenmatrix $\mathbf{Q}_0$ has indeed been pinned to a normalized version of $\mathbf{g}_{IV}$.

The other two eigenvalues are:

$$\lambda_{1,2} = 2 \frac{\partial \Psi}{\partial IV_{\mathbf{C}}}. \tag{8.16}$$

The arbitrary subspace takes a little effort to construct, and in practice you never actually compute these explicitly. However, to show that it *can* be done, we first define the twist matrices,

$$\mathbf{T}_x = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & -1 & 0 \end{bmatrix} \qquad \mathbf{T}_y = \begin{bmatrix} 0 & 0 & -1 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix} \qquad \mathbf{T}_z = \begin{bmatrix} 0 & 1 & 0 \\ -1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}.$$

Since the last two eigenvectors are arbitrary, you have to pick one of these matrices to start with. Just for the sake of argument, let's pick $\mathbf{T}_x$. Then the second eigenmatrix becomes:

$$\mathbf{Q}_1 = \mathbf{U}\mathbf{T}_x \Sigma \mathbf{V}^T \mathbf{A}. \tag{8.17}$$

```
function [lambdas] = Get_Analytic_Eigenvalues(Psi)
  syms mu I5 real;
  lambdas = sym(zeros(3,1));

  firstI5 = diff(Psi, I5);
  secondI5 = diff(firstI5, I5);

  lambdas(1) = 2 * (firstI5 + 2 * I5 * secondI5);
  lambdas(2) = 2 * firstI5;
  lambdas(3) = 2 * firstI5;

  lambdas = simplify(lambdas);
end
```

Figure 8.5.: Matlab/Octave code for obtaining the analytic eigenvalues of an IV_{C}-based anisotropic energy. For reasons that will become clear in the next section, IV_{C} is instead called `I5` in this code.

This effectively pins third eigenmatrix, which can be written in terms of $\mathbf{T}_y$ and $\mathbf{T}_z$,

$$\mathbf{Q}_2 = (\sigma_y \hat{\mathbf{a}}_y)\mathbf{U}\mathbf{T}_z\Sigma\mathbf{V}^T\mathbf{A} - (\sigma_z \hat{\mathbf{a}}_z)\mathbf{U}\mathbf{T}_y\Sigma\mathbf{V}^T\mathbf{A}, \tag{8.18}$$

where $\hat{\mathbf{a}}_{y,z}$ are the corresponding entries from $\hat{\mathbf{a}} = \mathbf{V}^T\mathbf{a}$.

8.2.5 Matlab/Octave Implementation

Code that implements these expressions is given in Fig. 8.5, and examples running them on Ψ_{AS} and Ψ_{Sqrt} are in Fig. 8.6. Using these scripts, we obtain the following analytic eigenvalues for Anisotropic StVK,

$$\lambda_0 = 2\mu(I_5 - 1) + 4\mu I_5 \tag{8.19}$$

$$\lambda_{1,2} = 2\mu(I_5 - 1). \tag{8.20}$$

and for Anisotropic Square Root:

$$\lambda_0 = \mu \tag{8.21}$$

$$\lambda_{1,2} = \mu\left(1 - \frac{1}{\sqrt{I_5}}\right). \tag{8.22}$$

8.3 Better Invariants, Again

8.3.1 An Inversion-Aware Invariant

The energies we've looked all suffer from a version of the problem we saw in §6.1.2. They can't tell if an element has been poked inside-out, i.e. inverted / reversed / prolapsed.

```
% Boilerplate to load up the symbolic package in Octave
isOctave = exist('OCTAVE_VERSION', 'builtin') ~= 0;
if (isOctave)
    pkg load symbolic;
end

% Declare the invariants and constants
syms I5 mu real;

% Build the Anisotropic StVK energy
Psi_AS = (mu / 2) * (I5 - 1)^2;
[lambdas_AS] = Get_Analytic_Anisotropic_Eigenvalues(Psi_AS)

% Build the Anisotropic Sqrt energy
Psi_Sqrt = (mu / 2) * (sqrt(I5) - 1)^2;
[lambdas_Sqrt] = Get_Analytic_Anisotropic_Eigenvalues(Psi_Sqrt)
```

Figure 8.6.: Matlab/Octave example calls to `Get_Analytic_Anisotropic_Eigenvalues` for Ψ_{AS} and Ψ_{Sqrt}.

Like the other Cauchy-Green-style invariants that came before it, $IV_{\mathbf{C}}$ has gone out of its way to throw away any reflection information, so it's not a simple matter of re-jiggering its usage in the energy. I haven't said much about the $V_{\mathbf{C}}$ invariant. Looking at it again, $V_{\mathbf{C}} = \mathbf{a}^T\mathbf{C}^T\mathbf{C}\mathbf{a}$, it just doubles down on the squaring strategy that stomps the information we care about, so it won't help us either. Once again, we need to dream up a new invariant.

Way back in §5.3.3.2, we got a new set of invariants by performing a $\mathbf{F}^T\mathbf{F} \rightarrow \mathbf{S}$ substitution. Then the pesky $\mathbf{F}^T\mathbf{F}$ doesn't get the chance to stomp any reflections. We're going to do the same thing here. Our new anisotropic invariants are:

$$I_4 = \mathbf{a}^T\mathbf{S}\mathbf{a} \tag{8.23}$$

$$I_5 = \mathbf{a}^T\mathbf{S}^T\mathbf{S}\mathbf{a}. \tag{8.24}$$

Similar to §5.3.3.2, we've introduced a new *lower-order* invariant, so some renumbering is needed. The new I_5 invariant is equivalent to $IV_{\mathbf{C}}$[5], but since we need to make room for our new I_4, we're renumbering it to 5. It's not ideal, but the Arabic numeral subscript versus the non-subscripted Roman numeral at least has a distinct visual contrast.

8.3.2 The Gradient of I_4

If we want to use the new I_4 in our energies, we will first need its gradient:

$$\frac{\partial I_4}{\partial \mathbf{F}} = \frac{\partial \mathbf{R}}{\partial \mathbf{F}} : \mathbf{F}\mathbf{a}\mathbf{a}^T + \mathbf{R}\mathbf{a}\mathbf{a}^T. \tag{8.25}$$

[5]Since $IV_{\mathbf{C}} = \mathbf{a}^T\mathbf{F}^T\mathbf{F}\mathbf{a} = \mathbf{a}^T\mathbf{V}\Sigma\mathbf{U}^T\mathbf{U}\Sigma\mathbf{V}^T\mathbf{a} = \mathbf{a}^T\mathbf{V}\Sigma\Sigma\mathbf{V}^T\mathbf{a} = \mathbf{a}^T\mathbf{S}^T\mathbf{S}\mathbf{a} = I_5$

Uh-oh, that $\frac{\partial \mathbf{R}}{\partial \mathbf{F}} : \mathbf{F}\mathbf{a}\mathbf{a}^T$ term doesn't look too friendly. Fortunately, we poured tons of effort into understanding the structure of $\frac{\partial \mathbf{R}}{\partial \mathbf{F}}$ back in §5.4. In 2D, this works out to:

$$\frac{\partial \mathbf{R}}{\partial \mathbf{F}} : \mathbf{F}\mathbf{a}\mathbf{a}^T = \left(\begin{bmatrix} 1 \\ 1 \end{bmatrix}^T \mathbf{V}^T \mathbf{a} \right) \left(\frac{\sigma_x - \sigma_y}{\sigma_x + \sigma_y} \right) \mathbf{U} \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} \mathbf{V}^T. \tag{8.26}$$

There's a lot going on here, but the term to focus on is $\frac{\sigma_x - \sigma_y}{\sigma_x + \sigma_y}$, because as $\sigma_x + \sigma_y \to 0$, the gradient (and forces) will explode to infinity.

Is this some weird corner case that will never actually happen? Or do simulations hit this state all the time, and using this gradient will doom all of our simulations to unexpectedly and spontaneously explode? In order to investigate, I went ahead and coded up this gradient the first time I derived it, and the infinity case was hit on the VERY FIRST SIMULATION I RAN. I ran a few other ones, hoping that maybe I had just gotten unlucky. The explosions occurred with alarming frequency. Clearly, this invariant is too volatile to be used in its raw form.[6]

8.3.3 The Sign of I_4

The information that is discarded by IV_C is whether the element has inverted *in the anisotropy direction*. This distinction is slightly different from the isotropic case. As shown

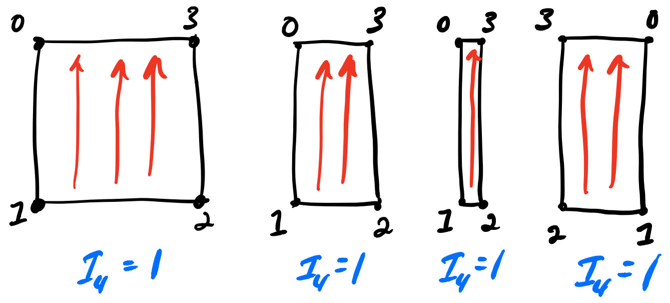


Figure 8.7.: An element inverts, but *not* in the fiber direction, shown in red. Therefore, the value of I_4 remains the same the entire time.

in Fig. 8.7, an element can be inverted, but unless it's inverted *in the fiber direction*, the invariant doesn't, and shouldn't, care about it. It is only when the fiber itself has been pushed inside out (Fig. 8.8) that a negative appears in I_4. What happens if we just try to extract and use *only the sign information* from I_4? Do the gradient and Hessian become less volatile?

[6]As for the Hessian of I_4, let's just say I had to figure out how to print things in landscape mode on legal-sized paper just to look at it. It's not pretty, so let's not even discuss it.

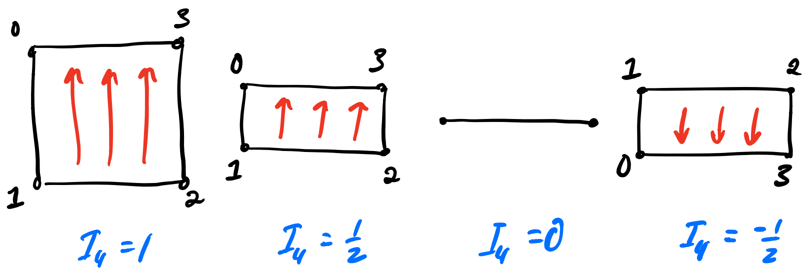


Figure 8.8.: An element inverts in the fiber direction, shown in red. The sign of I_4 then dutifully goes negative.

Let's write a sign-extracting signum function $\mathcal{S}(x)$:

$$\mathcal{S}(x) = \begin{cases} -1 & \text{if } x < 0 \\ 0 & \text{if } x = 0 \\ 1 & \text{if } x > 0 \end{cases} . \tag{8.27}$$

The derivative of $\mathcal{S}(x)$ is a fancy-looking-and-sounding Dirac delta function:[7]

$$\delta(x) = \begin{cases} 0 & \text{if } x \neq 0 \\ \infty & \text{if } x = 0 \end{cases} \tag{8.28}$$

There's still an infinity in there, so things aren't perfect, but this is still a pretty good result. Just to be clear, we are looking at some supremely uncomplicated functions here (Fig. 8.9). The $\mathcal{S}(x)$ is just a step function, and $\delta(x)$ would be the most boring function of all, a line of zeros, if it wasn't for its one spike to infinity. When we wrap I_4 inside $\mathcal{S}(I_4)$,

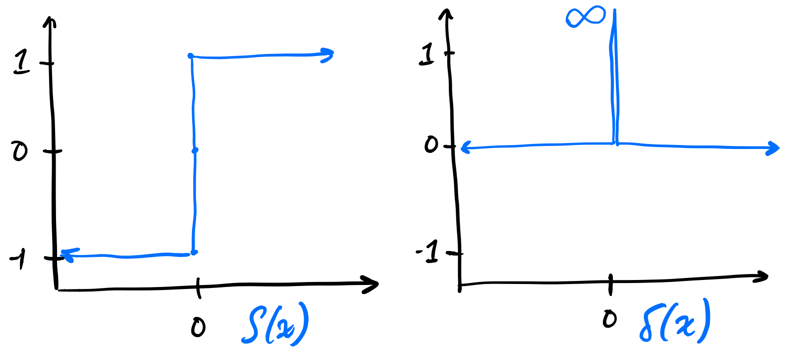


Figure 8.9.: Eqns. 8.27 and 8.28. Not much going on here, except that infinity.

the gradient becomes:

$$\frac{\partial \mathcal{S}(I_4)}{\partial \mathbf{F}} = \delta(I_4)\frac{\partial I_4}{\partial \mathbf{F}}. \tag{8.29}$$

[7]Oooooh Paul Dirac, quantum mechanics, etc.

The volatile $\frac{\partial I_4}{\partial \mathbf{F}}$, rife with infinities, still appears. However, any explosions to infinity are contained by the $\delta(I_4)$ term, which stomps (almost) everything to zero.[8] With this better-behaved result in hand, we can now look at building a better anisotropic energy.

8.4 An Inversion-Aware Anisotropic Energy

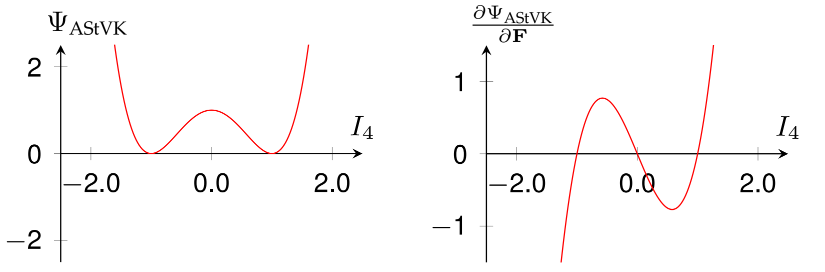
8.4.1 Previous Models, and Their Issues

The square-root model from before had a few attractive properties:

$$\Psi_{\text{ASqrt}} = \frac{\mu}{2}\left(\sqrt{I_5} - 1\right)^2. \tag{8.30}$$

We've now replaced $IV_{\mathbf{C}}$ with I_5 to follow our new numbering scheme, but they're equivalent.

The AStVK model was too non-linear; it raised everything to an unnecessarily high 4thpower, so the forces shoot off the top and bottom of the graph if you try to plot them out (see Fig. 8.10). In addition to introducing really big forces, this will also ruin the conditioning of the global Hessian, and slow down solver convergence. In contrast, the



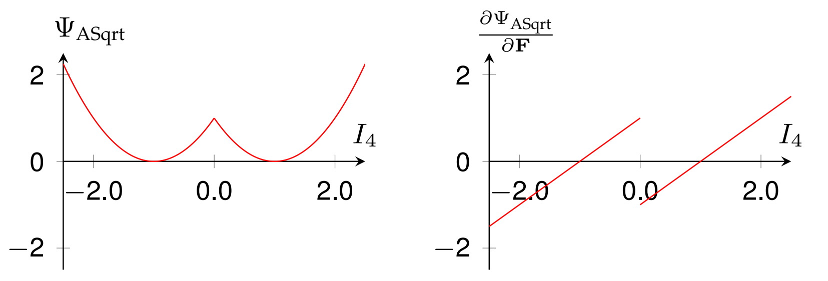
(a) Energy plot, Ψ_{AStVK} (b) PK1 of Ψ_{AStVK}, in the fiber direction

Figure 8.10.: Behavior of the Ψ_{AStVK} model, under stretching and compression.

ASqrt model shows a nearly-linear response in Fig. 8.11. The forces won't grow too large, even if you stretch your objects out really far. The conditioning of the global Hessian will get worse under large stretches, but again, not too quickly. Unfortunately, there is the distressing issue that the energy plot has a kink (Fig. 8.11 (a)) which then makes the force response discontinuous (Fig. 8.11 (b)).

Things are actually worse than that. The PK1 plot for ASqrt contains two roots, at 1 and -1, which means that both are stable states for the energy. If the element is perfectly inverted, i.e. the rest state experiences a perfect reflection in the fiber direction, then the energy will think it is at the rest state, and exert no force.

[8]We could argue about whether the zero should overwhelm the infinity in the limit, but I don't see a practical reason to.



(a) Energy plot, Ψ_{ASqrt} (b) PK1 of Ψ_{ASqrt}, in the fiber direction

Figure 8.11.: Behavior of the Ψ_{ASqrt} model, under stretching and compression.

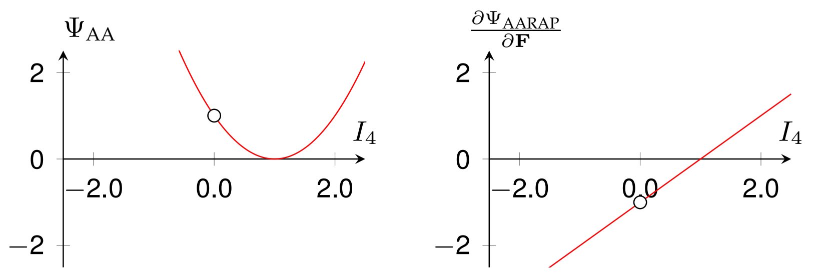
This is the danger of using I_5 instead of I_4: it can't tell the difference between a deformation of 1 (zero deformation) and -1 (perfect reflection) in the fiber direction. So the question is: can we build an anisotropic energy that has all the nice linear-looking properties of ASqrt, but doesn't have the discontinuity issues and spurious stable states?

8.4.2 An Anisotropic ARAP Model

The following energy accomplishes both of these goals:

$$\Psi_{\text{AA}} = \frac{\mu}{2}\left(\sqrt{I_5} - \mathcal{S}(I_4)\right)^2. \tag{8.31}$$

The plots for this new energy is given in Fig. 8.12. It gives a nice, kink-free parabola for the energy, and the PK1 resolves to just a line. In essence, we patched everything to the



(a) Energy plot, Ψ_{AARAP} (b) PK1 of Ψ_{AARAP}, in the fiber direction

Figure 8.12.: Behavior of our Ψ_{AARAP} model, under stretching and compression.

left of $I_4 = 0$ to give us the result we wanted. The only remaining issue is that at $I_4 = 0$, the Dirac delta will produce an an infinity.

Fortunately, by just looking at the plots, we can see that this isn't your usual catastrophic singularity that starts exploding towards infinity if you're anywhere near its neighborhood. It's a *point* singularity that generates a bad value in one specific case, but as long as you don't step directly on it, you're just fine.

A straightforward patch suggests itself almost immediately. For Ψ_{AARAP}, patching in the value of $\Psi_{\text{AARAP}}(2)$ whenever we hit the case of $\Psi_{\text{AARAP}}(0)$ should work just fine. For the PK1, if you ever get unlucky and hit $\frac{\Psi_{\text{AARAP}}(0)}{\partial \mathbf{F}}$ dead-on, then replace the infinity with $-\frac{\Psi_{\text{AARAP}}(2)}{\partial \mathbf{F}}$. The smooth, linear response will then be maintained.

The PK1 of this energy is:

$$\begin{aligned}\frac{\partial \Psi_{\text{AARAP}}}{\partial \mathbf{F}} &= \mu\left(\sqrt{I_5} - \mathcal{S}(I_4)\right)\left[\frac{\partial \sqrt{I_5}}{\partial \mathbf{F}} - \frac{\partial \mathcal{S}(I_4)}{\partial \mathbf{F}}\right] \\ &= \mu\left(\sqrt{I_5} - \mathcal{S}(I_4)\right)\left[\frac{1}{2\sqrt{I_5}}\mathbf{F}\mathbf{a}\mathbf{a}^T - \delta(I_4)\frac{\partial I_4}{\partial \mathbf{F}}\right],\end{aligned} \tag{8.32}$$

but if use the fact that $\delta(I_4)\frac{\partial I_4}{\partial \mathbf{F}} = 0$ almost everywhere, this simplifies to

$$\frac{\partial \Psi_{\text{AARAP}}}{\partial \mathbf{F}} = \frac{\mu}{2}\left(1 - \frac{\mathcal{S}(I_4)}{\sqrt{I_5}}\right)\mathbf{F}\mathbf{a}\mathbf{a}^T. \tag{8.33}$$

Taking this expression forward, the Hessian is then,

$$\frac{\partial^2 \Psi_{\text{AA}}}{\partial \mathbf{f}} = \mu\left[\left(1 - \frac{\mathcal{S}(I_4)}{\sqrt{I_5}}\right)\mathbf{H}_5 + 2\frac{\mathcal{S}(I_4)}{I_5^{3/2}}\mathbf{g}_5\mathbf{g}_5^T\right], \tag{8.34}$$

where, identical to the $IV_{\mathbf{C}}$ case,

$$\mathbf{g}_5 = \operatorname{vec}\left(\frac{\partial I_5}{\partial \mathbf{F}}\right) = \operatorname{vec}\left(2\mathbf{F}\mathbf{a}\mathbf{a}^T\right) \tag{8.35}$$

$$\mathbf{H}_5 = \operatorname{vec}\left(\frac{\partial^2 I_5}{\partial \mathbf{F}^2}\right) = 2\mathbf{a}\mathbf{a}^T \otimes \mathbf{I}. \tag{8.36}$$

If we treat $\mathcal{S}(I_4)$ as a constant, we can push everything through the code in Figs. 8.5 and 8.6 to obtain the eigenvalues

$$\lambda_0 = \mu \tag{8.37}$$

$$\lambda_{1,2} = \mu\left(1 - \frac{\mathcal{S}(I_4)}{\sqrt{I_5}}\right). \tag{8.38}$$

Building a semi-positive definite Hessian is then a matter of checking if $\lambda_{1,2} < 0$. If they are, we only need to build a rank-one matrix using λ_0 and the eigenmatrix $\mathbf{Q}_0 = {}^1/_{\sqrt{I_5}}\mathbf{F}\mathbf{a}\mathbf{a}^T$.

8.4.3 What Other Models Are Out There?

We've only looking at *one* possible anisotropic model that uses the new I_4 invariant. We found that I_4 possesses some difficult properties, so we tried to keep the good while discarding the bad. If you can think of a better one, that's great. Publish it as a paper, post the code publicly, or use it as the secret sauce in your proprietary simulator[9].

[9]Bear in mind that whenever I hear "mine's better, but it's a secret" I usually don't start wondering what the secret is. I start wondering whether the speaker is a dummy, a liar, or both.

Chapter 9

Tips for Computing and Debugging Force Derivatives

Computing the second derivative of our deformation gradient is necessary if we want to implement stable implicit integration methods. Hand rolling the individual terms can be a daunting task. Over the years, I've picked up a few tricks that I thought I'd share. In the presentation I said this wasn't going to be covered in the notes, but I now have the time. I also want to illustrate that I can't be trusted and that you should forever remain a healthy skeptic of the things I say.

Let's imagine we have one of our constitutive models on an element defined over four particles. We've taken the derivative of the deformation gradient F to compute the forces that act on the particles. Taking the derivatives of these forces, denoted f, with respect to their positions $\mathbf{x}_i$ yields the 12×12 Jacobian shown in figure 9.1.

$$
\begin{matrix}
\frac{\partial \mathbf{f}_0}{\partial \mathbf{x}_0} & \frac{\partial \mathbf{f}_0}{\partial \mathbf{x}_1} & \frac{\partial \mathbf{f}_0}{\partial \mathbf{x}_2} & \frac{\partial \mathbf{f}_0}{\partial \mathbf{x}_3} \\
\frac{\partial \mathbf{f}_1}{\partial \mathbf{x}_0} & \frac{\partial \mathbf{f}_1}{\partial \mathbf{x}_1} & \frac{\partial \mathbf{f}_1}{\partial \mathbf{x}_2} & \frac{\partial \mathbf{f}_1}{\partial \mathbf{x}_3} \\
\frac{\partial \mathbf{f}_2}{\partial \mathbf{x}_0} & \frac{\partial \mathbf{f}_2}{\partial \mathbf{x}_1} & \frac{\partial \mathbf{f}_2}{\partial \mathbf{x}_2} & \frac{\partial \mathbf{f}_2}{\partial \mathbf{x}_3} \\
\frac{\partial \mathbf{f}_3}{\partial \mathbf{x}_0} & \frac{\partial \mathbf{f}_3}{\partial \mathbf{x}_1} & \frac{\partial \mathbf{f}_3}{\partial \mathbf{x}_2} & \frac{\partial \mathbf{f}_3}{\partial \mathbf{x}_3}
\end{matrix}
$$

Figure 9.1.: An example 12×12 force Jacobian matrix composed of 3×3 blocks

If you don't take the *"Tensor-Way"* to derive the force Jacobian matrix and instead decide to crank these terms by hand, there are some tricks one can play to reduce the effort. The Jacobian is symmetric, so we only need to compute the upper-triangular blocks and only the upper-triangular portion of the diagonal blocks are needed. To save us even more effort we should remember the property that our forces must sum to zero.

$$\sum_{i=0}^{3} \mathbf{f}_i = 0$$

This means that the column blocks of our Jacobian should sum to a 3×3 matrix of zeros. Again because of our symmetry property, this means that the row blocks should also sum to a zero matrix.[1] One can use this property to check your work for errors. There will likely be some numerical round-off accumulation, so things won't exactly be zero, but if you make a mistake it will likely fail this property in an obvious way.

What else can we do with this knowledge? Well if we have the off-diagonal blocks of our first row, we can compute $\frac{\partial \mathbf{f}_0}{\partial \mathbf{x}_0}$ them them as follows:

$$\frac{\partial \mathbf{f}_0}{\partial \mathbf{x}_0} = -\left[\frac{\partial \mathbf{f}_0}{\partial \mathbf{x}_1} + \frac{\partial \mathbf{f}_0}{\partial \mathbf{x}_2} + \frac{\partial \mathbf{f}_0}{\partial \mathbf{x}_3}\right]$$

Then let's imagine that $\frac{\partial \mathbf{f}_1}{\partial \mathbf{x}_2}$ is has particularly nasty derivatives to perform, but that we have $\frac{\partial \mathbf{f}_1}{\partial \mathbf{x}_1}$ and $\frac{\partial \mathbf{f}_1}{\partial \mathbf{x}_3}$ along with the top row of blocks. We can compute $\frac{\partial \mathbf{f}_1}{\partial \mathbf{x}_2}$ in a similar fashion:

$$\frac{\partial \mathbf{f}_1}{\partial \mathbf{x}_2} = -\left[\frac{\partial \mathbf{f}_0}{\partial \mathbf{x}_1}^T + \frac{\partial \mathbf{f}_1}{\partial \mathbf{x}_1} + \frac{\partial \mathbf{f}_1}{\partial \mathbf{x}_3}\right]$$

I've found that these properties are also a handy debugging tool. We can play similar games with other terms as we work our way down the matrix rows to reduce the amount of work needed by any hand derivations or code implementation. Even if you don't crank the entries of these block terms by hand and go the *"Tensor-Way"* of Chapter 4, you might want to take advantage of our Jacobian properties in another way. Numerical round-off accumulation may result in forces that don't exactly sum to zero and a force Jacobian that is not numerically symmetric. Restoring the property to the forces is straightforward.

$$\mathbf{f}_0 = -\left[\mathbf{f}_1 + \mathbf{f}_2 + \mathbf{f}_3\right]$$

For the force Jacobian blocks we can take a different approach. One can average the off-diagonal blocks and assign the average back to the strictly upper-diagonal block, $i < j$.

$$\frac{\partial \mathbf{f}_i}{\partial \mathbf{x}_j} = \frac{1}{2}\left[\frac{\partial \mathbf{f}_i}{\partial \mathbf{x}_j} + \frac{\partial \mathbf{f}_j}{\partial \mathbf{x}_i}^T\right]$$

One then assigns the transpose of this average to the strictly lower-diagonal block, $j > i$

$$\frac{\partial \mathbf{f}_j}{\partial \mathbf{x}_i} = \frac{\partial \mathbf{f}_i}{\partial \mathbf{x}_j}^T$$

[1]Thanks to Tim Haynes for showing me this property many years ago at Disney's *Secret Lab*.

The diagonal blocks are then defined by the negative sum of the off-diagonal terms in the row.

$$\frac{\partial \mathbf{f}_i}{\partial \mathbf{x}_i} = -\sum_{j}^{i!=j} \frac{\partial \mathbf{f}_i}{\partial \mathbf{x}_j}$$

This will make the Jacobian symmetric. The rows and columns won't sum exactly to zero, but the error is usually reduced. This process is not required for things to work properly in a simulator. However, it can be handy in debugging for the force Jacobian to better reflect the properties that we know it should possess.

9.1 A Warning To Non-Cabal Members

Let's imagine you've derived all the necessary terms for the force Jacobian, verified them with a symbolic math program and then verified you have made absolutely no errors transforming the expressions into code. That is quite an achievement, but unfortunately you have fallen into a trap that has been known by a secret cabal of physics programmers for some time.[2] The right math is wrong! What does that mean? It simply means that the force Jacobian can become indefinite which will likely cause your simulator a problems. In some cases the force Jacobian will be indefinite, but other terms will mask it in the global matrix and you'll be blissfully unware. Other times it will lead to simulation artifacts or your linear system solver failing to converge.

So what can one do? Well, if you go through the rigor of computing the analytic eigensystem as presented in Chapter 7, you don't need to worry. That work pays off and allows one to precisely avoid the force Jacobian from becoming indefinite. If you cranked the individual terms by hand though, you have probably taken a brute-force approach that does not allow for this analysis. In that case you will need to perform some experiments and make some ad-hoc adjustments to the math.[3] I recommend that anyone writing a simulator, that uses the force Jacobian, have a mode that performs an eigendecompostion of it at the element level. Have this capability for all the instances of an implicit force in your simulator! This will help detect these issues and to help to analyze where and when they occur. For the cloth forces in Baraff and Witkin (1998), they occur only in the second term of the equation below from their paper, below:[4]

$$\frac{\partial \mathbf{f}_i}{\partial \mathbf{x}_j} = -k\left[\frac{\partial \mathbf{C}}{\partial \mathbf{x}_i}\frac{\partial \mathbf{C}}{\partial \mathbf{x}_j}^T + \frac{\partial^2 \mathbf{C}}{\partial \mathbf{x}_i \partial \mathbf{x}_j}\right]$$

[2]The first rule of the cabal is to always give credit where credit is due. This is something I had to painfully discover on my own, but over time I learned others had solved discovered this before me. Unfortunately that means I cannot properly uphold the first rule. I hope that I am not banished or worse.

[3]This is the approach taken with the cloth forces in Pixar's simulator *Fizt*.

[4]One should be perform the eigendecomposition with respect to the final sign of the Jacobian term from the left hand side that will need to be inverted.

Sometimes the the second term can be completely omitted. Other times it is necessary to identify the deformation conditions lead to indefiniteness and only discard it then. This was illustrated for the case of a simple linear spring in Choi and Ko (2002). Even when such ad-hoc approaches are employed, the properties of the force Jacobian from before must still hold with respect to the row and column block sums. Besides the effort of deriving the force Jacobian, I suspect the indefiniteness pitfall has lead to many abandon their implementation of Baraff and Witkin (1998) and pivot to either Position Based Dynamics, (PBD), or Extended PBD which do not require the force Jacobian. Since we have lifted the mask on this problem, we will move on to derive the forces and their gradients required to model cloth/thin shells in chapter 10.

Chapter 10

Thin Shell Forces

In this chapter we will derive the forces and derivatives needed to simulate stiff thin shells and cloth. To model cloth/thin shell behavior we need to describe forces models that resist changes to the stretching, shearing and bending modes of deformation. In this chapter we will follow the formulation and notation in Baraff and Witkin (1998) which differs from the tensor formulation we saw earlier[1]. The thin shell model discussed here is certainly not comprehensive. Many other models certainly exist and may have other features to offer.

10.1 Handy derivatives of a few vector operators

Before starting our derivations for the different cloth energy functions, forces and their gradients we present some identities which will prove useful later in our discussion.

10.1.1 Jacobian of a unit vector

Let's define the column vector $x_{ij} = p_j - p_i$ and $\hat{x}_{ij} = \frac{x_{ij}}{\|x_{ij}\|}$.

$$\frac{\partial \hat{x}_{ij}}{\partial p_k} = \frac{1}{\|x_{ij}\|}\left[I - \frac{x_{ij}x_{ij}^T}{\|x_{ij}\|^2}\right]\frac{\partial x_{ij}}{\partial p_k} \tag{10.1}$$

The resulting operator projects out any components of a vector in the direction of $\hat{x}_{ij}$ and scales the components orthogonal to $\hat{x}_{ij}$ by $\frac{1}{\|x_{ij}\|}\frac{\partial x_{ij}}{\partial p_k}$.

[1]If you really enjoy computing force derivatives and ever find yourself needing to name a child, I suggest *Jacob* for the first name, and *Ian* for the middle name. If you really want to commit, you can also change your last name to Matrix. If that sounds like an improbable last name, just remember John and Jenny Matrix, the Arnold Schwarzenegger and Alyssa Milano characters from *Commando* (1985).

10.1.2 The derivative of the dot product of two vectors

Define the scalar $\alpha = x^T y$ where x and y are vector functions of the column vector p_k.

$$\frac{\partial \alpha}{\partial p_k} = \left[\frac{\partial x}{p_k}\right]^T y + \left[\frac{\partial y}{p_k}\right]^T x \tag{10.2}$$

10.1.3 Jacobian of the cross product of two vectors

Define cross product vector $z = x \times y$ where x and y are vector functions of column vector p_k

$$\frac{\partial z}{\partial p_k} = \left[x^{\times}\frac{\partial y}{\partial p_k} - y^{\times}\frac{\partial x}{\partial p_k}\right] \tag{10.3}$$

Here the operator $^{\times}$ means the skew-symmetric matrix that is the cross product operator. For a vector v,

$$v^{\times} = \begin{bmatrix} 0 & -v_z & v_y \\ v_z & 0 & -v_x \\ -v_y & v_x & 0 \end{bmatrix}. \tag{10.4}$$

10.1.4 Energy Functions, Forces and their Jacobians

Baraff and Witkin (1998) define an energy function E for each mode of deformation with a stiffness factor$_ck$s to control how strong the resulting force is that resists the deformation. The constraint function $C(p)$ is a function of the current dynamic particle position state x which designed to be zero when there is no deformation in the given mode.

$$E = \frac{k_s}{2} C(p)^T C(p) \tag{10.5}$$

The restoring force acts opposite the gradient of E and using equation (10.2) is given by (10.6).

$$f_i = -k_s \frac{\partial C(p)}{\partial p_i}^T C(p) \tag{10.6}$$

Our next chapter on implicit integration techniques will reveal that we also require second positional derivative of E given by equation (10.7). This a the source of many pain for many people trying to implement Baraff and Witkin (1998), but done correctly it is worth the effort and in the next section, I'll do all of the heavy lifting for you.

$$\frac{\partial f_i}{\partial p_j} = -k_s \left[\frac{\partial C(p)}{\partial p_i}\frac{\partial C(p)}{\partial p_j}^T + \frac{\partial^2 C(p)}{\partial p_i \partial p_j}\right] \tag{10.7}$$

10.2 Stretch

We are going to assume that our thin shell surface is discretized into triangles. Each triangle vertex/particle has material space coordinates which describe its planar rest shape [2]. Since cloth is cut from bolts of fabric with inherent warp and weft directions, it makes sense to define the material coordinates in 2D *uv* coordinates.

Given a triangle with vertices p_i, p_j and p_k, define the column vectors $\Delta p_1 = p_j - p_i$ and $\Delta p_2 = p_k - p_i$. We then define $\Delta u_1 = u_j - u_i$, $\Delta u_2 = u_k - u_i$, $\Delta v_1 = v_j - v_i$ and $\Delta v_2 = v_k - v_i$.

We then define a linear mapping function from material space to world space $[w_u, w_v]$ in equation (10.8).

$$\begin{bmatrix} w_u \ w_v \end{bmatrix} = \begin{bmatrix} \Delta p1 \ \Delta p2 \end{bmatrix} \begin{bmatrix} \Delta u_1 & \Delta u_2 \\ \Delta v_1 & \Delta v_2 \end{bmatrix}^{-1} \tag{10.8}$$

The lengths $||w_u||$ and $||w_v||$ are unity when no stretching has occurred in the respective material space direction. This gives us what we need to define our function $C(x)$ for stretch shown in equation 10.9[3].

$$C(p) = \begin{bmatrix} ||w_u|| \text{ - } 1 \\ ||w_v|| \text{ - } 1 \end{bmatrix} \tag{10.9}$$

Our stretch energy term is given by (10.10), where k_s is the material stiffness and a_{uv} is the area of the triangle in material space.

$$E_{stretch} = a_{uv} \frac{k_s}{2} C_(p)^T C_(p) \tag{10.10}$$

To compute the restoring force, we are going to need to take the derivative of C with respect to the vertex positions p. We can safely ignore the constant factor because its derivative is zero.

$$\frac{\partial C(p)}{\partial p_i} = \begin{bmatrix} \frac{\partial ||w_u||}{\partial p_{ix}} & \frac{\partial ||w_u||}{\partial p_{iy}} & \frac{\partial c}{\partial p_{iz}} \\ \frac{\partial ||w_v||}{\partial p_{ix}} & \frac{\partial ||w_v||}{\partial p_{iy}} & \frac{\partial ||w_v||}{\partial p_{iz}} \end{bmatrix} \tag{10.11}$$

We can express $||w_u|| = (w_{ux}^2 + w_{uy}^2 + w_{uz}^2)^{\frac{1}{2}}$. To take the derivative with respect to a component c of vertex position p_n, we apply the chain rule to obtain equation (10.12). So

[2]It is important to realize that the representation in material space includes the size of the triangle as well.

[3]In Baraff and Witkin (1998), they define this equation with an area term. This does not result in the proper resolution independent material behavior. Instead we will factor in the area term in the energy function.

it seems like we only need the partials of w_u with respect to our positions now. The same pattern naturally holds true for $\frac{\partial \|w_v\|}{\partial p_{nc}}$.

$$\frac{\partial \|w_u\|}{\partial p_{nc}} = \frac{1}{\|w_u\|} \left[w_{ux} \frac{\partial w_{ux}}{p_{nc}} + w_{uy} \frac{\partial w_{uy}}{p_{nc}} + w_{uz} \frac{\partial w_{ux}}{p_{nc}} \right] \tag{10.12}$$

Let's return back to equation (10.8) and simplify for the terms of w_u and w_v. The inverse of our material matrix is (10.13).

$$\begin{bmatrix} \Delta u_1 & \Delta u_2 \\ \Delta v_1 & \Delta v_2 \end{bmatrix}^{-1} = \frac{1}{\Delta u_1 \Delta v_2 - \Delta u_2 \Delta v_1} \begin{bmatrix} \Delta v_2 & -\Delta u_2 \\ -\Delta v_1 & \Delta u_1 \end{bmatrix} \tag{10.13}$$

Multiplying our position vectors Δp_1 and Δp_2 through we get the forms in (10.14) and (10.15).

$$w_u = \frac{1}{\Delta u_1 \Delta v_2 - \Delta u_2 \Delta v_1} \begin{bmatrix} \Delta p_{1x} \Delta v_2 - \Delta p_{2x} \Delta v_1 \\ \Delta p_{1y} \Delta v_2 - \Delta p_{2y} \Delta v_1 \\ \Delta p_{1z} \Delta v_2 - \Delta p_{2z} \Delta v_1 \end{bmatrix} \tag{10.14}$$

$$w_v = \frac{1}{\Delta u_1 \Delta v_2 - \Delta u_2 \Delta v_1} \begin{bmatrix} \Delta p_{2x} \Delta u_1 - \Delta p_{1x} \Delta u_2 \\ \Delta p_{2y} \Delta u_1 - \Delta p_{1y} \Delta u_2 \\ \Delta p_{2z} \Delta u_1 - \Delta p_{1z} \Delta u_2 \end{bmatrix} \tag{10.15}$$

Examining (10.12) with (10.14) in hand, we can see that only terms inside the brackets that belong to component c will be nonzero. If you are getting tired, there's even more good news! For a fixed p_n the scalar value of the derivatives of $\frac{\partial w_u}{\partial p_{nc}}$ for all components c are the same $\frac{\partial w_{ux}}{\partial p_{ix}} = \frac{\partial w_{uy}}{\partial p_{iy}} = \frac{\partial w_{uz}}{\partial p_{iz}}$

Below are the derivatives of w_u and w_v wrt. positions p_i and p_j.

$$\frac{\partial w_u}{\partial p_i} = \frac{v_j - v_k}{\Delta u_1 \Delta v_2 - \Delta u_2 \Delta v_1} \tag{10.16}$$

$$\frac{\partial w_v}{\partial p_i} = \frac{u_k - u_j}{\Delta u_1 \Delta v_2 - \Delta u_2 \Delta v_1} \tag{10.17}$$

$$\frac{\partial w_u}{\partial p_j} = \frac{v_k - v_i}{\Delta u_1 \Delta v_2 - \Delta u_2 \Delta v_1} \tag{10.18}$$

$$\frac{\partial w_v}{\partial p_j} = \frac{u_i - u_k}{\Delta u_1 \Delta v_2 - \Delta u_2 \Delta v_1} \tag{10.19}$$

$$\frac{\partial w_u}{\partial p_k} = \frac{v_i - v_j}{\Delta u_1 \Delta v_2 - \Delta u_2 \Delta v_1} \tag{10.20}$$

$$\frac{\partial w_v}{\partial p_k} = \frac{u_j - u_i}{\Delta u_1 \Delta v_2 - \Delta u_2 \Delta v_1} \tag{10.21}$$

We can now write our first derivatives of the constraint function C wrt. p_i and p_j, but we will keep the partials separate wrt. the functions in the *u* and *v* directions. Why this is a good idea will become apparent shortly. Below $\hat{w}_u$ and $\hat{w}_v$ represent the unit vectors of w_u and w_v.

$$\begin{aligned} \frac{\partial C_u}{\partial p_i} &= \frac{\partial w_u}{\partial p_i}\hat{w}_u & \qquad \frac{\partial C_v}{\partial p_i} &= \frac{\partial w_v}{\partial p_i}\hat{w}_v \\ \frac{\partial C_u}{\partial p_j} &= \frac{\partial w_u}{\partial p_j}\hat{w}_u & \qquad \frac{\partial C_v}{\partial p_j} &= \frac{\partial w_v}{\partial p_j}\hat{w}_v \\ \frac{\partial C_u}{\partial p_k} &= \frac{\partial w_u}{\partial p_k}\hat{w}_u & \qquad \frac{\partial C_v}{\partial p_k} &= \frac{\partial w_v}{\partial p_k}\hat{w}_v \end{aligned}$$

The partial derivatives wrt. p_k could also be computed by the negative of the sum of the partials wrt. p_i and p_j, but we've included the partial wrt. p_k for completeness.

If we provide different stiffness values k_{su} and k_{sv} the stretch model allows for the modeling of anisotropic behavior in the *u* an *v* directions. The restoring force on particle p_n would be given by equation (10.22) where $C_u = \|w_u\| - 1$ and $C_v = \|w_v\| - 1$.

$$f_n = -a_{uv}\left[k_{su}\frac{\partial C_u}{\partial p_n}C_u + k_{sv}\frac{\partial C_v}{\partial p_n}C_v\right] \tag{10.22}$$

Keeping the terms separate for handling anisotropy is convenient, but there's another more important reason to do this. We have yet to compute the terms necessary for the second term of the force Jacobian in equation (10.7). The term itself is not as intimidating as one might initially think, but it can be a source of pain and agony. Under compression this term can cause the full Jacobian to go indefinite. This can manifest as odd simulation behavior and poor convergence of the linear system discussed as we mentioned in chapter 9. Fortunately by examining C_u and C_v we can see if the triangle element is in compression in the respective direction and if is, we simply omit the corresponding $\frac{\partial^2 C}{\partial p_i \partial p_j}$ term. Let's assume that our triangle element is stretching and proceed to compute the necessary terms.

Given the form of $\frac{\partial C_u}{\partial p_n}$ we can easily compute the second derivative terms by using the identity in (10.1) to arrive at equation (10.23). These 3x3 block matrix terms are symmetric which can be exploited for more efficient computation. The second partial of w_v follows the obvious pattern.

$$\frac{\partial^2 C_u}{\partial p_i \partial p_j} = \frac{\partial w_u}{\partial p_i}\frac{\partial w_u}{\partial p_j}\frac{1}{\|w_u\|}\left[I - \hat{w}_u \hat{w}_u^T\right] \tag{10.23}$$

The full stretch force Jacobian can be written as a sum of the terms of the partials wrt. C_u and C_v.

$$\frac{\partial f_i}{\partial p_j} = -a_{uv} * \left[k_{su}\frac{\partial C_u}{\partial p_i}\frac{\partial C_u}{\partial p_j}^T + k_{sv}\frac{\partial C_v}{\partial p_i}\frac{\partial C_v}{\partial p_j}^T + k_{su}\frac{\partial^2 C_u}{\partial p_i \partial p_j}C_u + k_{sv}\frac{\partial^2 C_v}{\partial p_i \partial p_j}C_v\right] \tag{10.24}$$

We remind the reader that it isn't necessary to compute all of the 3x3 Jacobian blocks. The Jacobian is symmetric, so only the upper or lower triangular blocks are needed [4]. Also, because our internal forces must sum to zero, the same is true for the rows and columns of the force Jacobian. Therefore the 3x3 blocks must sum to zero horizontally and vertically. Only three of the six upper triangular blocks need to be computed and the rest can be obtained from them.

10.2.1 Stretch Damping

Damping forces act opposite the direct of the restoring force and are proportional to the velocity. We define a damping factor $\tilde{k}_d$ which is a material parameter that is scaled by a_{uv}. Following the formulation in Baraff and Witkin (1998) we define the stretch damping force on particle p_i by equation (10.25). $\dot{C}_u = \sum_{n=0}^{2} \frac{\partial C_u}{\partial p_n} \cdot v_n$ where v_n is the velocity of triangle particle p_n.

$$f_{di} = -\tilde{k}_d\left[\frac{\partial C_u}{\partial p_i}\dot{C}_u + \frac{\partial C_v}{\partial p_i}\dot{C}_v\right] \tag{10.25}$$

The corresponding velocity gradient of the damping force on particle i wrt. v_j is in equation (10.26).

$$\frac{\partial f_{di}}{\partial v_j} = -\tilde{k}_d\left[\frac{\partial C_u}{\partial p_i}\frac{\partial C_u}{\partial p_j}^T + \frac{\partial C_v}{\partial p_i}\frac{\partial C_v}{\partial p_j}^T\right] \tag{10.26}$$

In Baraff and Witkin (1998) there's discussion of the spatial gradient of the damping force in equation (10.27). The authors discard the outer product terms because they break the symmetry of the linear system required by the conjugate gradient method they use to solve the sparse linear system needed for implicit integration. The authors also give an

[4] In *Fizt* we favor the upper triangular blocks.

omnious warning about excluding the second partial term. In *Fizt* this term exists, but I am skeptical of its importance[5].

$$\frac{\partial f_{di}}{\partial p_j} = -\tilde{k}_d \left[\frac{\partial C_u}{\partial p_i} \frac{\partial \dot{C}_u}{\partial p_j}^T + \frac{\partial C_v}{\partial p_i} \frac{\partial \dot{C}_v}{\partial p_j}^T + \frac{\partial^2 C_u}{\partial p_i \partial p_j} \dot{C}_u + \frac{\partial^2 C_u}{\partial p_i \partial p_j} \dot{C}_v \right] \tag{10.27}$$

With the formulas in hand one can compute the stretch forces and their Jacobians necessary for implicit integration. Deriving things in the fashion we have done here can seem tedious, but the knowledge we gain from doing so enables us to write very efficient code.

In Kim (2020) an alternate finite element tensor based formulation of the Baraff-Witkin stretch model is provided. Instead of discarding the second partial under compression, an analytic eigenanalysis is performed to guarantee the Jacobian remains positive semidefinite [6].

10.3 Shear

Our vectors w_u and w_v from our stretch formulation remain orthogonal if the triangle has no shearing deformation in world space. In Baraff and Witkin (1998), the energy function presented for shear does not use the unit vectors, which we see as a mistake. In *Fizt* even this formulation is not used. Instead a polar-spring formulation is used to measure the angle between w_u and w_v, with the rest angle being 90°. Here we will present an alternative formulation that has served the authors well elsewhere.

We define the shear constraint function as $C(p) = \hat{w}_u \cdot \hat{w}_v$. We can use equations (10.2) and (10.1) to derive $\frac{\partial C(p)}{\partial p_n}$ for each of the triangle particles.

$$\frac{\partial C(p)}{\partial p_i} = \frac{1}{\|w_u\|} \frac{\partial w_u}{\partial p_i} \left[I - \hat{w}_u \hat{w}_u^T \right]^T \hat{w}_v + \frac{1}{\|w_v\|} \frac{\partial w_v}{\partial p_i} \left[I - \hat{w}_v \hat{w}_v^T \right]^T \hat{w}_u \tag{10.28}$$

Let's examine equation (10.28) in a bit more detail. The matrix inside is symmetric so we can drop the outside transpose. The matrix inside is a projection that removes any component of the vector on the right hand side in the direction of the vector inside the matrix term. $\left[I - \hat{v}_1 \hat{v}_1^T \right] v_2$ removes any component of v_2 in the $\hat{v}_1$ direction. There's no need to assemble the matrices in (10.28). The same operation can be performed by $v_2 - (v_2 \cdot \hat{v}_1)\hat{v}_1$. Let's define the operator $projout(v, u)$ which projects out any component of v in the unit vector u direction. We can then simplify (10.28) to the form in (10.29).

[5]In a different simulator this term was ignored and things worked just fine. Whether they work "better" with the term is now an itch I will need to scratch sometime.

[6]When time permits, I'd like to run a behavior comparison between the analytical eigenanalysis and ad-hoc discarding of the entire second partial for the stretch Jacobian. It would also be interesting to see which formulation results in faster code.

$$\frac{\partial C(p)}{\partial p_i} = \frac{1}{\|w_u\|}\frac{\partial w_u}{\partial p_i} projout(\hat{w}_v, \hat{w}_u) + \frac{1}{\|w_v\|}\frac{\partial w_v}{\partial p_i} projout(\hat{w}_u, \hat{w}_v) \tag{10.29}$$

The same pattern holds for $\frac{\partial C(p)}{\partial p_j}$ and $\frac{\partial C(p)}{\partial p_k}$ and they only differ by the scaling terms $\frac{\partial w_u}{\partial p_n}$ and $\frac{\partial w_u}{\partial p_n}$ leading to efficient evaluation.

With $\frac{\partial C(p)}{\partial p_i}$, $\frac{\partial C(p)}{\partial p_j}$, $\frac{\partial C(p)}{\partial p_k}$ in hand we can compute the shear restoring force for each particle and the outer product term of equation (10.7) for the positional Jacobian blocks we need. Given the form of (10.28), computing the second partial doesn't seem very appealing. Differentiating a matrix wrt. a vector will result in a 3x3x3 tensor. Differentiating wrt. (10.29) seems more appealing, but before we embark on that, let's ask ourselves if it is necessary or even a good idea. It turns out that when there is shearing deformation that the matrix of second partials is always indefinite and for stable implicit integration the second partial isn't required. If we computed the full Jacobian, one option would be to perform an eigendecompostion, clamp any negative eigenvalues to zero and compute the clamped Jacobian from the eigenvectors and clamped eigenvalues. This would not be cheap and performance matters, so we punt on computing the second partial since it is more trouble than it is worth. If you want to add the second partial to your implementation, the second partial for the original Baraff-Witkin shear energy function is provided in Pritchard (2002).

10.3.1 Shear Damping

The formulation of shear damping parallels the one for stretch except for the omission of the second partial term contribution to the positional Jacobian. In practice I have not seen any artifacts from omitting this term. Pseudocode for implementing the stretch

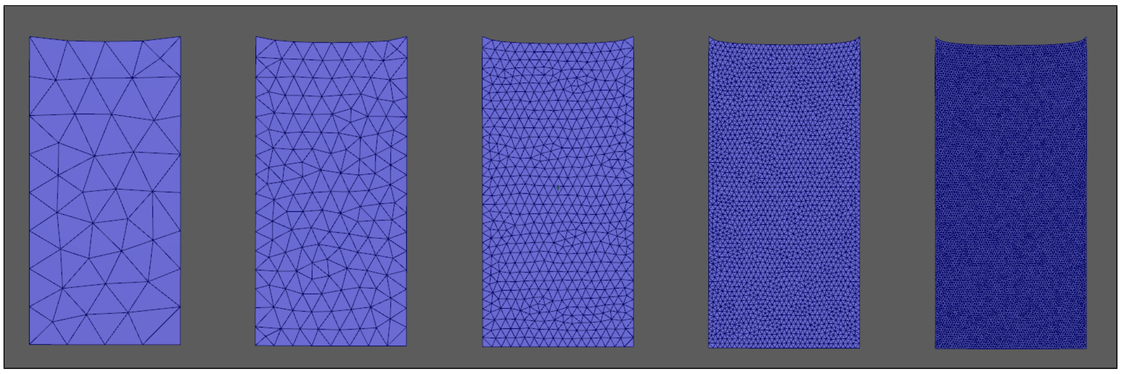


Figure 10.1.: The stretch and shear model from this chapter applied to five sheets of cloth hanging under gravity at different resolutions with the same material parameters.

and shear forces and their Jacobians is provided in the Appendix G. The pseudocode is written for readability and can be improved for performance[7].

As shown in Fig. 10.1, the stretch and shear model presented in this chapter provides fairly mesh resolution independent behavior[8]. In the author's experience one does not obtain this resolution invariant behavior in solvers based on position based dynamics[9].

In this example, density = 0.06 g/cm^2, $KStretchU = KStretchV = 4E5$, $KShear = 5E4$, $KBend = 1E5$. Each sheet is 50cm wide and 100cm tall.

10.4 Dihedral Bend

Thin shell bending forces play an important role in defining the material behavior of a cloth model. The Jacobian of the bending force can be a formidable adversary. Here we present the bending model currently in use in *Fizt*. Our formulation makes no assumptions about material inextensibility such as Garg et al. (2007) and Bergou et al. (2002). We will derive the bending forces and an approximate to the force Jacobian capable of simulating a materials ranging from satin to sheet metal[10]. The formulation presented was derived separately, but seems to parallel the derivation found in Pritchard (2002). It has been reduced to a much more compact form which leads to a very efficient implementation. [11]

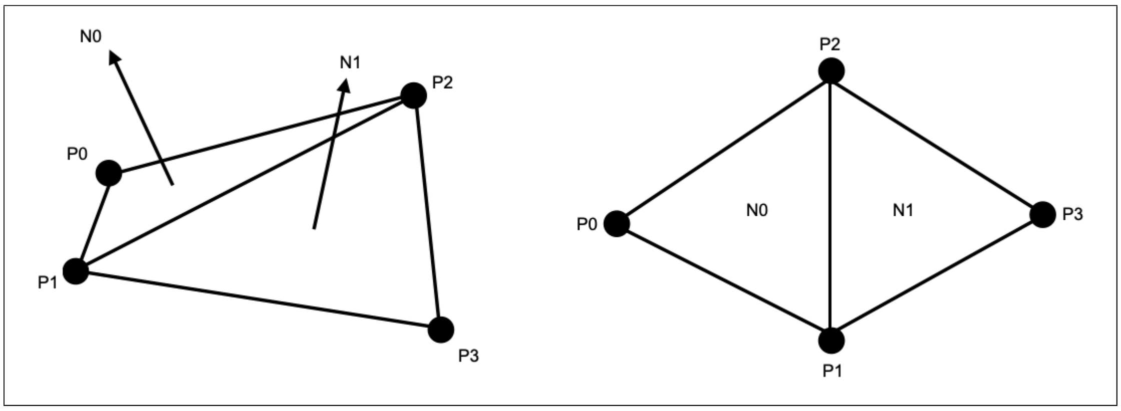


Figure 10.2.: Our four particle configuration for our dihedral edge e_{12}.

Our energy function for bending is given by equation (10.30), where θ is the current

[7]The performance of the routine isn't particularly bad either.

[8]This behavior is also dependent on assigning particle masses as a function of density and the one third of the area in material space of the incident faces.

[9]This can still occur in XPBD solvers, but then this is primarily due to the implementations requiring the user to specify the number of constraint iterations to run each time step, instead of solving a global system to a reasonable degree of convergence

[10]If one has a very high bending stiffness it is important to have stretch and shear forces of similar magnitude. If a face inverts the high bend stiffness will result in a large torque and nasty artifacts such as the mesh eating itself.

[11]At the time of submission and presentation, there were some gross errors in this section and the associated pseudo-code. The mistakes have now been corrected. Apologies, Dave Eberle.

dihedral angle and θ_r is the rest angle. Here, $\tilde{k}_s$ represents the material stiffness as well as a scaling term for the energy that aims to provide mesh resolution independent behavior. A discussion of this term is found in Grinspun et al. (2003). In the case of cloth, θ_r is usually zero, but can me made non-zero to model pleats and permanent folds in a material.

$$E = \frac{\tilde{k}_s}{2}(\theta - \theta_r)^2 \tag{10.30}$$

Following the derivation of the force in Baraff and Witkin (1998), the force is given by:

$$f_{k_s} = -\tilde{k}_s \frac{\partial \theta}{\partial p_k}(\theta - \theta_r) \tag{10.31}$$

Given the unit normals and the edge direction we can compute θ.

$$sin(\theta) = (\hat{n}_0 \times \hat{n}_1) \cdot \hat{e} \tag{10.32}$$

$$cos(\theta) = (\hat{n}_0 \cdot \hat{n}_1) \tag{10.33}$$

Using $tan(\theta) = \frac{sin(\theta)}{cos(\theta)}$ we write θ as equation (10.34).

$$\theta = \arctan(tan(\theta)) \tag{10.34}$$

Taking the derivative with respect to p_k yields:

$$\frac{\partial \theta}{\partial p_k} = \frac{\partial \arctan(tan(\theta))}{\partial p_k} \tag{10.35}$$

Taking the derivative of of left hand side of equation (10.35) with respect to p_k yields:

$$\frac{\partial \theta}{\partial p_k} = \frac{1}{1 + tan^2(\theta)} \frac{\partial tan(\theta)}{\partial p_k} \tag{10.36}$$

Using the quotient rule we can write $\frac{\partial tan\theta}{\partial p_k}$ as equation (10.35).

$$\frac{\partial tan\theta}{\partial p_k} = \frac{\frac{\partial sin(\theta)}{\partial p_k} cos(\theta) - sin(\theta)\frac{\partial cos\theta}{\partial p_k}}{cos^2(\theta)} \tag{10.37}$$

Using equation (10.37) and substituting $1 + tan^2(\theta)$ with $sec^2(\theta)$, equation (10.35) simplifies to (10.38).

$$\frac{\partial \theta}{\partial p_k} = \frac{\partial sin(\theta)}{\partial p_k} cos(\theta) - sin(\theta)\frac{\partial cos(\theta)}{\partial p_k} \tag{10.38}$$

10.4.1 Derivatives of $sin(\theta)$ and $cos(\theta)$

We can use the vector operation Jacobians from the beginning of this chapter to determine the Jacobians of our trigonometric functions.

$$\frac{\partial sin(\theta)}{\partial p_k} = \left[\hat{n}_0^\times \frac{\partial \hat{n}_1}{\partial p_k} - \hat{n}_1^\times \frac{\partial \hat{n}_0}{\partial p_k}\right]^T \hat{e} + \left[\frac{\partial \hat{e}}{\partial p_k}\right]^T (\hat{n}_0 \times \hat{n}_1) \tag{10.39}$$

Remembering that our unit vector Jacobian projects out any components in the direction, positive and negative, in the direction of the unit vector we can safely ignore the the term $\left[\frac{\partial \hat{e}}{\partial p_k}\right]^T (\hat{n}_0 \times \hat{n}_1)$, because it will always be zero.

$$\frac{\partial sin(\theta)}{\partial p_k} = \left[\hat{n}_0^\times \frac{\partial \hat{n}_1}{\partial p_k} - \hat{n}_1^\times \frac{\partial \hat{n}_0}{\partial p_k}\right]^T \hat{e} \tag{10.40}$$

$$\frac{\partial cos(\theta)}{\partial p_k} = \left[\frac{\partial \hat{n}_0}{\partial p_k}\right]^T \hat{n}_1 + \left[\frac{\partial \hat{n}_1}{\partial p_k}\right]^T \hat{n}_0 \tag{10.41}$$

Using our hinge example in Figure 10.2 for reference we can express things in terms of vectors from our particle positions.

$$p_{01} = p_1 - p_0 \quad p_{02} = p_2 - p_0 \tag{10.42}$$

$$p_{32} = p_2 - p_3 \quad p_{31} = p_1 - p_3 \tag{10.43}$$

$$n_0 = p_{01} \times p_{02} \quad n_1 = p_{32} \times p_{31} \quad e = p_2 - p_1 \tag{10.44}$$

The normal vectors are the result of the cross product operator. While one can certainly use equation 10.3 to obtain $\frac{\partial n}{\partial p_k}$ for n_0 and n_1, we derived the following quantities component-wise from the components of the cross products below. Assuming a CCW normal orientation, the skew-symmetric matrices are formed from the edge vector around the triangle opposite the vertex we are differentiating with respect to.

$$\frac{\partial n_0}{\partial p_0} = e^\times \quad \frac{\partial n_0}{\partial p_1} = -p02^\times \quad \frac{\partial n_0}{\partial p_2} = p01^\times \tag{10.45}$$

$$\frac{\partial n_1}{\partial p_3} = -e^\times \quad \frac{\partial n_1}{\partial p_1} = p32^\times \quad \frac{\partial n_1}{\partial p_2} = -p31^\times \tag{10.46}$$

Given the Jacobians of the normal vectors as skew cross product operators, we can also use equation 10.1 to compute the derivatives of our normalized normal vectors.

$$\frac{\partial \hat{n}_i}{\partial p_k} = \frac{1}{\|n_i\|}\left[I - \hat{n}_i \hat{n}_i^T\right] \frac{\partial n_i}{\partial p_k} \tag{10.47}$$

10.4.2 Approximating the force Jacobian

Following the notation of Baraff and Witkin (1998) if $C(p) = \theta$ then

$$\frac{\partial f_i}{\partial p_j} = -\tilde{k}\left[\frac{\partial \theta}{\partial p_i}\frac{\partial \theta}{\partial p_j}^T + \frac{\partial^2 \theta}{\partial p_i \partial p_j}\theta\right] \tag{10.48}$$

Referring to equation 10.38, we currently only have the means to compute the outer product term of equation 10.48. The Hessian term requires significantly more effort and has the potential to make the Jacobian indefinite. The good news is that for practical applications, that only the first term is needed. It is sufficient for stable implicit integration of materials with a wide range of material stiffness[12][13].

10.4.3 Bend Damping

We define the damping force according to Baraff and Witkin (1998). Here $\tilde{k}_d$ is the damping factor which also includes the same scaling term found in $\tilde{k}_s$.

$$f_{k_d} = -\tilde{k}_d \frac{\partial \theta}{\partial p_k}\dot{\theta} \tag{10.49}$$

$$\dot{\theta} = \sum_{i=0}^{3} \frac{\partial \theta}{\partial p_k}^T v_i \tag{10.50}$$

The corresponding damping Jacobian 3×3 block is $\frac{\partial f_i}{\partial v_j} = -\tilde{k}_d \frac{\partial \theta}{\partial p_i}\frac{\partial \theta}{\partial p_j}^T$

10.4.4 Implementation Details

Many of the terms required can be computed once an stored on the faces of the mesh. With a little bookkeeping the terms for a given dihedral edge can be accessed. A similar strategy is discussed in Tamstorf and Grinspun (2013) where an alternative energy function is used and the full Jacobian for it is provided.

A closer inspection of our operators reveals that the additional complexity of such bookkeeping may not be warranted. Using $(A - B)^T x = A^T x - B^T x$ and $(ABC)^T = C^T B^T A^T$, the implementation of equation 10.40 can be simplified. Let $ABC = \hat{n}_0^{\times} \frac{1}{\|n_1\|}\left[I - \hat{n}_1 \hat{n}_1^T\right]\frac{\partial n_1}{\partial p_k}$. Then transpose of our skew symmetric matrices on each end are merely

[12]One may wonder what physical behavior we have given up by discarding the Hessian term. It would be interesting first see if it offers anything using numerical clamping to guarantee positive semidefiniteness. Before going through the considerable effort of computing the Hessian manually for such an experiment, I suggest generating code with a symbolic package.(See next footnote.)

[13]Prof. Kim here. I actually did derive this manually, and it turns out that it's quite similar to the triangle normal Hessians in Appendix I. It's still too preliminary for inclusion here or in the `HOBAK` code, but if you derive it yourself and think "Hey, it's real similar to the collision stuff", I can confirm that you're not wrong.

negations that cancel each other out and our projection matrix represented by B is symmetric. As a result we can express 10.40 as equation 10.51.

$$\frac{\partial sin(\theta)}{\partial p_k} = \left[\frac{1}{\|n_1\|}\frac{\partial n_1}{\partial p_k}\left[I - \hat{n}_1\hat{n}_1^T\right]\hat{n}_0^\times - \frac{1}{\|n_0\|}\frac{\partial n_0}{\partial p_k}\left[I - \hat{n}_0\hat{n}_0^T\right]\hat{n}_1^\times\right]\hat{e} \tag{10.51}$$

This rearranged form provides an efficient way to compute $\frac{\partial sin(\theta)}{\partial p_k}$ with a reduced number vector operations than the original form. Let $proj(v_1, \hat{v}_2) = \left[I - \hat{v}_2\hat{v}_2^T\right]v_1$ which is equivalent to $v_1 - (v_1 \cdot)\hat{v}_2)\hat{v}_2$. We can rewrite 10.51 as 10.52 and see that we can compute the two projections of $\hat{n}_0 \times \hat{e}$ and $\hat{n}_1 \times \hat{e}$ once and then use the results across the computation of all the $\frac{\partial sin(\theta)}{\partial p_k}$ terms.

$$\frac{\partial sin(\theta)}{\partial p_k} = \frac{1}{\|n_1\|}\frac{\partial n_1}{\partial p_k}proj(\hat{n}_0 \times \hat{e}, \hat{n}_1) - \frac{1}{\|n_0\|}\frac{\partial n_0}{\partial p_k}proj(\hat{n}_1 \times \hat{e}, \hat{n}_0) \tag{10.52}$$

The effect of the skew symmetric matrices on both sides can be implemented as cross products. Some care must be taken nest the operations properly. We can also greatly simplify equation 10.41 with a similar manipulation of the transpose and replacing the matrix with our vector projection. In this case the transpose of the skew symmetric matrix is handled by negation. We can write equation 10.53 and see that it too can be compactly reduced to some vector operations. The computation of $\frac{\partial cos(\theta)}{\partial p_k}$ and $\frac{\partial sin(\theta)}{\partial p_k}$ with vector operations is illustrated in the pseudocode found in Appendix H.

$$\frac{\partial cos(\theta)}{\partial p_k} = \frac{-1}{\|n_0\|}\frac{\partial n_0}{\partial p_k}proj(\hat{n}_1, \hat{n}_0) + \frac{-1}{\|n_1\|}\frac{\partial n_1}{\partial p_k}proj(\hat{n}_0, \hat{n}_1) \tag{10.53}$$

Chapter 11

Implicit Integration Methods

11.1 Backward Differentiation Methods in *Fizt*

Robust deformation simulations require stable numerical integration techniques for systems of stiff ordinary differential equations (ODEs). The subject of numerical integration techniques is vast, but in this section we will focus on the backward differentiation formula (BDF) methods used in Pixar's *Fizt*.

The general s-order BDF formula with time step h of the state y is given by:

$$\sum_{i=0}^{s} a_k y_{n+k} = h\beta f(t_{n+s}, y_{n+s}) \tag{11.1}$$

The coefficients a_k and β are then chosen to achieve order s [1].

Given state of our system at t_n is $\mathbf{y}_n = (\mathbf{x}_n, v_n)$, we can write the first-order (BDF-1) form of our system as:

$$\begin{bmatrix} \mathbf{x}_{n+1} - \mathbf{x}_n \\ \mathbf{v}_{n+1} - \mathbf{v}_n \end{bmatrix} = h \begin{bmatrix} \mathbf{v}_{n+1} \\ \mathbf{M}^{-1}\mathbf{f}(\mathbf{x}_{n+1}, \mathbf{v}_{n+1}) \end{bmatrix} \tag{11.2}$$

A reader new to implicit integration might be alarmed by equation (11.2) since its intent is to determine the new state $\mathbf{y}_{n+1} = (\mathbf{x}_{n+1}, \mathbf{v}_{n+1})$, but our forces $\mathbf{f}$ on the right hand side are also being evaluated at this unknown future state. Here is where all the hard work of computing the Jacobians of our forces pays off. We approximate $\mathbf{f}_n(\mathbf{x}_{n+1}, \mathbf{v}_{n+1})$ with a first-order Taylor series approximation:

$$\mathbf{f}(\mathbf{x}_{n+1}, \mathbf{v}_{n+1}) \approx \mathbf{f}(\mathbf{x}_n, \mathbf{v}_n) + \frac{\partial \mathbf{f}}{\partial \mathbf{x}}\Delta\mathbf{x} + \frac{\partial \mathbf{f}}{\partial \mathbf{v}}\Delta\mathbf{v} \tag{11.3}$$

[1]This is done through the use of the Lagrange interpolation polynomial for the points $(\mathrm{t}_n, y_n)...(t_{n+s}, y_{n+s})$

Using this approximation, we can rearrange the terms in equation (11.2) to write a single equation to solve for either $\Delta\mathbf{x}$ or $\Delta\mathbf{v}$. Let's consider the case for $\Delta\mathbf{v}$ first. Using our Taylor approximation, we can write the second row of (11.2) as:

$$\mathbf{M}\Delta\mathbf{v} = h\left[\mathbf{f}(\mathbf{x}_n, \mathbf{v}_n) + \tfrac{\partial \mathbf{f}}{\partial \mathbf{x}}\Delta\mathbf{x} + \tfrac{\partial \mathbf{f}}{\partial \mathbf{v}}\Delta\mathbf{v}\right]. \tag{11.4}$$

Using the first row of (11.2), we can also write $\Delta\mathbf{x} = h[\mathbf{v}_n + \Delta\mathbf{v}]$. Substituting this into (11.4) and grouping the $\Delta\mathbf{v}$ terms to the left hand side, we arrive at:

$$\left[\mathbf{M} - h\tfrac{\partial \mathbf{f}}{\partial \mathbf{v}} - h^2\tfrac{\partial \mathbf{f}}{\partial \mathbf{x}}\right]\Delta\mathbf{v} = h\left[\mathbf{f} + h\tfrac{\partial \mathbf{f}}{\partial \mathbf{x}}\mathbf{v}_n\right]. \tag{11.5}$$

This is the same form used in Baraff and Witkin (1998), and is also known as the Backward-Euler method.

In the next chapter, we will discuss how to efficiently form the linear system for $\Delta\mathbf{v}$. Once $\Delta\mathbf{v}$ is obtained, we use it to obtain $\mathbf{v}_{n+1} = \mathbf{v}_n + \Delta\mathbf{v}$ and update the position state according to the first row of equation (11.2), $\mathbf{x}_{n+1} = \mathbf{x}_n + h\mathbf{v}_{n+1}$.

Now let's consider the case where we solve for $\Delta\mathbf{x}$. First we rearrange the top row of (11.2) into

$$\Delta\mathbf{v} = \frac{1}{h}(\mathbf{x}_n + \Delta\mathbf{x} - \mathbf{x}_n) - \mathbf{v}_n. \tag{11.6}$$

Plugging (11.6) into the second row of (11.2) and using the first-order Taylor approximation of $\mathbf{f}_{n+1}$, we arrive at the system for $\Delta\mathbf{x}$:

$$\left[\mathbf{M} - h\tfrac{\partial \mathbf{f}}{\partial \mathbf{v}} - h^2\tfrac{\partial \mathbf{f}}{\partial \mathbf{x}}\right]\Delta\mathbf{x} = h^2\mathbf{f}_n + h\left[\mathbf{M} - h\tfrac{\partial \mathbf{f}}{\partial \mathbf{v}}\right]\mathbf{v}_n. \tag{11.7}$$

These Backward-Euler (BDF-1) methods provide robust numerical integration in the sense that your simulation won't explode[2]. While Backward-Euler is stable with large time steps h, it is still only a first-order accurate scheme. This impacts the quality of secondary motion in the simulation, because Backward-Euler tends to bleed energy out of the system. One solution is to decrease the time step h, but increasing the number of builds and solves of (11.5) makes this a costly approach.

Fortunately, second-order BDF, also known as BDF-2, offers a more efficient solution that only incurs the additional overhead of tracking the previous system state $\mathbf{y}_{n-1} = (\mathbf{x}_{n-1}, \mathbf{v}_{n-1})$. The second-order BDF form of our system is given by:

$$\begin{bmatrix} \frac{3}{2}\mathbf{x}_{n+1} - 2\mathbf{x}_n + \frac{1}{2}\mathbf{x}_{n-1} \\ \frac{3}{2}\mathbf{v}_{n+1} - 2\mathbf{v}_n + \frac{1}{2}\mathbf{v}_{n-1} \end{bmatrix} = h\begin{bmatrix} \mathbf{v}_{n+1} \\ \mathbf{M}^{-1}\mathbf{f}(\mathbf{x}_{n+1}, \mathbf{v}_{n+1}) \end{bmatrix}. \tag{11.8}$$

We first solve the top row of equation (11.8), for $\mathbf{x}_{n+1}$ to obtain:

$$\mathbf{x}_{n+1} = \frac{4}{3}\mathbf{x}_n - \frac{1}{3}\mathbf{x}_{n-1} + \frac{2}{3}h[\mathbf{v}_n + \Delta\mathbf{v}]. \tag{11.9}$$

[2]This is true if your Jacobians are positive-semidefinite, but many linear system solvers are unable to cope with linear systems that are not positive definite.

We then rearrange (11.9) this further to solve for $\Delta \mathbf{x}$:

$$\Delta \mathbf{x} = \frac{1}{3}[\mathbf{x}_n - \mathbf{x}_{n-1}] + \frac{2}{3}h[\mathbf{v}_n + \Delta \mathbf{v}]. \tag{11.10}$$

Substituting (11.10) into our first-order Taylor approximation of $\mathbf{f}_{n+1}$, and then grouping terms with $\Delta \mathbf{v}$ to the left hand side, produces the linear system for the BDF-2 scheme:

$$\left[\mathbf{M} - \frac{2}{3}h\frac{\partial \mathbf{f}}{\partial \mathbf{v}} - \frac{4}{9}h^2\frac{\partial \mathbf{f}}{\partial \mathbf{x}}\right]\Delta \mathbf{v} = \frac{1}{3}\mathbf{M}\left[\mathbf{v}_n - \mathbf{v}_{n-1}\right] + \frac{2}{3}h\left[\mathbf{f}(\mathbf{x}_n, \mathbf{v}_n) + \frac{1}{3}\frac{\partial \mathbf{f}}{\partial \mathbf{x}}\left[\mathbf{x}_n - \mathbf{x}_{n-1} + 2h\mathbf{v}_n\right]\right] \tag{11.11}$$

The terms on the right hand side are grouped to provide a single sparse matrix-vector multiply against the terms of $\frac{\partial \mathbf{f}}{\partial \mathbf{x}}$. Once again, we then solve for $\Delta \mathbf{v}$ and update $\mathbf{v}_{n+1} = \mathbf{v}_n + \Delta \mathbf{v}$. The position update follows similarly from the first row of (11.8): $\mathbf{x}_{n+1} = \frac{4}{3}\mathbf{x}_n - \frac{1}{3}\mathbf{x}_{n-1} + \frac{2}{3}h\mathbf{v}_{n+1}$.

If we instead want to solve for $\Delta \mathbf{x}$, we can rearrange the top row of (11.8) into $\Delta \mathbf{x} = \frac{1}{h}[\frac{3}{2} - 2\mathbf{x}_n + \frac{1}{2}\mathbf{x}_{n-1}] - \mathbf{v}_n$. Substituting this into our first-order Taylor approximation and pushing the $\Delta \mathbf{x}$ terms from the bottom row of (11.8) to the left side yields the somewhat lengthy:

$$\left[\mathbf{M} - \frac{2}{3}h\frac{\partial \mathbf{f}}{\partial \mathbf{v}} - \frac{4}{9}h^2\frac{\partial \mathbf{f}}{\partial \mathbf{x}}\right]\Delta \mathbf{x} = \frac{\mathbf{M}}{3}\left[(\mathbf{x}_n - \mathbf{x}_{n-1}) + \frac{1}{3}h(8\mathbf{v}_n - 2\mathbf{v}_{n-1})\right] + \frac{2}{9}h\left[2h\mathbf{f}_n - \frac{\partial \mathbf{f}}{\partial \mathbf{v}}\left[\mathbf{x}_n - \mathbf{x}_{n-1} + 2h\mathbf{v}_n\right]\right]. \tag{11.12}$$

BDF-2 integration in terms of $\Delta \mathbf{v}$ was deployed in Fizt for Pixar's *Coco* and has been the default integration scheme ever since. Taking 10 time steps per frame, with a frame representing $\frac{1}{24}$ of a second, produces enough quality for our secondary motion needs.

You might be wondering: if BDF-2 provides such a nice accuracy boost, why not go to even higher-order? This is certainly possible, but could spell trouble, as higher-order BDF methods unfortunately do not have the same stability guarantees. A nice discussion of this can be found in Hauth and Etzmuss (2001). The presented BDF-2 forms also assume a constant time step. If the time step changes, the coefficients are different, and must be computed as a function of h_{curr} and h_{prev}.[3] More recently, higher-order implicit integration through a diagonalized implicit Runge-Kutta has been explored by Löschner et al. (2020).

Besides the accuracy and order of the implicit technique, other factors can affect the quality of motion. If an iterative method is used to solve for $\Delta \mathbf{v}$, the accuracy of the solution also has an impact. Since our Taylor approximation to $\mathbf{f}(\mathbf{x}_{n+1}, \mathbf{v}_{n+1})$ was only first- order accurate, we are making a linear approximation of our forces. To obtain a more accurate solution, one can employ Newton-Raphson iteration, where each iteration requires the assembly and solution of a new linear system. This strategy works well if one wants to produce quality motion and only take one step per-frame. However, the simple approach of using BDF-2, ten timesteps per frame, and a single Newton iteration at each step, has served us well in practice.

[3]Details are in Hauth and Etzmuss (2001)

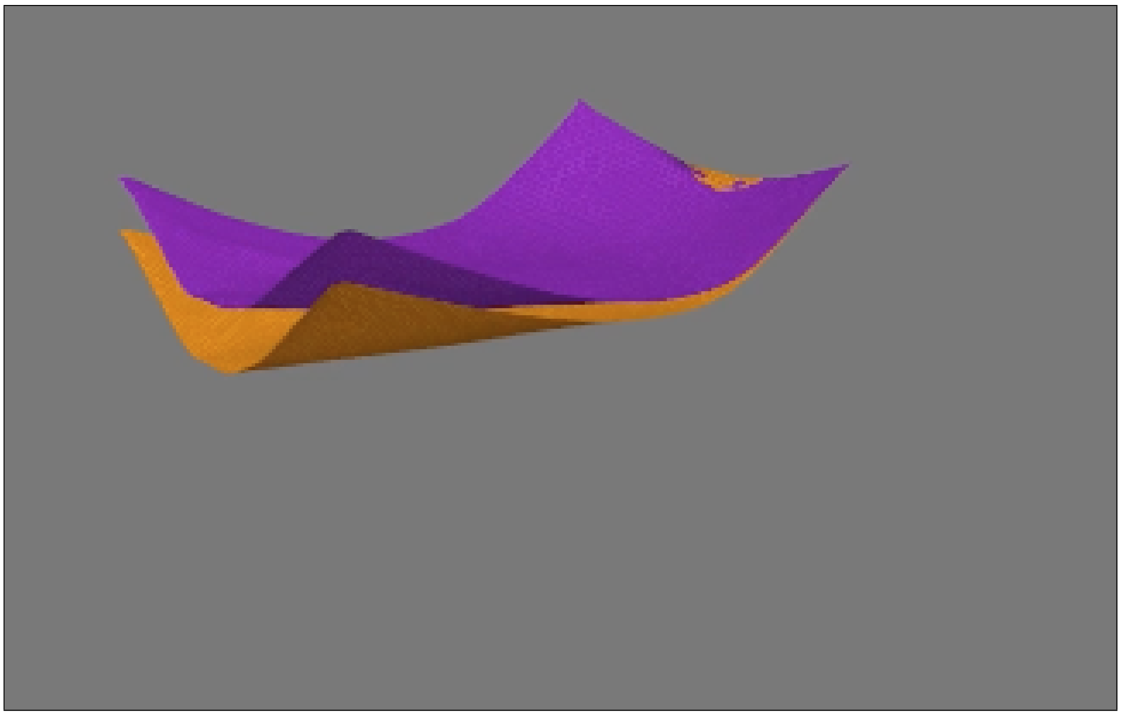


Figure 11.1.: Two sheets of cloth from separate simulators are shown at their apex from swinging under gravity. The BDF-2 result in purple retains more energy and swings higher than Backward-Euler show in orange.

11.2 Time Integration in HOBAK

11.2.1 Velocity-Based BDF-1

Now let's take a look at BDF-1 in the HOBAK code, in particular in

`BACKWARD_EULER_VELOCITY::solveRayleighDamped`.

The velocity-based BDF-1 problem $\mathbf{Ax} = \mathbf{b}$ is implemented as:

```
// assemble RHS from Eqn. 18 in [BW98]
_b = _dt * (R + _dt * K * _velocity + _externalForces);

// assemble system matrix A, LHS from Eqn. 18 in [BW98], but
// we negate the K term because of how hyperelastic energies work
_A = _M - _dt * C - _dt * _dt * K;
```

This matches the linear system from Eqn. (11.5)

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

but we have made the substitutions $h = \_dt$, $\mathbf{K} = \frac{\partial \mathbf{f}}{\partial \mathbf{x}}$ and $\mathbf{C} = \frac{\partial \mathbf{f}}{\partial \mathbf{v}}$, and external forces have been folded into $\mathbf{f}$.

11.2.2 Position-Based BDF-1

In the HOBAK code, specifically in

```
BACKWARD_EULER_POSITION::solveRayleighDamped
```

the position-based BDF-1 is implemented as:

```
const REAL invDt = 1.0 / _dt;
const REAL invDt2 = invDt * invDt;
_b = (invDt * _M - C) * _velocity + R + _externalForces;
_A = _M * invDt2 - (C + collisionC) * invDt - K; // w/ Rayleigh
```

This yields the exact system that we saw in (11.7)

$$\begin{aligned}\mathbf{A} &= \frac{1}{\Delta t^2}\mathbf{M} - \frac{1}{\Delta t}\mathbf{C} - \mathbf{K}\\ \mathbf{b} &= \mathbf{f}^{\text{int}} + \mathbf{f}^{\text{ext}} + \frac{1}{\Delta t}[\mathbf{M} - \Delta t\mathbf{C}]\mathbf{v}_t.\end{aligned}$$

11.2.3 Newmark and Newton-Raphson

As a bonus, let's take a look at Newmark integration from within a full-on Newton-Raphson solver. This one tends to be slightly livelier than the BDF schemes, but does **not have the same stability guarantees**, which is one of the main reasons we prefer BDF methods in production. However, we provide it here for research testing purposes.

In this case we're on a root-finding mission where we're trying to get some function to converge to zero. Let's call the the function $\mathbf{h}(\mathbf{x})$, since $\mathbf{f}(\mathbf{x})$ is already claimed by *force*. We want to drive it to zero:

$$\begin{aligned}\mathbf{h}(\mathbf{x}_{n+1}) &= -\mathbf{f}^{\text{ext}}\\ \mathbf{h}(\mathbf{x}_{n+1}) + \mathbf{f}^{\text{ext}} &= 0\end{aligned}$$

First we apply a Taylor truncation:

$$\begin{aligned}\mathbf{x}_{r+1} &= \mathbf{x}_r - \Delta\mathbf{x}\\ \mathbf{h}(\mathbf{x}_{r+1}) + \mathbf{f}^{\text{ext}} &\approx \mathbf{h}(\mathbf{x}_r) - \frac{\partial \mathbf{h}}{\partial \mathbf{x}_r}\Delta\mathbf{x} + \mathbf{f}^{\text{ext}}\\ \mathbf{h}(\mathbf{x}_r) - \frac{\partial \mathbf{h}}{\partial \mathbf{x}}\Delta\mathbf{x} + \mathbf{f}^{\text{ext}} &= 0\\ \frac{\partial \mathbf{h}}{\partial \mathbf{x}}\Delta\mathbf{x} &= \mathbf{f}^{\text{ext}} + \mathbf{h}(\mathbf{x}_r).\end{aligned}$$

Here, the subscript r has been used in lieu of n because it denotes Newton-Raphson iterations, not the next timestep. The iteration is initialized with $\mathbf{x}_r = \mathbf{x}_n$

Let's try this out with position-based Newmark. In this case, we have the updates for position:

$$\begin{aligned}\mathbf{v}_{n+1} &= \alpha_3(\mathbf{x}_{n+1} - \mathbf{x}_n) + \alpha_4\mathbf{v}_n + \alpha_5\mathbf{a}_n \\ \mathbf{a}_{n+1} &= \alpha_0(\mathbf{x}_{n+1} - \mathbf{x}_n) - \alpha_1\mathbf{v}_n - \alpha_2\mathbf{a}_n,\end{aligned}$$

Substituting into our force equation yields:

$$\begin{aligned}\mathbf{M}\mathbf{a}_{n+1} - \mathbf{C}\mathbf{v}_{n+1} - \mathbf{f}^{\text{int}}(\mathbf{x}_{n+1}) &= -\mathbf{f}^{\text{ext}} \\ \mathbf{M}\left(\alpha_0(\mathbf{x}_{n+1} - \mathbf{x}_n) - \alpha_1\mathbf{v}_n - \alpha_2\mathbf{a}_n\right) - \mathbf{C}\left(\alpha_3(\mathbf{x}_{n+1} - \mathbf{x}_n) + \alpha_4\mathbf{v}_n + \alpha_5\mathbf{a}_n\right) - \mathbf{f}^{\text{int}}(\mathbf{x}_{n+1}) &= -\mathbf{f}^{\text{ext}}\end{aligned}$$

We can plug in our current Newton-Raphson iterate $\mathbf{x}_r = \mathbf{x}_{n+1}$ to clarify things slightly:

$$\mathbf{M}\left(\alpha_0(\mathbf{x}_r - \mathbf{x}_n) - \alpha_1\mathbf{v}_n - \alpha_2\mathbf{a}_n\right) - \mathbf{C}\left(\alpha_3(\mathbf{x}_r - \mathbf{x}_n) + \alpha_4\mathbf{v}_n + \alpha_5\mathbf{a}_n\right) - \mathbf{f}^{\text{int}}(\mathbf{x}_r) = -\mathbf{f}^{\text{ext}} \tag{11.13}$$

Relating things back to the generic form yields:

$$\begin{aligned}\mathbf{h}(\mathbf{x}_r) &= \mathbf{M}\left(\alpha_0(\mathbf{x}_r - \mathbf{x}_n) - \alpha_1\mathbf{v}_n - \alpha_2\mathbf{a}_n\right) - \mathbf{C}\left(\alpha_3(\mathbf{x}_r - \mathbf{x}_n) + \alpha_4\mathbf{v}_n + \alpha_5\mathbf{a}_n\right) - \mathbf{f}^{\text{int}}(\mathbf{x}_r) \\ \frac{\partial\mathbf{h}(\mathbf{x}_r)}{\partial\mathbf{x}_r} &= \mathbf{M}\alpha_0 - \mathbf{C}\alpha_3 - \mathbf{K}_r\end{aligned}$$

where $\mathbf{K}_r = \frac{\partial\mathbf{f}^{\text{int}}(\mathbf{x}_r)}{\partial\mathbf{x}_r}$. Plugging back into the Newton-Raphson process gives:

$$\begin{aligned}\frac{\partial\mathbf{h}}{\partial\mathbf{x}}\Delta\mathbf{x} &= -\mathbf{f}^{\text{ext}} - \mathbf{h}(\mathbf{x}_r) \\ (\alpha_0\mathbf{M} - \alpha_3\mathbf{C} - \mathbf{K}_n)\,\Delta\mathbf{x} &= -\mathbf{f}^{\text{ext}} + \mathbf{M}\left(\alpha_0(\mathbf{x}_r - \mathbf{x}_n) - \alpha_1\mathbf{v}_n - \alpha_2\mathbf{a}_n\right) - \\ &\quad \mathbf{C}\left(\alpha_3(\mathbf{x}_r - \mathbf{x}_n) + \alpha_4\mathbf{v}_n + \alpha_5\mathbf{a}_n\right) - \mathbf{f}^{\text{int}}(\mathbf{x}_r)\end{aligned}$$

This is then implemented in `NEWARK::solveRayleighDamped` as:

```
_temp = _alpha[0] * (_position - _positionOld) - _alpha[2] * _accelerationOld
     -_alpha[1] * _velocityOld;
_temp = _M * _temp;

// compute the RHS of the residual:
// C (alpha_4 (q_{i+1} - q_{i}) + alpha_5 q^{dot}_{i} + alpha_6(q^{dot dot}_{
    i}))
_b = _alpha[3] * (_position - _positionOld) + _alpha[4] * _velocityOld +
    _alpha[5] * _accelerationOld;
_b = C * (-1.0 * _b);

// assemble full residual: LHS + RHS + R - F
_b += _temp - R - _externalForces;

// assemble system matrix A
_A = _alpha[0] * _M - _alpha[3] * C - K;
```

Go peek at `HOBAK/src/Timestepper/NEWMARK.cpp` to see how this is then solved multiple times to convert to a solution. If you peek at `HOBAK/src/Timestepper/BDF_1.cpp` and `HOBAK/src/Timestepper/BDF_2.cpp`[4], you will also see versions of BDF and BDF-2 that take multiple Newton-Raphson steps. These are provided for illustrative and research purposes only; they are not the methods used in production.

[4]Without line search. Still need to add that.

Chapter 12

Constrained Backward-Euler

Large Steps in Cloth Simulation, Baraff and Witkin (1998), ushered in a new era of deformable simulation by demonstrating the stability benefits of implicit integration. Despite the time that has passed, it remains the foundation of Pixar's cloth and volume simulator, called *Fizt*. Things in this chapter will be discussed in terms of Backward-Euler or BDF-1 from chapter 11, but generalize to the BDF-2 method as well[1].Using Backward-Euler we assemble and solve the linear system below at each time step:[2]

$$\left[\mathbf{M} - h\frac{\partial \mathbf{f}}{\partial \mathbf{v}} - h^2\frac{\partial \mathbf{f}}{\partial \mathbf{x}}\right]\Delta\mathbf{v} = h\mathbf{f}_n + h^2\frac{\partial \mathbf{f}}{\partial \mathbf{x}}\mathbf{v}_n,$$

$$\mathbf{A}\mathbf{x} = \mathbf{b}.$$

This linear system is sparse, but can still be quite large in practice. We represent our system in a Block Compressed Row Storage format (BCRS) with 3×3 blocks. We use Preconditioned Conjugate Gradient (PCG) to solve our linear system, but handle hard constraints different from the original paper, which we will discuss shortly. We use hard constraints where two-way coupling is not needed. Some attachments and collisions with character geometry, or other kinematic objects, fall into this category. We use penalty forces to handle attachment and collision between dynamic surfaces for two-way coupling, as did the original paper. Later in this chapter, we will also discuss techniques we have found useful to increase performance of the assembly and solution of this system despite its changing structure.

12.1 Constraint Filtering in Baraff-Witkin Cloth

In this section, we will give a brief overview of the constraint mechanism described in Baraff and Witkin (1998). To precisely control the motion of a particle in any of its Degrees

[1]The equation for Backward-Euler is more compact and convenient for our discussion.

[2]We only take one Newton step per time step for performance concerns and typically take ten steps per frame for motion quality.

of Freedom (DOFs), a mass modification scheme was developed. Given the current particle state $(\mathbf{x}_n, \mathbf{v}_n)$, one can easily compute the required $\Delta\mathbf{v}$ to reach a target velocity or position state at the end of the time step. We project out any free DOFs from $\Delta\mathbf{v}$ and following the convention of the original paper, we label this quantity $\mathbf{z}$. To enforce the constraint during the solution of our linear system, the concept of per-particle constraint filters was introduced. Each particle has a filter $\mathbf{S}_i$ with its constrained DOFs as follows:[3]

$$\mathbf{S}_i = \begin{cases} \mathbf{I} & \text{if zero DOFs are constrained} \\ \mathbf{I} - \hat{\mathbf{p}}_i\hat{\mathbf{p}}_i^T & \text{if one DOF } (\hat{\mathbf{p}}_i) \text{ is constrained} \\ \mathbf{I} - \hat{\mathbf{p}}_i\hat{\mathbf{p}}_i^T - \hat{\mathbf{q}}_i\hat{\mathbf{q}}_i^T & \text{if two DOFs } (\hat{\mathbf{p}}_i \text{ and } \hat{\mathbf{q}}_i) \text{ are constrained} \\ 0 & \text{if all three DOFs are constrained} \end{cases} \tag{12.1}$$

These filters are applied within the inner loop of of the original *Fizt* solver. At each iteration the solution is projected onto a state which satisfies the constraints, but nothing about how this projection might impede the convergence of the solver was considered.

12.1.1 Pre-filtered Preconditioned Conjugate Gradient (PPCG)

In a later version, creatively named *Fizt2*, we adopted the Pre-filtered Preconditioned Conjugate Gradient (PPCG) method presented in Tamstorf et al. (2015). We are only using the pre-filtering component from §8 in the paper, which already gives a significant speedup. We will touch on preconditioning further in §12.7.

The PPCG formulation eliminates the need to apply the filters in the inner loop of PCG. Unlike the method of Ye (2009), the PPCG method does not change the size of our linear system, which enables us to maintain certain aspects of our system data structures. PPCG also makes it more convenient to send our system to a direct solver like CHOLMOD or PARDISO. We use a direct solver as fallback to handle severely ill-conditioned systems.

The formulation is:

$$\left(\mathbf{SAS}^T + \mathbf{I} - \mathbf{S}\right)\mathbf{y} = \mathbf{Sc} \tag{12.2}$$

$$\mathbf{y} = \mathbf{x} - \mathbf{z} \tag{12.3}$$

$$\mathbf{c} = \mathbf{b} - \mathbf{Az}. \tag{12.4}$$

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 $\mathbf{S}$.[4] 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.

[3] In chapters 10 and 11, we overload the use of the symbol ˆ to describe a unit vector, where in chapter 4 it was used to describe the cross-product matrix of a vector.

[4] Since $\mathbf{S}$ is symmetric, we could just write $\mathbf{SAS}$, but the transpose makes the projection a little more visually obvious.

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 $\mathbf{S}_i = 0$. Now we are in trouble because we have a 3×3 block of zeros along the diagonal. The Gerschgorin 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 ($r = \sum_{j,j \neq i} |a_{ij}|$). 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 $\mathbf{I} - \mathbf{S}_i$ fix will paste an identity matrix precisely on top of that zero block, and replace those zeros along the diagonal with ones. The Gerschgorin discs are now centered around one, and eigenvalues will be near one, which is a much better state.

Example	Prefiltered PCG	Modified PCG
Miguel	4.05×	2.09×
Nana	2.95×	2.03×
Dress	1.95×	1.55×
Hector	2.09×	1.12×
Mama	2.92×	1.55×
Average	2.79× faster	1.67× faster

Figure 12.1.: Speedup of solver stage using Prefiltered PCG from Tamstorf et al. (2015), without multigrid, and Modified PCG from Ascher and Boxerman (2003) over the original Baraff and Witkin (1998) algorithm.

As observed by the authors (page 9, first column, second paragraph), boosting the filtered subspace in this way improves the conditioning of the matrix, so PPCG should converge even faster than the MPCG method in Ascher and Boxerman (2003). We found this to be true in our tests as well. PPCG runs 2.79× faster than the original Baraff and Witkin (1998) solve, while Ascher and Boxerman (2003) only runs $1.67\times$ faster (Table 12.1). PPCG gives a 36.0% wall-clock improvement over the 12.6% improvement achieved by Ascher and Boxerman (2003) (Table 12.2). Again, this 2.79× speedup was achieved *even without the multigrid component* of the paper.

Finally, we performed a performance breakdown of *Fizt* using the PPCG formulation (Table 12.3). While we had become accustomed to seeing a roughly even split between the solver, matrix assembly, and collision processing (i.e. CCD) stages, we found that the collision stage was now the tallest nail.

The *wrong* conclusion here would be that, since the solver only takes up 16.5% of the running time, further solver research is now unnecessary, and we should instead pour all of our research effort into accelerating collisions. This is only *one performance snapshot* circa 2015, and it suggested at the time that a short-term effort into squeezing extra performance out of the collision code might be a good idea. The numbers have shifted

Example	Prefiltered PCG	Modified PCG
Miguel	53.9%	17.0%
Nana	29.1%	10.5%
Dress	39.4%	13.9%
Hector	17.0%	-1.0%
Mama	40.9%	22.7%
Average	36.0% faster	12.6% faster

Figure 12.2.: Overall wall-clock performance improvements of cloth solver using Prefiltered PCG from Tamstorf et al. (2015), without multigrid, and Modified PCG from Ascher and Boxerman (2003) over the original Baraff and Witkin (1998) algorithm. Numbers include system assembly and collision processing.

since then, but this snapshot is still interesting, as long as you do not extrapolate too much from them.

Example	PPCG Solve	Matrix Assembly	Collisions
Miguel	15.9%	32.6%	51.5%
Nana	8.1%	25.4%	66.5%
Dress	30.8%	31.4%	37.8%
Hector	8.27%	13.0%	78.0%
Mama	19.7%	30.8%	49.5%
Average	16.5%	26.6%	56.5%

Figure 12.3.: Simulator time breakdown, circa 2015. Our conclusion was that the collision stage was now the ripest target for optimization, *not* that the PCG component is now somehow *solved forever* and that the focus their research on collision acceleration.

Our main conclusion is that when implementing a Baraff and Witkin (1998)-type solver, the current best practice is to use the PPCG constraint-filtering formulation from Tamstorf et al. (2015), even if you do not end up using all their multigrid components.

12.2 Performance Improvement Techniques

System assembly and solution are critical to overall simulator performance. Parallel implementations of both are essential problem sizes we usually run with *Fizt*. We find that we are not compute bound on modern CPU architectures, but instead limited by memory bandwidth. Therefore, reducing memory traffic is a worthy avenue of pursuit for gaining performance. In the following sections, we will discuss some techniques that we have used to improve the performance of system assembly and our PPCG solver.

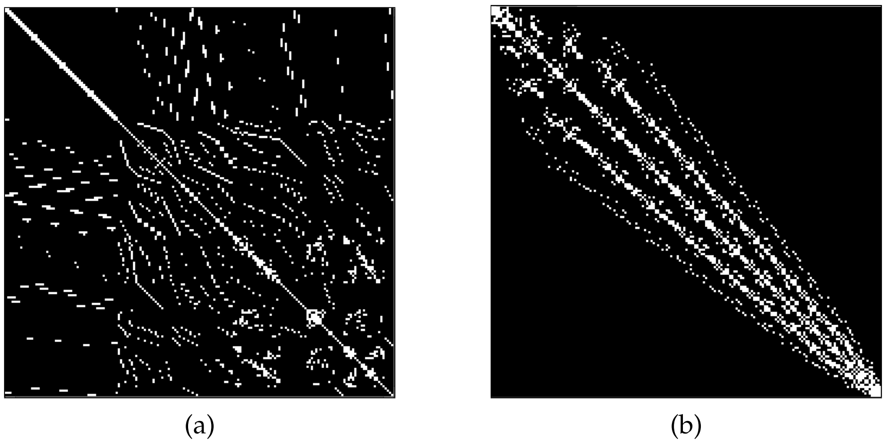


Figure 12.4.: Sparsity pattern: (a)Default mesh ordering (b) After Reverse Cuthill-McKee

12.3 Reverse Cuthill-McKee

At every iteration of the PPCG solver, we multiply our matrix **A** with a vector. It is trivial to parallelize this operation over the rows of **A** which does not require any locking. Since **A** is sparse, the multiplication of the row blocks with a vector does not access the memory of the vector uniformly. We would like to reduce the amount of random access to the vector during this operation.

One way to reduce random access is to reorder the particles in a way that minimizes minimizes the matrix bandwidth. The Boost Graph Library has a number of reordering strategies, but Reverse Cuthill-McKee has worked well for our purposes. At the beginning of our simulation, we make a graph of the fixed system connectivity. Since *Fizt* does not do adaptive re-meshing or tearing, we can exploit the fixed structure of any cloth or volume meshes in the system. Any permanent attachments, such as springs, between dynamic objects are also considered. The graph edges are made from the ij entries above the diagonal of our global Jacobian matrix. One iterate through all the fixed elements and create a unique collection of these edges and use an initial global ordering of the particles. Once the reordering is performed, we assign each simulated particle an index from this reordering which we call the *global solver index*. Local element and orderings remain unchanged. The *global solver index* now indicates the row index of the particle in our linear system.

12.4 System Assembly

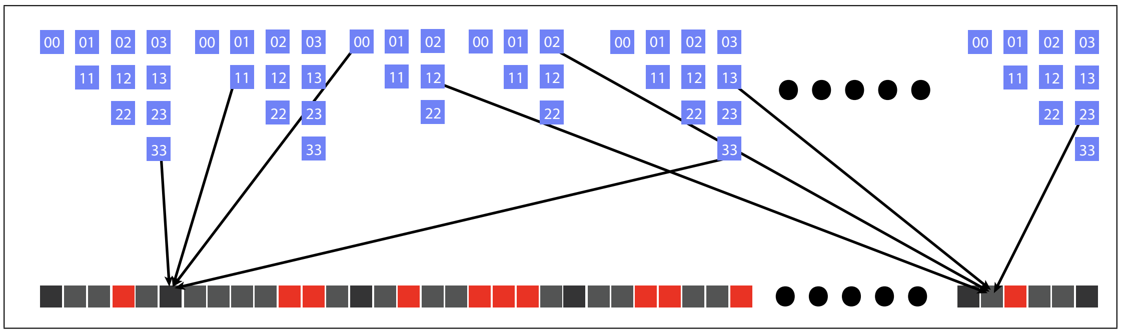
As mentioned, a considerable amount of structure remains fixed during our simulation. The sparsity pattern of our dynamic objects remain fixed, but entries from collision penalty forces are transient and there may be other spring forces that are also transient. We divide our force elements into these two broad categories of fixed and transient.

Fixed force elements have local storage for the forces and their Jacobians with respect to $\mathbf{x}$ and $\mathbf{v}$. To support dihedral bending forces and tetrahedral elements, they have fixed storage for configurations of up to four particles. Since our Jacobians are symmetric, we only allocate the storage required for the upper diagonal block matrices. Given local element particle indices i and j such that $i \leq j$ the index of the 3×3 block in the appropriately sized storage array is given by the following code snippet:

```
uchar FixedElement::localPairToArrayIndex(uchar i, uchar j)
{
    // assert is here for the cautious.
    // If you pass negative indices that's
    // another thing you should never do!
    assert(i <= j);
    uchar n = (_numParticlesInElement << 1) - 1;
    return j + ((n - i ) * i) >> 1;
}
```

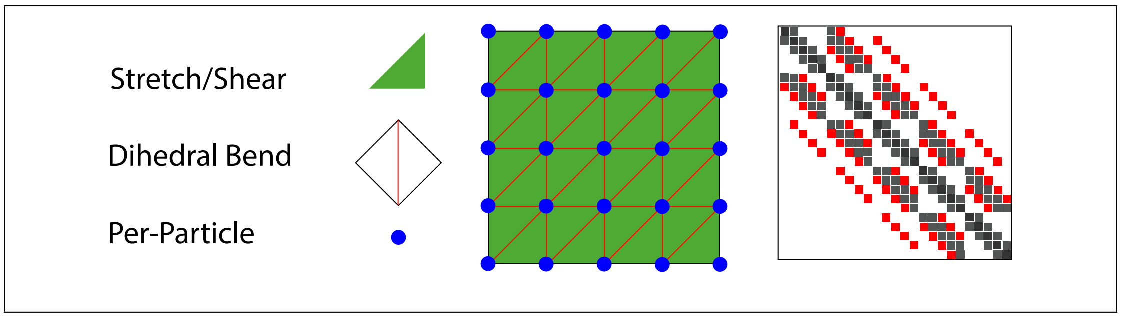
We also create a Fixed Global Sparse Matrix (FGSM) with 3×3 blocks for the entries of $\frac{\partial \mathbf{f}}{\partial \mathbf{v}}$ and $\frac{\partial \mathbf{f}}{\partial \mathbf{x}}$. Each block in the FGSM has a collection of pointers to the blocks of the fixed elements that contribute to it. At the beginning of our simulation, we construct the FGSM and flatten it into a single vector containing all of these blocks. This vector only needs to contain the upper triangular blocks of the system for reasons I'll explain shortly.

The first step in system assembly proceeds by traversing the fixed force elements in parallel to compute the forces and Jacobians which are stored locally. In a second pass, we traverse the blocks of the FGSM to accumulate the results of the first pass. Once the upper triangular blocks are done accumulating all their contributions, the lower diagonal blocks receive a transposed copy of the corresponding upper diagonal accumulated result. This helps reduce the memory bandwidth requirements of the second pass. We use a similar scheme for accumulating the forces from the elements to a global vector with a 3×1 entry per particle. Assembly in parallel with can be done in a single pass using locks and with less memory usage, but the advantage of the two-pass approach is that it is lock-free.



Every simulated particle has a unique *global solver index* that indicates its associated block-row in the linear system. Our global system structures contain block entries both above and below the diagonal. While this requires more memory, it allows for simple and coherent traversal when we need to perform per-row operations. When accumulating into the FGSM blocks, we examine the corresponding pair of global solver indices of the the block to determine whether we can add the block directly, or need to add its transpose. Some Jacobians have symmetric blocks, but others do not. Making a mistake of incorrectly orienting the block during accumulation can lead to long nights of debugging. The resulting simulations may have odd behavior, or your global system may go indefinite. You have been warned.

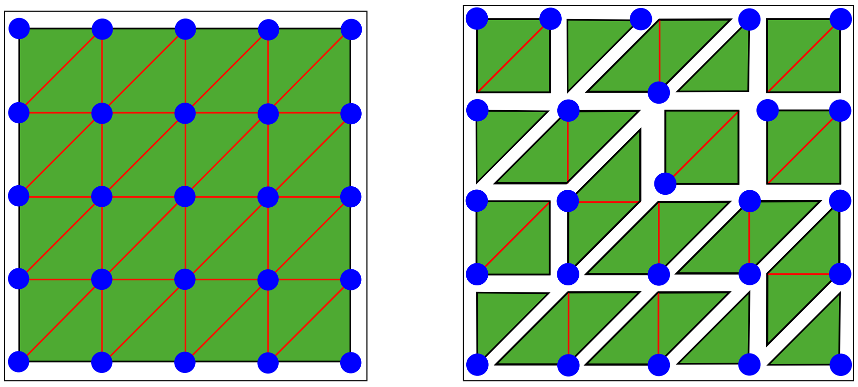
A deformable simulator may have many fixed forces, including simple per-particle forces like gravity, drag, goal-springs, etc. Having a clean and well structured element class for each force type might seem natural at first, but this has a downside. The more elements one needs to traverse, the more memory bandwidth will limit your performance. Consider the sparsity pattern of the following cloth example with point, triangle and dihedral forces:



It makes sense to at least group the element forces by their mesh component domains, but we can go further. Instead of having a force element that handles each type of mesh feature, we aggregate overlapping forces onto the same fixed force element. Some elements naturally will be responsible for more force types than others and some will remain isolated.

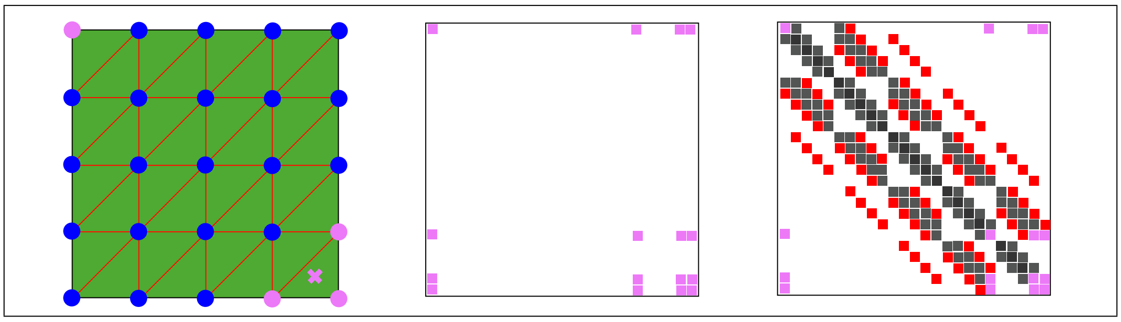
In the case of cloth, we aggregate stretch/shear, bending and per-particle forces. When accumulating the Jacobian blocks over the whole mesh as depicted in image (a), we must traverse 97 elements and accumulate from 1266 element blocks. These estimates assumes each stretch/shear contributes 6 blocks, each dihedral element contributes 10 blocks, and each particle contributes one block. We are only considering blocks from the upper diagonal of the element force Jacobians and our numbers are doubled to account for both $\frac{\partial \mathbf{f}}{\partial \mathbf{v}}$ and $\frac{\partial \mathbf{f}}{\partial \mathbf{x}}$.

By exploiting the aggregation of elements in (b), we only traverse 46 elements and accumulate from 870 element blocks. In this example, the aggregation strategy more than halved the number of elements traversed and reduced our element block access by almost a third. I don't recommend the aggregation approach for a a first implementation, but if you do implement it, remember that the orientation of the blocks must be consistent at the element level when accumulating the Jacobian blocks from different forces.


(a) (b)

Transient force elements are the second source of forces and Jacobians. Unlike fixed force elements, these do not have any local storage. Instead, we define a Transient Global Sparse Matrix (TGSM) which has per-thread row storage of the blocks using a flat-map structure. We traverse our transient elements in parallel, and each one deposits its block contributions into this structure. A parallel per-row reduction step is then performed over the per-thread flat-map structures in the TGSM. We handle force accumulation with a different structure and reduction pass.

We build two matrices to pass to our solver, but many entries from the TGSM overlap with the blocks of the FGSM. In the image below we illustrate the overlap in block entries from a vertex-face contact with the FGSM. We accumulate any overlapping entries from the TGSM into the FGSM in another per-row parallel pass. This process also collects indices to the non-overlapping entries of the TGSM for our next stage of populating our BCRS structures. This process effectively removes the overlapping blocks from the TGSM with respect to our solver.



At this stage, the FGSM and TGSM both contain the accumulated blocks of $\frac{\partial \mathbf{f}}{\partial \mathbf{v}}$ and $\frac{\partial \mathbf{f}}{\partial \mathbf{x}}$. We have not constructed either side of the linear system needed for our PPCG solver. We perform this construction as we build up two separate BCRS structures. As we populate the blocks of our BCRS structures, we first construct the term for $\mathbf{A}$ and then apply any necessary constraint filters $\mathbf{S}$. When we process a block on system diagonal, we add the $\mathbf{I} - \mathbf{S}$ term.

The BCRS layout for the FGSM remains fixed. Only the block values need to be updated, which can be done efficiently in parallel. Further, since the bandwidth of the fixed system has been reduced, we use type `short` to represent our column indices as offsets from the main diagonal for a tiny reduction in memory bandwidth during the sparse-matrix vector multiplies. The BCRS layout for the TGSM is computed at each step from its remaining entries. This task is serial, but requires only one traversal over the rows of the TGSM to determine the storage requirements and rows offsets for the BCRS structure. Once the memory is allocated, these entries are also populated in parallel. We use type `unsigned int` for column indices because an entry may exist in any column from an interaction between two particles. In large systems, a `short` will not have the range for this task. Assuming we have our force accumulation, vector we can also construct the corresponding $\mathbf{b} - \mathbf{Az}$ term for our Right Hand Side (RHS) during the traversal. We then only need to apply our constraint filters $\mathbf{S}$ to this term to complete the system assembly. Instead of performing a second pass over the data, we perform the construction of our RHS during our parallel construction of the LHS.

12.5 System Reduction and Boundary Conditions

At the beginning of each frame we examine which particles will remain fully constrained over it[5]. Sometimes large regions of our simulated objects become fully constrained and we can exploit this. If all the fixed force elements incident to a particle also have all fully constrained particles we remove the particle from the system so that it does not appear in the rows of our linear system[6]. We assign such particles a *global solver index* < 0. Fully constrained elements are also removed from the system assembly. We reindex the remaining particles *global solver index* to the correct range. We don't re-apply our bandwidth reduction for performance reasons. While this won't reduce the bandwidth further we take comfort in knowing it won't grow any larger either.

For each fully constrained and removed particle we calculate $\mathbf{v}_k$ and $\Delta\mathbf{v}_\mathrm{k}$, using finite differences of the known positions. Even though these particles may have been removed from the system, it is possible that collision forces or soft constraints will require the interaction between a simulated particle and one that has been removed. The simulated particle receives the force, but we still need to express the interaction, with what is essentially now a kinematic particle, through the $\frac{\partial \mathbf{f}}{\partial \mathbf{v}}$ and $\frac{\partial \mathbf{f}}{\partial \mathbf{x}}$ terms of our system. The force element Jacobian blocks of the between simulated particles including itself, will get accumulated into the global matrices just as before. Jacobian blocks between a simulated and removed particle must be handled differently since the removed particle is not represented in our system.

Instead we add $[h\frac{\partial \mathbf{f}_s}{\partial \mathbf{v}_k} + h^2\frac{\partial \mathbf{f}_s}{\partial \mathbf{x}_k}]\Delta\mathbf{v}_\mathrm{k}$ from the mixed block contributions of the LHS to

[5]We also examine if any removed particles become unconstrained and if they and any associated elements need to be added back. We only remove particles if the number to be removed exceeds a threshold of 200.

[6]We leave a boundary of fully constrained particles that have both simulated and non-simulated particles in our system presently even though technically this could be avoided.

the RHS. We also add any mix block velocity contribution from the $[h^2 \frac{\partial \mathbf{f}_s}{\partial \mathbf{x}_k}]\mathbf{v}_k$ term in the RHS. This technique is also used to handle arbitrary spring forces between points between a dynamic and kinematic object without increasing the size of the system.

12.6 PPCG Solver Details

Fizt computes almost everything in double precision. The primary exception is our PPCG solver, and the system we pass to it. When we assign the block values to our BCRS structures, we convert to single precision. We perform the same conversion to the RHS after applying our constraint filters. We pass a single-precision version to our PPCG solver to further reduce the memory bandwidth requirements of the sparse matrix vector multiplication.

Our PPCG solver is non-standard in that it takes two separate BCRS structures to represent the Left Hand Side (LHS), but the benefits of exploiting the fixed structure during assembly appears to be worth the awkwardness[7]. We have to handle the application of our LHS in this split form. We first determine which rows have terms from the TGSM BCRS and FGSM BCRS. Each row gets a function pointer appropriate to the task of multiplying the necessary entries with a vector. We also utilize loop unrolling of the row traversal with a block size of 4. The ranges and remainders for this unrolling are computed once per solver call solver for the entries of the TGSM BCRS. There are probably more gains to be had by spending more long nights with a profiling tools, a snapshot of our performance gains between early 2015 and Summer 2016 are presented in figure 12.5. The original times per frame do not include the addition of continuous collisions, which will be discussed later. The 2016 timings include the additional overhead from continuous collisions.

12.7 Preconditioning

Fizt uses a 3×3 Block Jacobi preconditioner that is the inverses of the diagonal blocks from the LHS. While simple, we have found that it works well in practice. More sophisticated preconditioners have been attempted, such as the parallel Incomplete Cholesky of Chow and Patel (2015), and the other preconditioners in the ViennaCL library (Rupp et al. (2016)). We found that they were not able to consistently accelerate our tests. Multigrid preconditioners like the one of Tamstorf et al. (2015) are also a possibility, but their advantages only begin to become significant when meshes have very large vertex counts.

One obvious question is whether 3×3 is the optimal size for a Block-Jacobi preconditioner. We did indeed run a series of tests to verify this. Table 12.6 shows timings we collected a sandbox quasistatic simulator (*Glove*) in order to study whether a different block size might benefit wider deployment in *Fizt*. The mesh consisted of 70,831 vertices and 141,579 triangles.

[7]We experimented with using ViennaCL on the CPU with the same convergence tolerance and our custom solver was still faster.

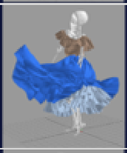
Example	Simulated Vertices/Faces	Assembly	Solve	Average Frame
	20834 / 40643	2.07 vs. 0.60	3.57 vs. 0.65	8.50 vs. 3.26
	27384 / 49826	2.45 vs. 0.88	1.77 vs. 0.46	9.47 vs. 4.97
	49647 / 95744	4.84 vs. 1.74	6.86 vs. 1.51	15.9 vs. 6.76
	46359 / 90783	2.98 vs. 1.10	4.90 vs. 1.23	12.0 vs. 5.32

Figure 12.5.: Fizt examples circa 2015/2016. All were run with 6 threads at 10 steps per frame. All times in seconds.

For Table 12.6, we ran a quasistatic skin simulation, where the skin was simulated using a cloth simulation as described in Kautzman et al. (2018). For measurement purposes, the simulation was run to four digits of precision for three Newton iterations. We report the number of PCG iterations needed, as well as the overall wall-clock time. The 3×3 block is the clear winner on wall-clock time. The larger block sizes predictably reduce the number of PCG iterations, but the growing cost of computing the block inverses consistently outweigh this reduction.

The next question is whether a faster but less precise block inverse would win back some of the speedup. Again, as a preliminary test we tried a variety of factorizations available in the Eigen (Guennebaud et al. (2010)) library, and the data is shown in Table 12.7. None of the factorization strategies produced dramatically improved results.

To follow this line of inquiry to its conclusion, the last thing to try is an Incomplete Cholesky or LU factorization that approximates the block inverses. However, we halted the study based on the existing data. Even if an incomplete factorization could be found that exactly (and impossibly) matched the full factorization, and ran in the same wall-clock time as the 3×3 case (again impossible), it would only reduce the number of PCG iterations by 20%. Since the PCG solver generally consumes 33% of the wall-clock time, the expected speedup to *Fizt* would only be around 6.7%. Investing more effort in chasing this level of acceleration did not seem promising, so the inquiry was halted.

Block Size	Iteration 1	Iteration 2	Iteration 3	**Total Time**
1	981	822	2028	17.03s
3	625	531	1023	12.47s
6	654	542	1203	13.30s
10	721	597	1217	13.98s
12	628	541	1123	13.71s
25	666	553	1321	14.71s
50	598	513	850	13.69s
100	486	507	826	15.00s

Figure 12.6.: Performance of Block-Jacobi Preconditioning over different block sizes. We ran a quasistatic cloth solve for three Newton iterations, and report the number of PCG iterations for each iteration, as well as the wallclock time of the overall solve.

Factorization	Iteration 1	Iteration 2	Iteration 3	**Total Time**
$\mathbf{LDL}^T$	486	507	826	15.44s
Column-pivoting **QR**	498	509	837	17.42s
Full-pivoting **QR**	498	509	837	22.00s
Householder **QR**	498	509	837	17.63s
Partial-pivoting **LU**	498	509	837	17.24s
Full-pivoting **LU**	498	509	837	17.45s

Figure 12.7.: Performance of Block-Jacobi Preconditioning for 100×100 blocks, using different factorization strategies. The setup is otherwise the same as Table 12.6.

12.8 A Word on Determinism

Fizt is made parallel using *OpenMP*. This includes our PPCG solver. Dynamic scheduling is used, but care must be taken when doing so. Using dynamic scheduling in a loop containing reduction can lead to non-deterministic results because of differences in the order of rounding error accumulation. Having a suite of regression tests that exercise the simulator is a critical part of maintaining production code, but this can only be achieved though if we have deterministic behavior. Otherwise, false positives will have us chasing non-existent bugs.

Other common sources of non-determinacy fall into a handful of categories. Do not store anything as a pointer in an ordered container like a set or map if traversal of the container can impact rounding error accumulation. Contacts are one such category. We find and gather penalty force contacts in a parallel in non-deterministic fashion, but choose not to compute the response on the fly. Instead we sort them before computing and accumulating their force and Jacobian terms which ensures that their contributions are accumulated the same way each time for a given thread count. There is a performance hit for a determinism guarantee. *Fizt* has a deterministic mode for testing and a faster non-deterministic mode for production use.

12.9 Writing Efficient *OpenMP* Code

Writing a single threaded version for a first implementation is advised to keep things simple for an initial pass of debugging. Once you are confident in your implementation though, you are going to want speed. Using *OpenMP* naively is pretty straightforward. One can parallelize loops with simple for-loop or task-loop `pragmas` and can get a nice gains over a single threaded implementation quickly that will scale to 4-6 threads. The ease of using these `pragmas` can lead to writing inefficient code however and will limit the scalability to higher thread counts. In early 2021, a major refactor of *Fizt* was performed to have the entire simulation occur in a single parallel region to reduce thread creation overhead. In this refactor many barriers were removed. Task parallelism was also added to the BVH intersection, but steps needed to be taken to limit the number of tasks spawned. We also found that `omp for` had less overhead than `omp taskloop` for parallelism over arrays. The performance impact on *Fizt* before and after this refactor can be seen in figures 12.8 and figure 12.5. In this example, Buzz's suit is comprised of 67K simulated vertices with two layers in constant contact and over 200K vertices in the detailed collision geometry. Despite the pinching and geometric complexity *Fizt* can simulate this in a little over 12 seconds per-frame with taking 10 steps per frame using 12 threads[8].

[8]On can see that the thread scaling plateaus after 12 threads. The 2021 refactor was just one of a series on ongoing efforts to improve performance.

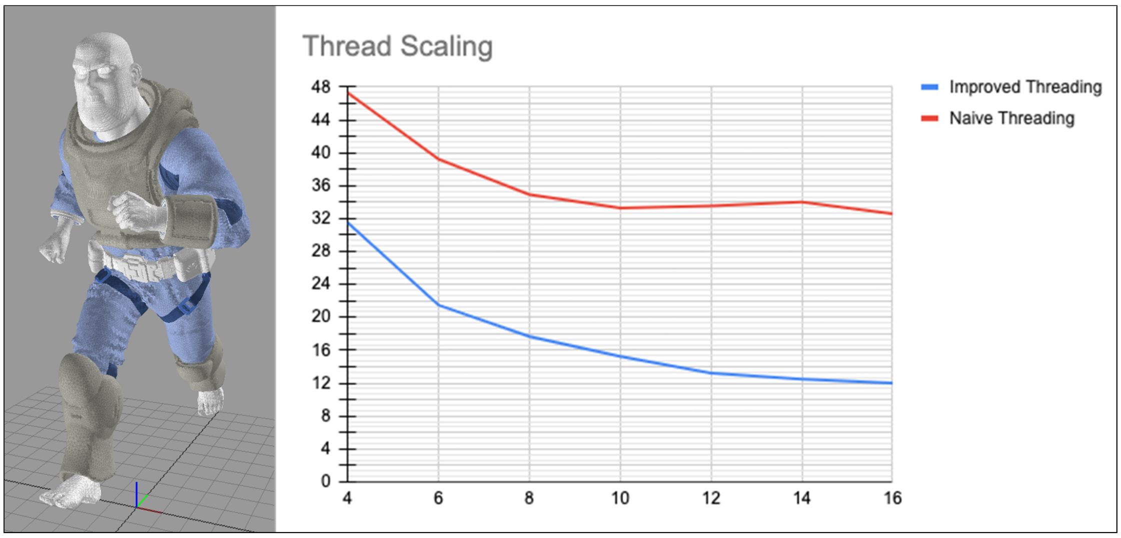


Figure 12.8.: Timings for Buzz example over different thread counts before and after the threading refactor. Simulation times per-frame on the vertical axis are in seconds.

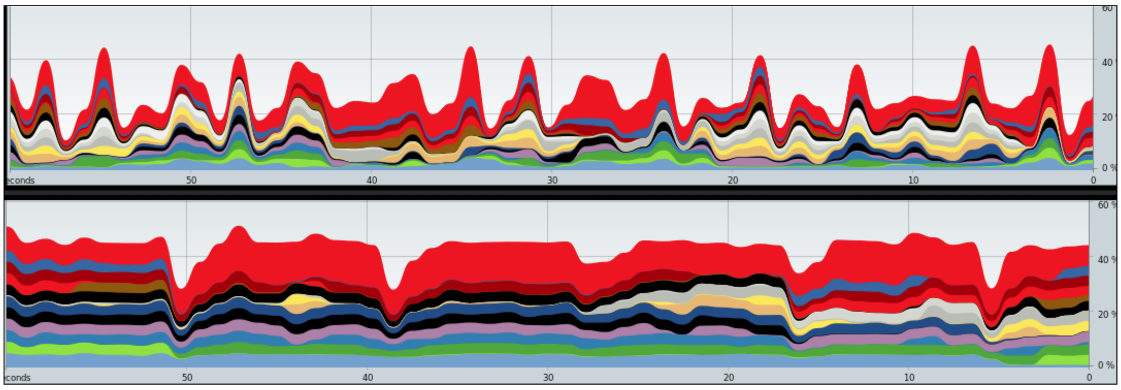


Figure 12.9.: CPU utilization graph on the Buzz example running with 16 threads, half the number of available threads, before and after the threading refactor.

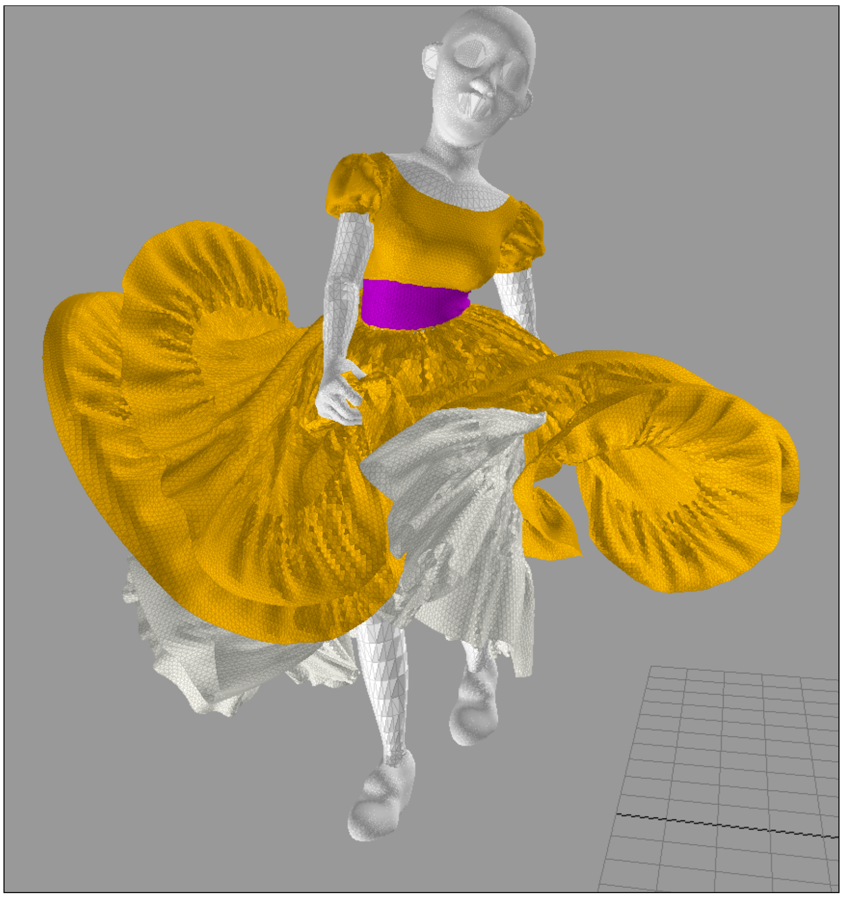


Figure 12.10.: After threading improvements, the dancer asset from *Coco* with over 105K simulated vertices simulates in 6 seconds per-frame with 16 threads.

Chapter 13

Collision Processing

Collision detection and response are critical components of any dynamics simulator. A vast body of work on collision detection exists and a full day course on the topic could not do justice to it. In this chapter we will only discuss the techniques that we employ in our simulator *Fizt*, and acknowledge that there are a number of other ways to approach these problems. That said our combined approach has proven itself able to handle challenging scenarios found in a production environment and often ignored by the literature.

Fizt is used to simulate cloth and volumes that must be able to collide with the animated geometry which, we refer to as *kinematic geometry*. Collisions between *dynamic geometry*, including self-collision, must also be handled by our system. All of our collision geometry is described by manifold triangles meshes and in the case of tetrahedral volumes we only collide with the surface mesh. Each triangle mesh has an axis aligned bounding box hierarchy which is updated at each time step, and is typically rebuilt once per frame. The number of separate objects in *Fizt* is typically small, so we do not bother with a broad-phase collision algorithm like those typically found in rigid body simulators.

13.1 Proximity Queries

Proximity queries look at the geometry at a discrete snapshot in time and are used to detect potential collisions between mesh features. The goal of the response is to prevent actual collisions from occurring over a the time step. Our response to proximity-based contacts is encoded in our linear system, and interacts with the all the other forces in our system.

Vertex-face queries between dynamic particles and the faces of our kinematic meshes are performed first. Our kinematic meshes are mostly closed and consistently oriented. We define a collision envelope around the surface by offset and inset distance parameters. To prevent against redundant contact data, we partition the space above and below each mesh face using this envelope. We refer to these partitioned regions as *cells*. The *cells* provide us with a way to cull many of the undesirable contacts that occur when the mesh resolution is much smaller than the specified collision envelope [1].

To construct the *cell* regions of the faces, we first define a unique bisector plane for each edge[2]. We define the edge normal, $\mathbf{n}_e$ as the average of the available neighboring face normals. The normal of the bisector plane for this edge is then defined by $\mathbf{n}_{eb} = \mathbf{n}_e \times \mathbf{e}_{ij}$ with $\mathbf{e}_{ij} = \mathbf{x}_j - \mathbf{x}_i$. We then normalize to get $\hat{\mathbf{n}}_{eb}$ and define distance from a point $\mathbf{x}_p$ to the bisector plane by $d_{\text{eplane}} = \hat{\mathbf{n}}_{eb} \cdot (\mathbf{x}_p - \mathbf{x}_i)$. We then orient our plane inward toward its first face neighbor, f_0, by ensuring that the vertex opposite the edge produces a positive distance to the edge bisector plane.

To examine whether a particle is in a face cell, we first check that the position is between all three bisector planes by computing d_{eplane} for all three edges. When performing this check, we examine whether the face is f_0 or f_1 with respect to the edge. If it is f_1, we negate the distance returned. This allows for a simple check that $d_{\text{eplane}} >= 0$ for all three edge bisector planes regardless of which face we are querying. Then we compute $d_{\text{fplane}} = \hat{\mathbf{n}}_f \cdot (\mathbf{x}_p - \mathbf{x}_0)$ to determine if it is within our collision envelope in the normal directions and if it is above or below the face plane.

At first glance, this looks like a closest-point lookup into a clipped Voronoi diagram. While this does characterize the local picture, the global view is subtly different. Each face has its own clipped Voronoi cell, but the Voronoi cells do *not* partition space, and are allowed to overlap. An example of this is shown in Fig. 13.1. Since cloth can double back on itself in space, it is in fact possible for a particle to be inside more than one cell at the same time. The particle is not allowed to be inside two *adjacent* cells at the same time, but if it is inside the collision cells of two faces that are far away from each other, we consider the particle to be in collision with both faces.

A particle position that satisfies all of these checks is inside our cell region, but this does not necessarily mean it violates our collision envelope. As an edge becomes more creased, it becomes more probable that the point can be inside the cell, but outside our collision envelope. We illustrate such this in the second image of figure 13.2. After we verify the point is inside the cell, we compute closest point to the face, (Schneider and Eberly (2003)), and compare the distance to our collision envelope.

To create the bounding boxes of our cell regions, we compute the vectors at each vertex defined by the intersection of the incident bisector planes. Using these directions, we enclose the the extruded vertices in both the offset and inset directions. Instead of scaling the direction by the inset or offset directly, we use the magnitude required to reach the inset or offset in the normal direction with respect to the face[3].

[1]Some post processing of the per-particle contact data is still required despite our region partitioning. It is still possible that particle could contact multiple incident faces at a vertex or edge boundary.

[2]One should always guard against degenerate cases like collapsed faces, edges or vanishing edge normals.

[3]Finding the minimum bounding-box of a face cell requires some attention when the dihedral angle between edges becomes large.

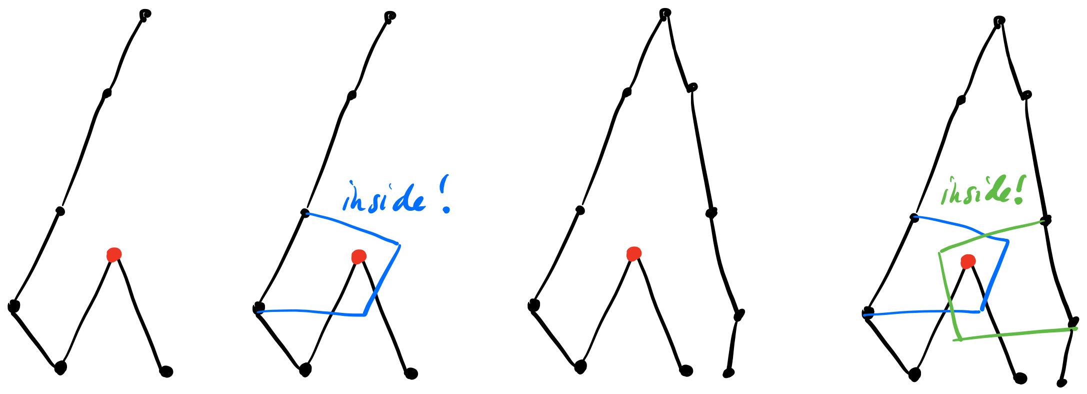


Figure 13.1.: A 2D version of our collision cell strategy. On the left, we are looking at the red vertex, and seeing which line segments in the mesh it is in collision with. On the middle left, it is inside the blue cell, so the vertex is in collision with that segment. On the middle right, what if the mesh doubles back on itself? On the right, it is *totally legal* for the vertex to be inside cells of two segments, both the blue and green ones. The vertex is in collision with *both* segments.

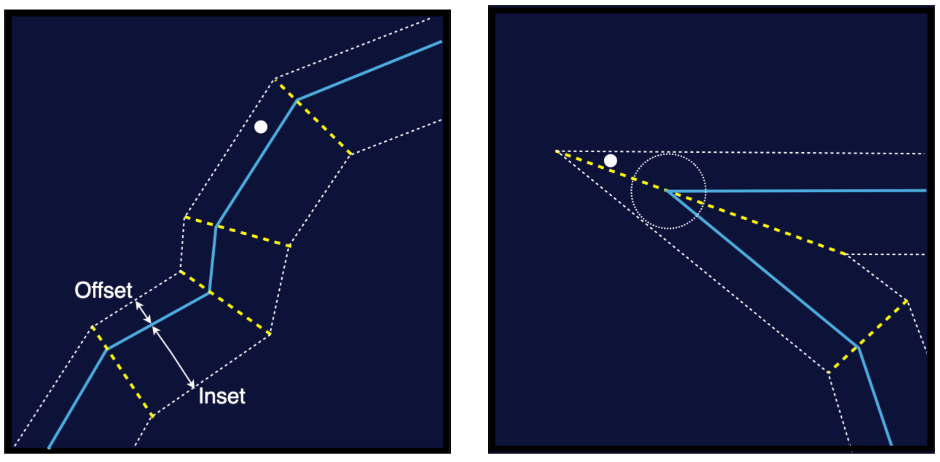


Figure 13.2.: Particle-Kinematic Face Cell Check: (a) Cell region containing a particle in the envelope. (b) Particle is inside the cell, but outside envelope.

Large inset values for relatively thin regions can cause the inset region to extend through the backside of an opposing region of the the surface. This can lead to particles getting an erroneous response that pulls them through the other side of a surface. An inset reduction step is performed each time the simulator receives a new sample of the *kinematic geometry*. The reduced insets at these samples are then interpolated across the time steps. When we discover a particle that satisfies all of our contact criteria, we create a contact constraint for the particle and add the kinematic face to it. If a particle already has a contact constraint, we add the new face to it.

13.2 Filtered Constraints for One-Way Response

Our kinematic meshes have scripted motion that remains unchanged by the simulator. The scripted motion allows us to know the kinematic positions at the end of each time step. One can think of kinematic objects being as made of particles with infinite mass, which means that collision with a dynamic particle is a one-way interaction.

A proximity contact with a single kinematic face has a straightforward response. We define local coordinates for the particle at the current state using the u, v coordinates and its normal offset from the face. The u and v are the first two barycentric coordinates of the particle's projection into the plane, but may lie outside the boundary of the face. Since we know the state of the kinematic triangle at the end of the step, we can construct the particles's desired world space position at the end of the step. Using the current particle state $\mathbf{x}$ and $\mathbf{v}$, we can determine the $\Delta\mathbf{v}$ needed to reach this desired position. We use the constraint filtering mechanism described in §12.1 to ensure our system solution achieves the target $\Delta\mathbf{v}$ in the normal direction at the end of the step. Constraining only the normal direction allows it to slide tangentially. If we included any collision offset correction in our desired target position, it would result in unnecessary energy being added to the system. The technique described so far is only used to match the velocity of the point on the kinematic face. Offset correction is handled by the position alteration technique described section 6.4 of Baraff and Witkin (1998). They compute the $\Delta\mathbf{x}$ term in the normal direction at the end of the step that is required to reach the desired collision offset. Then $\Delta\mathbf{x}$ is added to the appropriate term multiplied by $\frac{\partial \mathbf{f}}{\partial \mathbf{x}}$ in the RHS of the system. Finally, $\Delta\mathbf{x}$ is added to the position state update. The addition of the term to the RHS allows the connected particles to be informed about the additional displacement during the solve.

Following section 5.4 of Baraff and Witkin (1998), we determine whether a constrained particle that is sliding on a kinematic surface should be released by computing the residual constraint forces as the last step of our linear system solve. If the constraint force was pulling toward the surface, we do not constrain the particle in the next time step. There are more accurate ways to handle this, but as noted in Baraff and Witkin (1998), it is far less expensive than dealing with the combinatorial complexity of handling proper inequality constraints.

When a particle is in proximity of multiple kinematic faces, we employ a variant of the

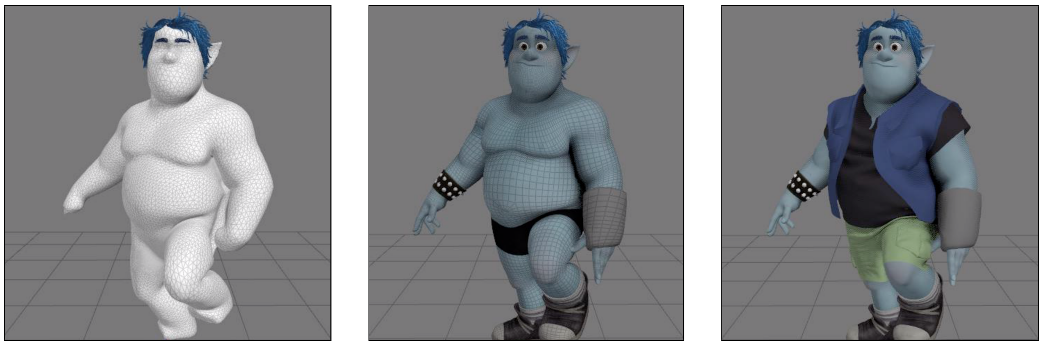


Figure 13.3.: Character Self-Intersection Resolution: (a) Volume simulation with self-collision. (b) Deformed subdivision surface. (c) Clean cloth simulation result.

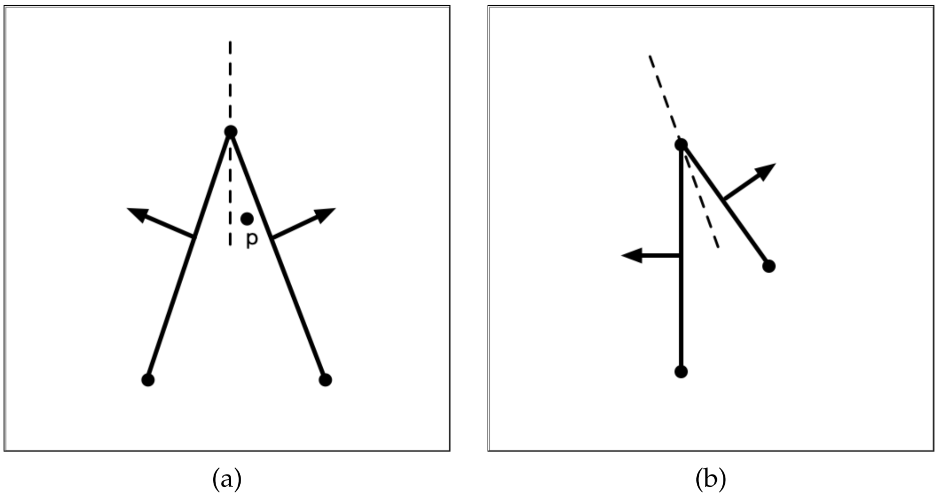
Flypapering technique described in Baraff and Witkin (2003). Each dynamic object in *Fizt* has an identical mesh representation that has its vertex positions warped by our character rig. Instead of using a weighted average of the target positions from multiple faces, we fully constrain the particle to move relative to one of the faces found by our proximity query. We choose the face with the motion that minimizes the difference in displacement between the constrained motion it would have on the particle and the motion of the warped position during the step. Said another way, we choose the face that would most closed match the velocity produced by the rigged motion.

In recent films we have taken a very different approach to handling self-intersection of our characters. We now run a volume simulation to deform the character geometry in order to leave space for a cloth simulation in a second pass. This is depicted in figure 13.3 and further details can be found in Wong et al. (2018). This technique is relies on a robust and efficient constitutive model, but also heavily depends on our collision detection and response between dynamic surfaces which we will cover next.

13.3 Proximity Between Dynamic Meshes

Our second set of proximity queries is for collision, and self collision, between dynamic surfaces. Dynamic surfaces such as cloth are not closed, and do not have the concept of an inset region. Instead, we have a self-offset separate from our collision offset which is used for collisions between different dynamic meshes. Our vertex-face proximity query is very similar to that of kinematic meshes, but it must be altered to handle open boundaries. When a face has a boundary edge, we extend its bounding box by the maximum of the self-offset and offset values in the appropriate directions. To do this, we examine the sign from the negative of the boundary edges's bisector normal. The box is extended with the maximum of our two offsets in the appropriate cardinal directions.

Our particle-in-cell test is no longer sufficient, but is still necessary. Without some notion of space-partitioning, a large self-offset on a fine resolution mesh could lead to undesirable self collisions, which would deform the our mesh in ugly ways. Let's examine a couple of particle-face cases where our previous check breaks down. In case (a) below the separating plane between the two faces would only allow the particle to collide with one face when it could be inside the collision envelope of both. In case (b) the self-collision between a face and a neighboring particle would be rejected by the bisector plane between the faces.



To address these issues, and to handle boundary edges, we present the new test criteria summarized by the following pseudo-code:

```
bool InFaceRegion(p, triFace)

    // If the projection of the point into the face plane
    // resides inside the face boundary
    if pproj in triFace:
        return true

    distToPlane[3] = {triFace.DistToBisector(0,p),
                      triFace.DistToBisector(1,p),
                      triFace.DistToBisector(2,p)}

    // Test if the point is in the cell of the face
    if all distToPlane >= 0:
        return true
    else if not hasBoundaryEdge:
        return false

    // Is the point outside  the plane of a boundary edge,
    // and inside the planes of all non-boundary edges?
    if (not triFace.IsBoundaryEdge(0) and distToPlane[0] < 0) or
       (not triFace.IsBoundaryEdge(1) and distToPlane[1] < 0) or
       (not triFace.IsBoundaryEdge(2) and distToPlane[2] < 0):
    {
        return false
    }

    // The point is in the region of the face
    return true
```

After a point passes this check, we determine if it is within our collision envelope. When a dynamic contact is found, the four particles and the barycentric coordinates between the closest points are stored.

Edge-to-edge contacts also employ our `InFaceRegion` test, but in a slightly different fashion. We first find the closest point between the two edges, e_a and e_b, using the techniques described in Möller et al. (2008), which in turn references Rhodes (2001). [4] If the closest points are on the edge segments, and not at any of the vertices, we examine if the distance between the two closest points are within our collision envelope. After establishing the distance this criteria, we check that the point of contact on e_a passes our `InFaceRegion` test with the available adjacent faces of e_b. Similarly, we examine whether the point of contact on e_b passes our `InFaceRegion` test with the available adjacent faces of e_a. Only if the test pass on both sides do we create a dynamic contact for later processing.

Certain feature pairs are purposefully omitted. A vertex should never be tested against

[4]In this test, we ignore the case when the closest point between edges is an end point, because that will be covered by our vertex-face test. A separate test for when the edges are deemed parallel is also performed.

any of its incident faces[5] and two edges that share a vertex should never be tested.

13.4 Penalty Forces for Two-Way Response

For a long time in computer graphics, penalty forces have been frowned upon, and probably still are by many. To be effective, these forces must typically have a large magnitude, which leads to stiff systems will make explicit integrators slow or unstable. We are using implicit integration, so this is not a concern for us. Even so, we still don't know the magnitude of our penalty force required to make any guarantee that it will prevent a potential collision from occurring. If we wanted to enforce equality constraints, our linear system would have to be augmented with Lagrange multipliers, and still would have the problem of our constraints exhibiting sticking behavior. Handling this properly involves solving large scale optimization problems with inequality constraints. Solving this problem between deformable bodies has come a long way in recent years, but the computation times remain unsuitable for our production use. Penalty forces have served us well in practice, so we will cover some of their implementation details.

Before proceeding further, there is some convention and notation to discuss. Our calculated response force will act in the direction of the collision normal, which we define to point from entity b to entity a. Particles on the a and b side of the contact are assigned signed barycentric weights w with the convention that $w_a \geq 0$ and $w_b \leq 0$. Our penalty contact object stores the signed weights in a single array w that shares the same ordering as the particles.[6]

This makes computing relative quantities between the points of contact a weighted sum. In the case of vertex-face proximity, we assign a positive weight of one to the vertex particle and the negative of the barycentric coordinates to face particles. We will refer to the points of contact on each side by $\mathbf{p}_a$ and $\mathbf{p}_b$. For now, let us assume that we are not recovering from a tangled situation as described by Baraff and Witkin (2003) and that our penalty spring is in a state of compression. We define $\Delta\mathbf{x} = \sum_{i=0}^{3} w_i \mathbf{x}_i$ and $\Delta\mathbf{v} = \sum_{i=0}^{3} w_i \mathbf{v}_i$. Our desired collision distance is d_{offset}. The spring stiffness and damping parameters, which must be positive, are k_s and k_d. The proximity contact normal is defined as $\hat{\mathbf{n}}_c = \frac{\Delta\mathbf{x}}{\|\Delta\mathbf{x}\|}$. We can now express the force acting on $\mathbf{p}_a$ as:

$$\mathbf{f}_a = -[(k_s(\|\Delta\mathbf{x}\| - d_{\text{offset}}) + k_d(\Delta\mathbf{v} \cdot \hat{\mathbf{n}}_c)]\hat{\mathbf{n}}_c. \tag{13.1}$$

We deliberately chose to compute the response at $\mathbf{p}_a$ for our vertex-face example. Distributing the response force to the particles is a scaling of $\mathbf{f}_a$ by the signed weight of each particle. This follows from a simple application of the chain-rule and our sign convention, which conveys that $\mathbf{f}_b = -\mathbf{f}_a$.

[5]The actual code comment reads: "Never collide with your own face. This is always sound advice!"

[6]We will see in Chapter 14 that this corresponds to multiplying by a $\frac{\partial \mathbf{t}}{\partial \mathbf{x}}$ term (Eqn. 14.11) that appears when taking the energy derivative.

$$\mathbf{f}_i = w_i * \mathbf{f}_a \tag{13.2}$$

We must now compute the force gradients of our spring for implicit integration. The spatial gradient block of $\mathbf{f}_a$ with $\mathbf{x}_{pa}$ is:

$$\frac{\partial \mathbf{f}_a}{\partial \mathbf{x}_{pa}} = -k_s[\hat{\mathbf{n}}_c\hat{\mathbf{n}}_c^T] - k_s(\|\Delta\mathbf{x}\| - d_{\text{offset}})[\mathbf{I} - \hat{\mathbf{n}}_c\hat{\mathbf{n}}_c^T] \tag{13.3}$$

The full spatial force gradient for a linear spring is given in equation (13.3), but it presents us with a problem. For a spring in compression, the second term can make our system indefinite, as described in Choi and Ko (2002). This term filters motion orthogonal to the direction of our spring response, so we discard it and only keep the outer product term:

$$\frac{\partial \mathbf{f}_a}{\partial x_{pa}} = -k_s[\hat{\mathbf{n}}_c\hat{\mathbf{n}}_c^T] \tag{13.4}$$

Our signed weight convention make it easy to distribute these Jacobians to the blocks that correspond to our particle pairs. If i and j are the indices of any two particles on in our contact configuration then:

$$\frac{\partial \mathbf{f}_i}{\partial \mathbf{x}_j} = (w_i w_j)\frac{\partial \mathbf{f}_a}{\partial \mathbf{x}_{pa}}. \tag{13.5}$$

The same relationship naturally holds for the analogous velocity gradient block:

$$\frac{\partial \mathbf{f}_a}{\partial \mathbf{v}_{pa}} = -k_d[\hat{\mathbf{n}}_c\hat{\mathbf{n}}_c^T]. \tag{13.6}$$

The conventions introduced in this section make it trivial to generalize code to handle the response for edge-edge proximity contacts, but it also generalizes further. We can define arbitrary forces to act on any point of a mesh, distribute the forces to the mesh particles, and distribute the Jacobians to the blocks of the linear system. Think of the cool things you might construct with this power!

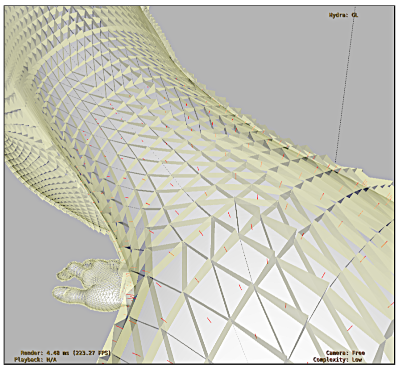
13.5 Faux Friction Effects

Accurate frictional forces are not an option for the performance requirements of *Fizt*. Instead, for dynamic contacts, we allow artists to control tangential motion though a tangential damping parameter k_{td}. Our tangential damping force is defined $\mathbf{f}_t = -k_{td}\mathbf{v}_{trel}$ which acts opposite the direction of the relative tangential velocity. As before, we use the signed weights w to distribute this force to the particles involved. We add the contribution $-k_{td}[\mathbf{I} - \hat{\mathbf{n}}_c\hat{\mathbf{n}}_c^T]$ to our velocity gradient block $\frac{\partial \mathbf{f}_a}{\partial \mathbf{v}_{pa}}$ before distributing the weighted blocks of the particle pairs. This approach is cheap and artist friendly, but it can never model static friction behavior because there must be some relative tangential motion present in order for a resisting force to be applied.

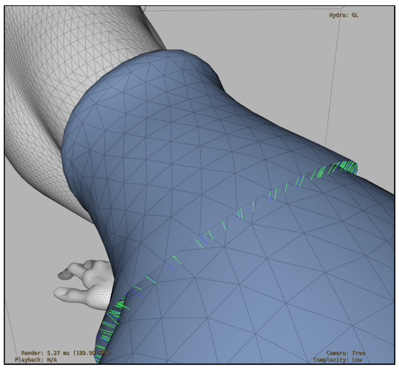
Friction effects between dynamic particles and kinematic faces are handed differently. After accumulating all the forces on each particle, if the particle has a contact with a single kinematic face, we use the accumulated force to estimate the normal force. When the normal force is negative, a tangential spring force is then applied. The magnitude of this tangential spring force is set to $\mu\|\mathbf{n}_f\|$ to approximate dynamic friction behavior[7].

13.6 Debugging Proximity Contact Detection

Writing small unit tests is highly recommended, but one might omit or misunderstand certain cases. In addition to unit tests, I recommend developing the ability to visualize algorithms in action. Pixar's Universal Scene Description (USD), has proven quite useful for this task. *Fizt* has the option to write out the geometry and partition cells at sub-frames. We also write out proximity contacts as linear curves that vary in number over time. Using Pixar's `usdview` application, this data can be inspected over time. Image (a) below shows the partition cells on a kinematic mesh along with lines that correspond to dynamic particle contacts with it. Image (b) shows the vertex-face and edge-edge self contacts. Similar visualization output also exists for our continuous collision and global intersection analysis algorithms, which will be discussed in later sections.



(a)



(b)

13.7 Continuous Collision Detection

Thin and fast-moving geometry will lead to temporal aliasing issues if a simulator relies solely on proximity queries. Instead of testing discrete states for probable contacts, Continuous Collision Detection (CCD), determines the true collisions that occur between two position states of our geometry. CCD techniques for deformable surfaces were first made popular by Moore and Wilhelms (1988), Provot (1995) and Bridson et al. (2002). A large volume of work followed over the years to make the technique more robust

[7]This uses an explicit guess for the normal force, and does not reflect the true normal force at the point of contact over the time step.

to numerical issues and to improve the performance. However, many papers on CCD emphasize the robustness of their method and make strict guarantees that *if* things start in a penetration-free state, the algorithm will reliably maintain it. It is our experience that animation often requires things to intersect at one time or another, and that such guarantees are impossible in practice. Also in a production environment, it is an extra burden for the artists to guarantee a pristine initial state. In the last section of this chapter, we will discuss how coupling CCD with the results of another algorithm can help us handle these issues.

In *Fizt*, CCD is performed after we have integrated to find a new candidate position state. CCD assumes a linear trajectory of the particles over the time step, which is conveniently true for our Backward-Euler integrator as well. Our BVH structures enclose the mesh primitives at both of these states. Candidate vertex-face and edge-edge pairs are then discovered by leaf bounding-box overlaps.

The primitive test for both cases share a common idea. Using the subscripts s and e to denote the start and end positions of a vertex, we parameterize the motion of each vertex with respect to the variable $t \in [0,1]$ by $\mathbf{x}_i(t) = (1-t)\mathbf{x}_{is} + t(\mathbf{x}_{ie})$. We construct a tetrahedron from the four vertices at the begin state, and want to determine if the volume ever goes to zero during the interval. This corresponds to all four points becoming co-planar.

Let us first examine the vertex-face case. We define the face normal over time by:

$$\mathbf{n}(t) = (\mathbf{x}_1(t) - \mathbf{x}_0(t)) \times (\mathbf{x}_2(t) - \mathbf{x}_0(t)) \tag{13.7}$$

When $\mathbf{n}(t) \cdot (\mathbf{x}_p(t) - \mathbf{x}_0(t)) = 0$ our vertex v_p is coplanar to the face.

A similar test can be made between two edges e_a and e_b. Each edge has a two vertex trajectories, $\mathbf{x}_0(t)$ and $\mathbf{x}_1(t)$. We define the normal by:

$$\mathbf{n}(t) = (\mathbf{x}_{a1}(t) - \mathbf{x}_{a0}(t)) \times (\mathbf{x}_{b1}(t) - \mathbf{x}_{b0}(t)) \tag{13.8}$$

The equation for our test becomes $\mathbf{n}(t) \cdot (\mathbf{x}_{a0}(t) - \mathbf{x}_{b1}(t)) = 0$.

Both tests result in an equation that is, at most cubic in t. Only the real roots $t \in [0,1]$ are of interest. Even though there is an analytic formula for the roots of a cubic polynomial, robustly finding these roots is not easy. How small can the leading coefficient get before the equation is considered a quadratic? When inserting roots back into the cubic equation, what is an acceptable error? Exact boolean tests for collision have been developed by Brochu et al. (2012) and Tang et al. (2014), and while they are recommended to avoid missing a collision, they do not find the root needed when a collision does occur. We recommend the analytic techniques described in Jim Blinn's multi-part article, *How to solve a cubic equation*, but found it necessary to polish our roots with a numerical technique. The Boost math library provides an implementation of Schröder's method, which we've found adequate for this purpose. We fully acknowledge that this approach is not as robust as other methods available in the literature.

Geometric tests are performed for each polished root $t \in [0, 1]$ in sorted order. The position of the vertices are first advanced to the time of our candidate root t. Numerical roundoff will likely keep the points from being "exactly" coplanar, so this should never be assumed by the geometric test. If we determine that the point of contact lies within our features, then we can stop our search and record the contact. Otherwise we must check any remaining root candidates.

A good unit testing framework is always your friend, but this can be especially true when developing algorithms can that be affected adversely by numerical errors. One can only begin to have some confidence after procedurally testing millions of collision scenarios that cover various scales and trajectories. Even having taken such measures for our implementation, failures are still possible since we don't perform the exact methods mentioned. In the rare case of failure, another method which we will describe later will allow us to recover in most cases.

13.8 CCD Response of a Single Collision

After CCD provides us with the collisions that occurred between candidate states, we need to resolve these collisions. Particles that were fully constrained are considered to have infinite mass. The vertices of *kinematic geometry* features can then be thought of as fully constrained since we cannot alter their motion with our response. Instead of operating directly on the velocities of the vertices, we alter their end positions during each iteration of our response algorithm. Once the entire response algorithm has completed, any particles with altered positions will have their velocity states changed to reflect this to be consistent with these altered positions.

To resolve a single continuous collision, we will follow the notation used to described inelastic projection described from Harmon et al. (2008). Although their work focuses on a multiple simultaneous contacts, the math provides a compact form for resolving a single contacts as well[8]. Our discussion of Harmon et al. (2008) will also vary slightly since we choose to adjust the end positions instead of the candidate velocities. This allows our CCD detection and response module to have knowledge of only the two position states.

Let's consider the vertex-face collision case. The displacement of each vertex in the time interval is described by $\Delta\mathbf{x}_i = \mathbf{x}_e - \mathbf{x}_s$. Let's describe the displacement of all the vertices involved in a collision by the configuration space vector $\mathbf{q}$. We define a scalar constraint equation to describe the normal relative displacement of our contact points in equation (13.9). We orient the collision normal at the time of impact, $\hat{\mathbf{n}}_c$, so that equation $C(\mathbf{q}) < 0$ since we know the relative normal motion must be negative for a collision to occur.

[8]When processing a single contact, these expressions might appear intimidating, but they reduce to simple summations.

$$C(\mathbf{q}) = \hat{\mathbf{n}}_\mathbf{c} \cdot [\Delta\mathbf{x}_1 - (u\Delta\mathbf{x}_2 + v\Delta\mathbf{x}_3 + w\Delta\mathbf{x}_4)] \tag{13.9}$$

The gradient wrt. $\mathbf{q}$ of our constraint equation becomes:

$$\nabla\mathbf{C} = [\hat{\mathbf{n}}_c, -u\hat{\mathbf{n}}_c, -v\hat{\mathbf{n}}_c, -w\hat{\mathbf{n}}_c] \tag{13.10}$$

Continuing the derivation in Harmon et al. (2008), we arrive at equation (13.11). The right hand side describes the desired change we want in the normal relative motion. Once we solve for λ, we update our end positions with equation (13.12). Hard constraints from our dynamics solution can be respected, by multiplying $\mathbf{M}^{-1}$ with $\mathbf{S}_i$ and incorporating it into both of these equations[9].

$$\nabla\mathbf{C}\mathbf{M}^{-1}\nabla\mathbf{C}^\mathbf{T}\lambda = \nabla\mathbf{C}\mathbf{q} \tag{13.11}$$

$$\mathbf{x}'_{ie} = \mathbf{x}_{ie} - \mathbf{M}^{-1}\nabla\mathbf{C}^\mathbf{T}\lambda \tag{13.12}$$

At this point, all normal relative motion between the points in contact will vanish. Let $\mathbf{x}_s$ be a vector of the vertex start positions and $d_s = \nabla\mathbf{C}\mathbf{x}_s$ define the initial relative normal distance. We want all relative normal motion to stop when $d_s \leq d_{\text{offset}}$. When $d_s > d_{\text{offset}}$ we must alter the RHS of equation (13.11). If we want the relative normal motion to stop at d_{offset}, we want a post-correction normal displacement magnitude = $d_{\text{offset}} - d_s$. The desired change between our pre-correction and post-correction normal displacement magnitude defines our right hand side and is given by $\nabla\mathbf{C}\mathbf{q} + \nabla\mathbf{C}\mathbf{x}_s - d_{\text{offset}}$. I do not recommend modifying the RHS for trying to achieve post-correction normal offset greater than the initial one. Compensating for the collision offset would add energy, and it is probably best to model a fully dissipative response for deformable bodies. Applying inelastic projection to handle edge-edge collisions is straightforward. If multiple simultaneous contacts are found, this can be extended to to the technique in Harmon et al. (2008). It would result in a linear system of size $N_c \times N_c$ where N_c is the number of simultaneous contacts.

13.9 Resolving CCD Collisions in Chronological Order

In a single time step, there can be many collisions between our two position states, but one cannot process them all at once. Some of the found collisions won't occur if others are resolved first and new collisions can be formed by the resolution of others. We present a summary of our resolution scheme in the pseudo-code below:

[9]If the point of contact on both sides is closer to the constrained particles on each side than the unconstrained ones, the resulting deformation from the response can be quite large. Special care must be taken in these situations.

```
 1 bool HandleContinuousCollisions(CollisionDetector, maxIters)
 2 
 3     CollisionDetector->DetectInitialCollisions()
 4     collisions = CollisionDetector->GetResults()
 5     iter = 0
 6     bool collisionFree = true
 7     bool allRegionsResolved = false
 8     if not collisions.empty():
 9         collisionFree = false
10         GatherCollisionsPerDynamicVertex(collisions)
11         CreateContactRegions(collisions)
12 
13         while !collisionFree and iter < maxIters:
14             // Solve first contacts in each disjoint regions
15             // and update the end position state marking
16             // incident mesh features for the BVH update.
17             SolveContactRegions()
18             iters++
19             allRegionsResolved = AreAllContactRegionsCollisionFree()
20             // Steps for incremental detection
21             CollisionDetector->UpdateBVHsWithMarkedFeatures()
22             CollisionDetector->PerformIncrementalDetection()
23             CollisionDetector->GetResults(newCollisions)
24 
25             // Determine whether collisions in newCollisions
26             // are genuinely new or are already represented
27             // in the contact regions. Remove already discovered
28             // collisions from newCollisions.
29             ProcessContacts(newCollisions)
30             if not newCollisions.empty():
31                 // Contact regions may just expand
32                 // or expand and merge with others
33                 ExpandAndMergeContactRegions(newCollisions)
34             else if allRegionsResolved:
35                 collisionFree = true
36 
37     return collisionFree
```

`HandleContinuousCollisions` takes in our collision detection module and a maximum number of iterations to perform. We perform the initial collision detection pass in line 3. If any collisions were found, we gather the collisions per dynamic vertex and store them chronological order in line 10. We create groups of contacts, called *ContactRegions*, that are connected by shared vertices including ones with *kinematic geometry*. We compute and apply our response for the earliest contacts in each region in `SolveContactRegions` on line 17. Contact regions are solved independently in parallel each iteration. After applying the response, `AreAllContactRegionsCollisionFree` sweeps over the contact regions to see if all of their contained contacts have been resolved. When a collision in the region still occurs we update its Time of Impact(TOI), and barycentric coordinates.

Even when `AreAllContactRegionsCollisionFree` indicates all of our contacts have been resolved, we are not done because the response may have generated new collisions.

In lines 20-23 an incremental detection pass is then performed to discover any new contacts. Our incremental detection consists of first marking all the leaf level nodes of our BVHs as *clean*, and then labeling the nodes that contain a feature with a vertex that has moved by the last response iteration as *dirty*. We propagate the *dirty* flag up the tree from the leaves when updating the BVH extents. During the incremental detection pass we can then ignore descending down any pairs that are both *clean*.

Inside `ProcessContacts`, line 27, we examine if the additional contacts should cause contact regions to be expanded or merged. A new collision should always fall into either of those categories. It is the responsibility of the *ContactRegions* to keep the unresolved collisions they contained in chronological order. We then repeat this process until all the collisions have bee resolved or we have exceeded our `maxIters`. A dynamic vertex being pinched between two opposing kinematic surfaces sometimes cannot be resolved with this method. We will discuss our remedy to that problematic situation and others in §13.10.

If all collisions have not been resolved by `maxIters`, the inelastic projection described in Harmon et al. (2008) can be used to halt all normal relative motion in a contact region. This extra failsafe method is a good idea for systems that do not implement a means for recovery. *Fizt* has a means of recovery for some intersection cases, which will begin to discuss shortly. While CCD and the associated response can make sure collisions aren't missed due to temporal aliasing, there is one drawback. The response is completely decoupled from the dynamics of the material model and excessive deformation may occur as a result.

13.10 Global Intersection Analysis

Character animation often leads to situations where the *kinematic geometry* intersects leaving the simulated surface nowhere to go and can easily lead to constrained situations that can cause it to intersect. To handle these situations, the *Flypapering* technique was introduced in the *Untangling Cloth* paper Baraff and Witkin (2003) [10]. The Global Intersection Analysis, (GIA) algorithm was also introduced in this paper, which we continue to rely heavily on. We give a brief summary of the GIA algorithm in this section and discuss its application to proximity and CCD in section §13.10.

Edge-face intersection pairs are first found using our BVH. Closed contours of intersection are then computed by traversing the intersections and using mesh connectivity. A flood-fill algorithm then colors the regions on both sides of each contour. The GIA algorithm makes the reasonable assumption that the region with the smaller area is the region of intersection, though this is certainly not true in pathological cases. The algorithm also does not support surfaces that are non-manifold. Since not all intersection configurations lead to a closed contour of intersection, we still label the mesh features that are involved in intersection.

[10] As previously noted we now run a volume simulation to deform the character geometry, Wong et al. (2018), before running cloth to remove a majority of *Flypapering* cases.

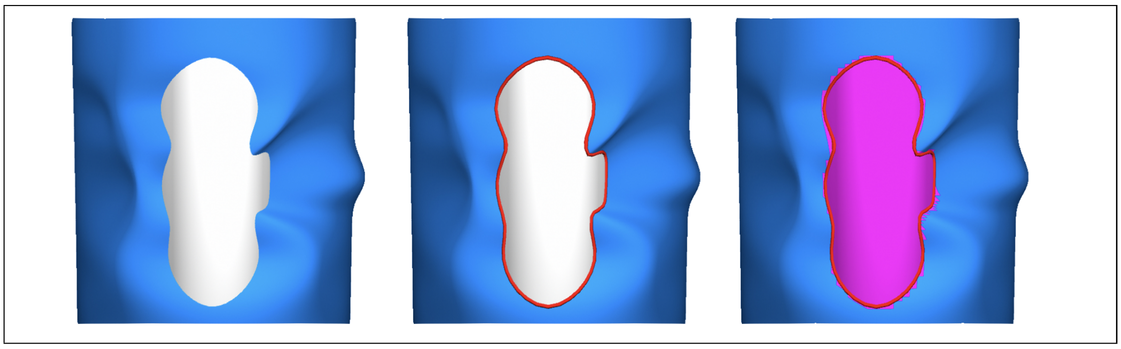


Figure 13.4.: An example the information GIA produces. Left to right: Two intersecting meshes. The intersection contour. The minimum colored area discovered by the flood fill.

13.11 Using GIA with Proximity and CCD

Fizt performs GIA before proximity queries and CCD. GIA is applied to all mesh pairs in the simulator, including the kinematic meshes. We also apply it to the kinematic meshes at the end of the time step.

Kinematic objects are assumed to have meshes that are oriented and closed. When a particle of a deformable body is detected beneath the surface of a kinematic object, a natural first response would be to push it to the desired collision offset of the face it is underneath. When objects are thin or penetrations are deep, this can easily result in the particle being pushed out the wrong side of the surface. This will lead to tangling from which a simulation would rarely ever recover. To alleviate this issue, we look to see if the face we plan to eject through has been labeled as being in a GIA contour, or is a part of an intersection. Only if it is labeled do we deliver the response. This heuristic will not handle all cases, but in practice, particles closer to the intersection region are pushed out first. That response then causes connected particles that are in deeper penetration to emerge from the correct side of the surface.

When two dynamic surfaces are intersecting, we use the GIA coloring information to change our penalty response. In the case of cloth, we do not have a notion of an inside and outside. When a colored particle has a proximity contact with a face of the same color, we instruct our penalty force to pull it through to the appropriate offset distance on the other side. This follows the approach to untangling cloth intersections described in Baraff and Witkin (2003).

The untangling response just described will be thwarted by our CCD pass unless it also uses information from GIA. When CCD discovers a contact between two surfaces that have been labeled to be in intersection by GIA, we ignore the collision so that it will not prevent the penalty response from resolving the situation. Once the GIA intersection is resolved, CCD can do its best at preventing intersection from recurring. By coupling GIA with CCD we can safely allow for recovery without compromising the robustness of our CCD with any local heuristics[11].

There can be scenarios where a dynamic contact cannot be resolved, because it is trapped between kinematic surfaces that are passing through each other. We use the kinematic intersection states of the kinematic object to ignore continuous collisions that involve kinematic features that are pinched, becoming pinched, or becoming unpinched.

Instead of attempting to engineer a collision strategy that is never allowed to fail, *Fizt* has taken an approach of assuming failure is inevitable in the face of production work. What is important is to have the ability to recover from any failures gracefully. While our implementations of GIA, proximity queries and continuous collision could be more robust, by working together they are able to handle many demanding production scenarios.

13.12 Things Not Covered

Fizt has evolved at Pixar over 20 years and there are many features and topics that have not been discussed. The list that follows is far from complete but is meant to provide some more information and perspective.

- Many custom implicit forces and constraints exist to give the artists complex control during asset tailoring and shot work.
- We have many controls to change the rest shape of the cloth over time to deal with drastic character stretch/squash and to help facilitate wrinkles in certain regions.
- Particle masses are not constant to support a consistent look despite drastic changes rest shape area. This is one reason why some newer and faster techniques that exploit a single system factorization are not applicable to us.
- Our cloth material model is rooted in Baraff and Witkin (1998), but has become more complex to allow the artists the control they've needed to provide a wide variety of looks over many films.[12]
- *Fizt* has the ability to cope with fast motion by dialing in the amount of rotation and translation experienced by the simulation relative to the motion of a frame on the character.

There are still a lot of robustness and performance improvements that can be made to *Fizt* and for many it may seem that we are slow to adopt the latest advances in the literature. If physical accuracy was our primary goal, then this would certainly be true. Our production work however prioritizes stability[13], speed and artist directability above physical accuracy. Only physical plausibility is required.

[11]Many were tried before discovering GIA is a far more robust approach.
[12]Perhaps in future versions of this course we will discuss it in detail.
[13]Numerical stability as well as code stability.

There is a lot more to our simulation pipeline than the simulator itself. Our tailoring/asset creation tools exist in Maya, but have had many years of custom development which is still ongoing. Our pre-roll system to simulate characters into their shot pose is another tool that our artists constantly rely on. To evolve and support a system this large requires the hard work of many people from Pixar's Tools-Sim and Research teams.

Chapter 14

Collision Energies

14.1 What Energies?

We've seen a few penalty forces you can apply between vertex-face and edge-edge collisions in §13.4. Usually these take the form of *compute the force, then, like, y'know, spread them out to the vertices.* This isn't quite keeping with the philosophy of the rest of these course notes, where we wrote down an energy first, and then computed some derivatives to obtain a force. Let's do that now.

An energy-based approach to collision penalty forces is entirely possible, and can illuminate what's really going on when just staring at the forces doesn't get you anywhere. Let's take a closer look at some possible collision energies, and the forces that arise from them. These are described here and included in the `HOBAK` source code for experimental and pedagogical purposes. But for reasons we will see *not all of these* are in Fizt.

There's going to be a bunch of calculus slug-festing in the following pages, but reader beware: *just because the derivatives are complicated and fancy-looking doesn't always mean they're better.* When we finally arrive at the actual energies used in Fizt, you might pull your hair out.

14.2 A Vertex-Face Energy

14.2.1 A Position-Based Energy

Let's look at the collision energy from McAdams et al. (2011). It's a spring energy defined in world-space as:

$$\Psi_{\text{Mc}}(\mathbf{x}_p, \mathbf{x}_s) = k(\mathbf{x}_p - \mathbf{x}_s)^T \mathbf{N}(\mathbf{x}_p - \mathbf{x}_s). \tag{14.1}$$

Here, the point $\mathbf{x}_p$ is the vertex in collision, and $\mathbf{x}_s$ is the closest point on the colliding triangle[1] (see Fig. 14.1). The matrix $\mathbf{N}$ is the outer product of the normal of the colliding triangle, $\mathbf{N} = \mathbf{n}\mathbf{n}^T$.

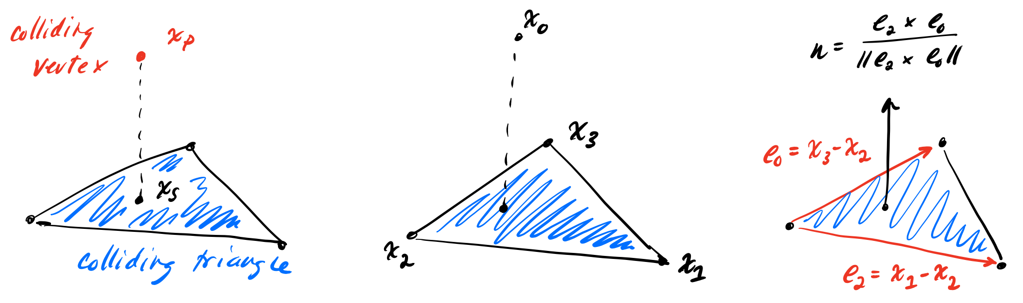


Figure 14.1.: On the **left**, the vertex-face collision configuration. In the **middle**, this is how we'll label the vertices. The vertices on the face are ordered *clockwise*. On the **right**, we can use this ordering to compute a normal for the face.

There's four vertices involved in this collision. Let's label the *vertex* in collision as $\mathbf{x}_0$, and the *triangle* in collision as the vertices $\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3$. We also assume that the triangle's vertices are arranged in *clockwise* order. We can then write the point $\mathbf{x}_s$ on the colliding triangle in terms of the barycentric coordinates (α, β, γ):

$$\mathbf{x}_s = \alpha\mathbf{x}_1 + \beta\mathbf{x}_2 + \gamma\mathbf{x}_3. \tag{14.2}$$

The normal can also be defined using this vertex ordering:

$$\mathbf{e}_0 = \mathbf{x}_3 - \mathbf{x}_2 \tag{14.3}$$
$$\mathbf{e}_1 = \mathbf{x}_0 - \mathbf{x}_2 \tag{14.4}$$
$$\mathbf{e}_2 = \mathbf{x}_1 - \mathbf{x}_2 \tag{14.5}$$
$$\mathbf{n} = \frac{\mathbf{e}_2 \times \mathbf{e}_0}{\|\mathbf{e}_2 \times \mathbf{e}_0\|}. \tag{14.6}$$

Hey wait a minute, since when do we define energies in terms of $\mathbf{x}_p$ and $\mathbf{x}_s$ instead of $\mathbf{F}$? You caught me. I don't have the $\mathbf{F}$-based version of this energy in hand yet. It *should* be possible to define this[2], but it's not there yet, and it's not what's listed in McAdams et al. (2011), so let's just go with the $\mathbf{x}$-based version for now. You should expect a calculus slug-fest ahead.

[1]Revisiting the original McAdams et al. (2011) paper, I now see that I reversed these two labels by accident. The $\mathbf{x}_p$ is the "proxy" point on the triangle, and the $\mathbf{x}_s$ is the vertex in collision. As we'll see in §14.3, I find this whole energy vaguely irritating, so I am NOT FIXING THE LABELS.

[2]Sounds like a good project for a graduate student.

14.2.2 What's the Energy Doing?

It's a little easier to see what this energy is doing if we define the direction

$$\mathbf{t} = \mathbf{x}_0 - \alpha\mathbf{x}_1 - \beta\mathbf{x}_2 - \gamma\mathbf{x}_3. \tag{14.7}$$

The collision energy is then taking this direction, and extracting the component that is normal to the triangle, i.e. see $\mathbf{t}^T\mathbf{n}$ in Fig. 14.2. We want to push the vertex and triangle apart, and *nothing else.* If the vertex and triangle want to slide tangent to each other, they should be welcome to do so. Don't insert any forces that impedes any perfectly acceptable sliding behavior. An entirely equivalent energy is then:

$$\Psi_{\text{Mc}}(\mathbf{x}_p, \mathbf{x}_s) = k(\mathbf{t}^T\mathbf{n})^2. \tag{14.8}$$

We're actually going to use this form of the energy instead, because it organizes the derivatives a little more cleanly.

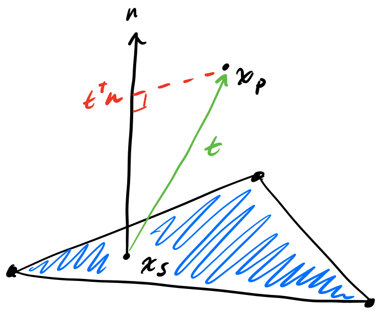


Figure 14.2.: The energy 14.1 is taking the difference $\mathbf{t} = \mathbf{x}_p - \mathbf{x}_s$ and extracting the part that lies along the triangle normal $\mathbf{n}$.

14.2.3 The Collision Force

Let's go ahead and take the derivative, which in turn will give us the collision force. Since we labelled all the vertices in terms of some $\mathbf{x}_i$, we can now define the $\Re^{12}$ vector:

$$\mathbf{x} = \begin{bmatrix} \mathbf{x}_0 \\ \mathbf{x}_1 \\ \mathbf{x}_2 \\ \mathbf{x}_3 \end{bmatrix} \tag{14.9}$$

The derivative them becomes:

$$\frac{\partial \Psi_{\text{Mc}}(\mathbf{x}_p, \mathbf{x}_s)}{\partial \mathbf{x}} = 2k\left(\mathbf{t}^T\mathbf{n}\right)\left[\frac{\partial \mathbf{t}}{\partial \mathbf{x}}^T \mathbf{n} + \frac{\partial \mathbf{n}}{\partial \mathbf{x}}^T \mathbf{t}\right]. \tag{14.10}$$

The $\frac{\partial \mathbf{t}}{\partial \mathbf{x}}$ term is the easy one, and has a relatively simple form:

$$\frac{\partial \mathbf{t}}{\partial \mathbf{x}} = \begin{bmatrix} \mathbf{I}_{3\times 3} & -\alpha \mathbf{I}_{3\times 3} & -\beta \mathbf{I}_{3\times 3} & -\gamma \mathbf{I}_{3\times 3} \end{bmatrix} \in \Re^{3\times 12}. \tag{14.11}$$

The C++ implementation is also straightforward:

```cpp
///////////////////////////////////////////////////////////////////////
///////////////////////////////////////////////////////////////////////
MATRIX3x12 MCADAMS_BARY_COLLISION::vDiffPartial(const VECTOR3& bary)
{
  MATRIX3x12 tPartial;
  tPartial.setZero();
  tPartial(0,0) = tPartial(1,1)  = tPartial(2,2) = 1.0;
  tPartial(0,3) = tPartial(1,4)  = tPartial(2,5) = -bary[0];
  tPartial(0,6) = tPartial(1,7)  = tPartial(2,8) = -bary[1];
  tPartial(0,9) = tPartial(1,10) = tPartial(2,11) = -bary[2];

  return tPartial;
}
```

In contrast, the $\frac{\partial \mathbf{n}}{\partial \mathbf{x}}$ term is known as the *normal derivative* or *normal gradient*, a term that is so excruciatingly painful to compute that it has sent many a student screaming for the hills, racing off to find a monastery/nunnery/gender-neutral temple where they can spend their remaining days trying to scar over the mental claw-marks exacted by this terrifying term. Or, sometimes they ignore the term entirely and hope for the best, spending the rest of their days wondering if every simulator instability they encounter can be traced back to that one time, years ago, when it came time to stand up to that terrifying $\frac{\partial \mathbf{n}}{\partial \mathbf{x}}$ term, and their courage failed.

Fortunately, I have worked this term out for you in Appendix I. With all these terms in hand, the force vector itself is computed as

$$\mathbf{f} = -a \frac{\partial \Psi_{\text{Mc}}(\mathbf{x}_p, \mathbf{x}_s)}{\partial \mathbf{x}}, \tag{14.12}$$

where a denotes the area of the collision, which we set to the rest-state area of the triangle, and 1/3 the rest-state area of the all the triangles in the one-ring of the colliding vertex. There's a negative in front of the whole thing, because remember the force is always the *negative* gradient of the energy.

This produces a vector $\mathbf{f} \in \Re^{12}$, which is the force at each of the four vertices, all stacked all together, and needs to be cut up and distributed to each of the involved vertices. This can all be seen in action in the function:

`TET_MESH::computeVertexFaceCollisionForces(const REAL& collisionStiffness)`

14.2.4 The Collision Hessian

Now we come to the sticky part, and have to compute the Hessian of the collision energy. Just to keep things from running off the end of the page, let's define a gradient term,

$$\mathbf{g} = \frac{\partial \mathbf{t}}{\partial \mathbf{x}}^T \mathbf{n} + \frac{\partial \mathbf{n}}{\partial \mathbf{x}}^T \mathbf{t} \in \Re^{12}. \tag{14.13}$$

Pushing forward with the derivatives, we then get

$$\frac{\partial^2 \Psi_{\text{Mc}}(\mathbf{x}_p, \mathbf{x}_s)}{\partial \mathbf{x}^2} = 2k \left[\mathbf{g}\mathbf{g}^T + \mathbf{t}^T \mathbf{n} \left[\left(\frac{\partial^2 \mathbf{t}}{\partial \mathbf{x}^2} \right)^T \mathbf{n} + \mathbf{t}^T \frac{\partial^2 \mathbf{n}}{\partial \mathbf{x}^2} + \frac{\partial \mathbf{t}}{\partial \mathbf{x}}^T \frac{\partial \mathbf{n}}{\partial \mathbf{x}} + \frac{\partial \mathbf{n}}{\partial \mathbf{x}}^T \frac{\partial \mathbf{t}}{\partial \mathbf{x}} \right] \right]. \tag{14.14}$$

Since $\frac{\partial \mathbf{t}}{\partial \mathbf{x}}$ was just a bunch of constants, it must be the case that $\frac{\partial^2 \mathbf{t}}{\partial \mathbf{x}^2} = 0$. Thus, the Hessian simplifies to:

$$\frac{\partial^2 \Psi_{\text{Mc}}(\mathbf{x}_p, \mathbf{x}_s)}{\partial \mathbf{x}^2} = 2k \left[\mathbf{g}\mathbf{g}^T + \mathbf{t}^T \mathbf{n} \left(\mathbf{t}^T \frac{\partial^2 \mathbf{n}}{\partial \mathbf{x}^2} + \frac{\partial \mathbf{t}}{\partial \mathbf{x}}^T \frac{\partial \mathbf{n}}{\partial \mathbf{x}} + \frac{\partial \mathbf{n}}{\partial \mathbf{x}}^T \frac{\partial \mathbf{t}}{\partial \mathbf{x}} \right) \right]. \tag{14.15}$$

We have everything needed for this expression in hand, *except* for the $\frac{\partial^2 \mathbf{n}}{\partial \mathbf{x}^2}$ term. Oh, it's never easy, is it? First you slay Grendel, and then around the corner comes his mom, Angelina Jolie. Don't worry. It's okay. I did it for you.

The *normal Hessian* is described in grisly detail in the second part of Appendix I. Now you have a $\Re^{12\times 12}$ matrix to dice up and distribute into the global matrix. You can see this all in action in the function:

```
TET_MESH::computeVertexFaceCollisionClampedHessian
```

14.3 Another (Better? Identical?) Vertex-Face Energy

14.3.1 Another Position-Based Energy

Something irritates me about the $\Psi_{\text{Mc}}(\mathbf{x}_p, \mathbf{x}_s)$ energy (sorry Alex, nothing personal, reasonable people can disagree, right?). Why do we even need some $\mathbf{x}_s$ that is a barycentric combination of $\mathbf{x}_1, \mathbf{x}_2$ and $\mathbf{x}_3$?

Why can't we just use the closest point to $\mathbf{x}_p$ on the triangle (let's call it $\mathbf{x}_t$)? If we use the distance between $\mathbf{x}_p$ and the closest point for our spring, it still seems pretty much like Hooke's Law, right? I drew it all out in Fig. 14.3 if you're still not following.

It takes a little effort to get the point $\mathbf{x}_t$, but we'll see that by the time we get to the energy, everything will simplify nicely. Projecting $\mathbf{x}_p$ onto the triangle spanned by $\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3$ is

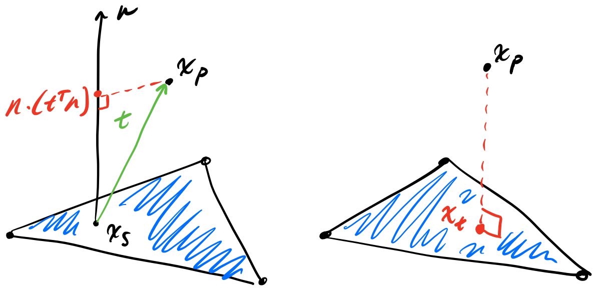


Figure 14.3.: Instead of using some barycentric-coordinate-defined point $\mathbf{x}_s$ (left), why not find the closest point $\mathbf{x}_t$ on the triangle (right), and just use that?

some straightforward vector arithmetic:

$$\mathbf{x}_t = \mathbf{x}_p - \left(\mathbf{n}^T(\mathbf{x}_p - \mathbf{x}_2)\right)\mathbf{n} \tag{14.16}$$

$$= \mathbf{x}_0 - \left(\mathbf{n}^T(\mathbf{x}_0 - \mathbf{x}_2)\right)\mathbf{n}. \tag{14.17}$$

Treating this as our $\mathbf{x}_s$ in the McAdams et al. (2011) energy (but no barycentric coordinates needed!) we start with something messy, but it cleans up nicely:

$$\begin{aligned}
\Psi_{\text{vf}}(\mathbf{x}) &= k(\mathbf{x}_p - \mathbf{x}_t)^T\mathbf{n}\mathbf{n}^T(\mathbf{x}_p - \mathbf{x}_t) \\
&= k\left(\mathbf{x}_0 - \left(\mathbf{x}_0 - \left(\mathbf{n}^T(\mathbf{x}_0 - \mathbf{x}_2)\right)\mathbf{n}\right)\right)^T \mathbf{n}\mathbf{n}^T \left(\mathbf{x}_0 - \left(\mathbf{x}_0 - \left(\mathbf{n}^T(\mathbf{x}_0 - \mathbf{x}_2)\right)\mathbf{n}\right)\right) \\
&= k\left(\mathbf{n}^T(\mathbf{x}_0 - \mathbf{x}_2)\mathbf{n}\right)^T \mathbf{n}\mathbf{n}^T \left(\mathbf{n}^T(\mathbf{x}_0 - \mathbf{x}_2)\mathbf{n}\right) \\
&= k\left(\mathbf{n}^T(\mathbf{x}_0 - \mathbf{x}_2)\right)^2 \mathbf{n}^T\mathbf{n}\mathbf{n}^T\mathbf{n}
\end{aligned}$$

In the last line, since $\mathbf{n}^T\mathbf{n} = 1$, we can simplify one last piece to get our final energy:

$$\Psi_{\text{vf}}(\mathbf{x}) = k\left(\mathbf{n}^T\mathbf{t}_{\text{vf}}\right)^2 \tag{14.18}$$

$$\mathbf{t}_{\text{vf}} = (\mathbf{x}_0 - \mathbf{x}_2) \tag{14.19}$$

This ... is actually really close to Eqn. 14.8. We can even think of it like we just decided to use the barycentric coordinate $(\alpha, \beta, \gamma) = (0, 1, 0)$ every single time. The dot product against $\mathbf{n}$, will always project off any component that's not normal to the triangle, so it actually doesn't matter what (α, β, γ) we use.[3] If we just pin it to one thing, and then we don't have to mess around with (α, β, γ) at all.

Everything then becomes ever-so-slightly simpler.

[3]Well, you can still use something dumb like $(\alpha, \beta, \gamma) = (0, 0, 0)$ and get yourself in trouble. It doesn't matter what *valid* barycentric coordinate you use.

14.3.2 The Collision Force

The force then becomes slightly simpler. Taking the derivative, we get:

$$\frac{\partial \Psi_{\text{vf}}(\mathbf{x})}{\partial \mathbf{x}} = 2k\left(\mathbf{t}_{\text{vf}}^T \mathbf{n}\right)\left[\frac{\partial \mathbf{t}_{\text{vf}}}{\partial \mathbf{x}}^T \mathbf{n} + \frac{\partial \mathbf{n}}{\partial \mathbf{x}}^T \mathbf{t}_{\text{vf}}\right]. \tag{14.20}$$

The $\frac{\partial \mathbf{t}_{\text{vf}}}{\partial \mathbf{x}}$ term is now slightly simpler than the $\frac{\partial \mathbf{t}}{\partial \mathbf{x}}$ term in Eqn. 14.11, since we can just freeze it to $(\alpha, \beta, \gamma) = (0, 1, 0)$. We get:

$$\frac{\partial \mathbf{t}_{\text{vf}}}{\partial \mathbf{x}} = \begin{bmatrix} \mathbf{I}_{3\times 3} & \mathbf{0} & -\mathbf{I}_{3\times 3} & \mathbf{0} \end{bmatrix} \in \Re^{3\times 12}. \tag{14.21}$$

Everything else, like the $\frac{\partial \mathbf{n}}{\partial \mathbf{x}}$ term, is exactly the same as before.

14.3.3 The Collision Hessian

Everything rolls forward with the Hessian as well. We define another gradient term,

$$\mathbf{g}_{\text{vf}} = \frac{\partial \mathbf{t}_{\text{vf}}}{\partial \mathbf{x}}^T \mathbf{n} + \frac{\partial \mathbf{n}}{\partial \mathbf{x}}^T \mathbf{t}_{\text{vf}} \in \Re^{12}, \tag{14.22}$$

and plug it into the full Hessian

$$\frac{\partial^2 \Psi_{\text{vf}}(\mathbf{x})}{\partial \mathbf{x}^2} = 2k\left[\mathbf{g}_{\text{vf}}\mathbf{g}_{\text{vf}}^T + \mathbf{t}_{\text{vf}}^T\mathbf{n}\left(\mathbf{t}_{\text{vf}}^T \frac{\partial^2 \mathbf{n}}{\partial \mathbf{x}^2} + \frac{\partial \mathbf{t}_{\text{vf}}}{\partial \mathbf{x}}^T \frac{\partial \mathbf{n}}{\partial \mathbf{x}} + \frac{\partial \mathbf{n}}{\partial \mathbf{x}}^T \frac{\partial \mathbf{t}_{\text{vf}}}{\partial \mathbf{x}}\right)\right].$$

No new terms appeared, except for the $\frac{\partial \mathbf{t}_{\text{vf}}}{\partial \mathbf{x}}$ that is slightly simpler than $\frac{\partial \mathbf{t}}{\partial \mathbf{x}}$, and so we're all done. The moral here is that if you're doing the proxy-point sprinkling in McAdams et al. (2011), go ahead and use Ψ_{Mc}. If you're just trying to get a vanilla vertex-face collision energy though, I don't see any reason not to use the simpler Ψ_{vf}.

14.4 Non-Zero Rest-Length Vertex-Face Energies

So far, the two energies we've seen have assumed that you want a zero rest-length. This isn't true in general though. Often you want some small air gap in between your collision so that stuff like clothing can slip through (see Wong et al. (2018)), or just so that stuff isn't visually inter-penetrating as much.

Fortunately, adding a small non-zero rest-length ε to all the preceding energies is straightforward. Without further ado, here it is for the McAdams et al. (2011) energy:

$$\Psi_{\text{Mc}}(\mathbf{x}_p, \mathbf{x}_s) = k\left(\mathbf{t}^T\mathbf{n} - \varepsilon\right)^2 \tag{14.23}$$

$$\frac{\partial \Psi_{\text{Mc}}(\mathbf{x}_p, \mathbf{x}_s)}{\partial \mathbf{x}} = 2k\left(\mathbf{t}^T\mathbf{n} - \varepsilon\right)\left[\frac{\partial \mathbf{t}}{\partial \mathbf{x}}^T \mathbf{n} + \frac{\partial \mathbf{n}}{\partial \mathbf{x}}^T \mathbf{t}\right] \tag{14.24}$$

$$\frac{\partial^2 \Psi_{\text{Mc}}(\mathbf{x}_p, \mathbf{x}_s)}{\partial \mathbf{x}^2} = 2k\left[\mathbf{g}\mathbf{g}^T + \left(\mathbf{t}^T\mathbf{n} - \varepsilon\right)\left(\mathbf{t}^T\frac{\partial^2 \mathbf{n}}{\partial \mathbf{x}^2} + \frac{\partial \mathbf{t}}{\partial \mathbf{x}}^T\frac{\partial \mathbf{n}}{\partial \mathbf{x}} + \frac{\partial \mathbf{n}}{\partial \mathbf{x}}^T\frac{\partial \mathbf{t}}{\partial \mathbf{x}}\right)\right]. \tag{14.25}$$

And equivalently, here it is for our Another-Better-Identical energy:

$$\Psi_{\text{vf}}(\mathbf{x}) = k\left(\mathbf{t}_{\text{vf}}^T\mathbf{n} - \varepsilon\right)^2 \tag{14.26}$$

$$\frac{\partial \Psi_{\text{vf}}(\mathbf{x})}{\partial \mathbf{x}} = 2k\left(\mathbf{t}_{\text{vf}}^T\mathbf{n} - \varepsilon\right)\left[\frac{\partial \mathbf{t}_{\text{vf}}}{\partial \mathbf{x}}^T \mathbf{n} + \frac{\partial \mathbf{n}}{\partial \mathbf{x}}^T \mathbf{t}_{\text{vf}}\right] \tag{14.27}$$

$$\frac{\partial^2 \Psi_{\text{vf}}(\mathbf{x})}{\partial \mathbf{x}^2} = 2k\left[\mathbf{g}_{\text{vf}}\mathbf{g}_{\text{vf}}^T + \left(\mathbf{t}_{\text{vf}}^T\mathbf{n} - \varepsilon\right)\left(\mathbf{t}_{\text{vf}}^T\frac{\partial^2 \mathbf{n}}{\partial \mathbf{x}^2} + \frac{\partial \mathbf{t}_{\text{vf}}}{\partial \mathbf{x}}^T\frac{\partial \mathbf{n}}{\partial \mathbf{x}} + \frac{\partial \mathbf{n}}{\partial \mathbf{x}}^T\frac{\partial \mathbf{t}_{\text{vf}}}{\partial \mathbf{x}}\right)\right]. \tag{14.28}$$

Complete C++ implementations of these are available in the `HOBAK` classes `MCADAMS_COLLISION` and `VERTEX_FACE_COLLISION`.

Sure looks like these two energies are functionally the same. I dropped an elastic `E` in `HOBAK` using both of them, and they produced the same result. You can see in Fig. 14.4.

14.5 The *Actual* Vertex-Face Energy Used in Fizt

14.5.1 Just To Drive You Crazy

Just to drive you crazy, I'm now going to casually mention that neither of the energies we went over are actually in Fizt. The energy corresponding to the penalty force used in production is actually this one[4]:

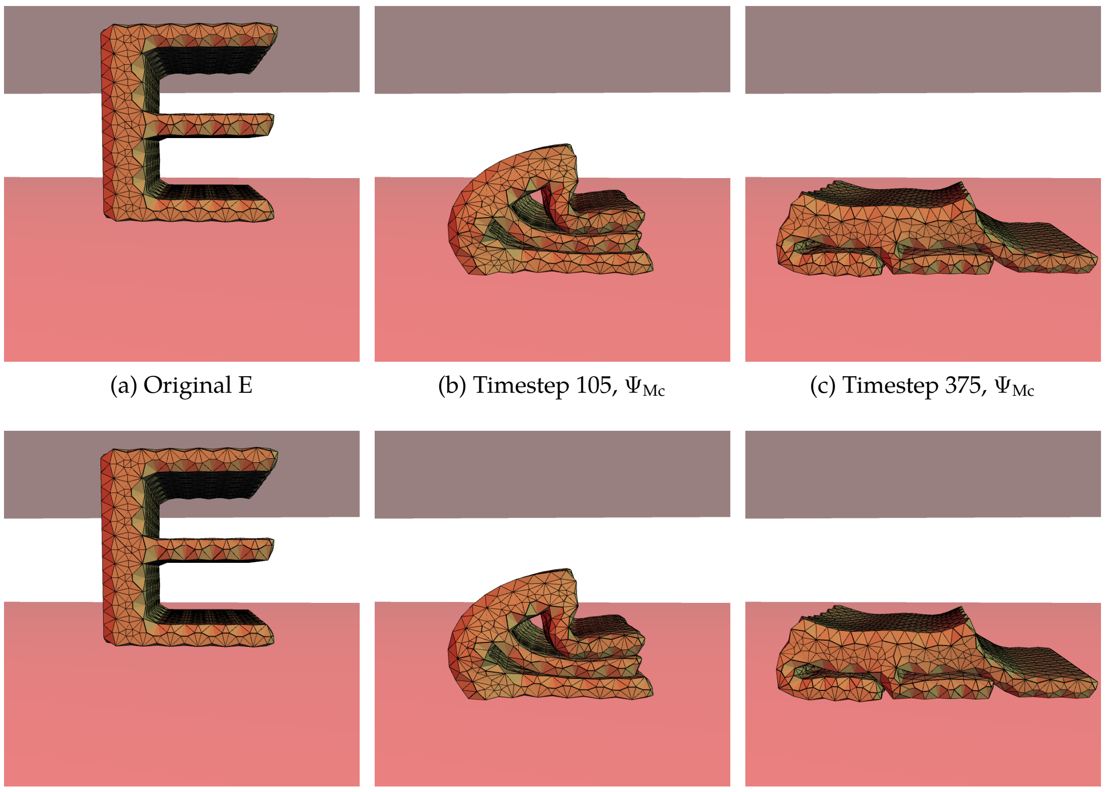
$$\Psi_{\sqrt{\text{vf}}} = \frac{k}{2}(\|\mathbf{t}\| - \varepsilon)^2 \tag{14.29}$$

$$\mathbf{t} = \mathbf{x}_p - \mathbf{x}_t. \tag{14.30}$$

Whaaat. There's no triangle normal anywhere in this energy! What was the point of all these difficult derivatives we just saw! The gradient then works out to:

$$\frac{\partial \Psi_{\sqrt{\text{vf}}}}{\partial \mathbf{x}} = k(\|\mathbf{t}\| - \varepsilon)\frac{\mathbf{t}}{\|\mathbf{t}\|}\frac{\partial \mathbf{t}}{\partial \mathbf{x}}$$

[4] I've labelled constants to be consistent with this chapter, not Chapter 13, so the namings won't line up completely.



(a) Original E (b) Timestep 105, Ψ_{Mc} (c) Timestep 375, Ψ_{Mc}

(d) Original E (e) Timestep 105, Ψ_{vf} (f) Timestep 375, Ψ_{vf}

Figure 14.4.: Dropping an E-shaped mesh on the ground, using the vertex-face collision energies Ψ_{Mc} and Ψ_{vf}. Do you see a difference? I don't, even when I flip between the two in Photoshop. Pretty sure the energies are identical.

Compare this to the spring force portion of Eqn. 13.1 (without damping), which I've reproduced here for your convenience:

$$\mathbf{f}_a = -(k_s(\|\Delta\mathbf{x}\| - d_{\text{offset}})\frac{\Delta\mathbf{x}}{\|\Delta\mathbf{x}\|}. \tag{14.31}$$

It's the same, except for the negative in front[5], and the $\frac{\partial\mathbf{t}}{\partial\mathbf{x}}$ matrix on the end, which serves as the *let's spread the force out to the vertices* multiply.[6]

It's just a spring force between our vertex and the collision point on the triangle. Our original objection to this style of energy is that you only want the penalty force to push

[5]Remember we multiply $\frac{\partial\Psi_{\sqrt{\text{vf}}}}{\partial\mathbf{x}}$ by negative area to get the force.

[6]If you're a big fan of McAdams et al. (2011), you might remember that their complete energy was $\Psi = k(\mathbf{x}_p - \mathbf{x}_s)^T\left((1-\alpha)\mathbf{N} + \alpha\mathbf{I}\right)(\mathbf{x}_p - \mathbf{x}_s)$, and have a sneaking suspicion that $\Psi_{\sqrt{\text{vf}}}$ corresponds to the $\alpha = 1$ case. If you look closer, Ψ_{Mc} never takes the norm of $\mathbf{t}$, so there's a square root missing. They're close, but not quite the same.

in the *normal* direction. Like I said when we first formulated the Ψ_{Mc} energy, you can get otherwise get additional non-physical stickiness in the surface tangent directions, which isn't really the behavior we want.

This is certainly a valid objection, but in practice, this artifact doesn't seem to appear all that much. We conjecture that this is because each force only appears for the span of a single timestep, after which we run another proximity query, and the $\mathbf{x}_t$ point gets updated. Without a consistent $\mathbf{x}_t$ direction to apply the non-physical stickiness in, the artifact is much less visible. Your mileage may vary, especially if you decide to use really humungous timestep sizes, but this is what we've seen in production. Fortunately, HOBAK has all of these energies implemented, so if you start seeing stickiness, you can always just toggle to a different energy to see if that's the culprit.

One situation where it *does* makes intuitive sense to use Ψ_{diff} instead of Ψ_{Mc} is when the vertex-face collision is essentially a vertex-edge collision (Fig. 14.5, left). In this

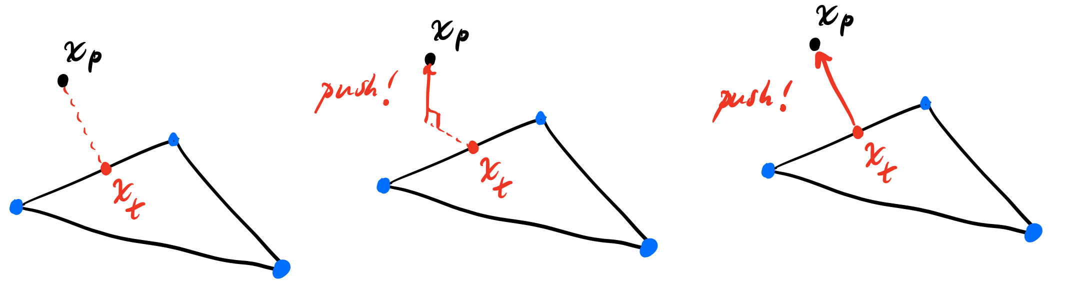


Figure 14.5.: If the collision vertex $\mathbf{x}_p$ is not above the triangle, $\mathbf{x}_t$ lies on an edge, and it becomes a vertex-edge collision (left). In that case, which direction should be push? Still along the normal direction (middle)? Seems a little weird. Why not just in the direction of the difference, $(\mathbf{x}_p - \mathbf{x}_t)$ (right)?

vertex-edge case, should we still just push in the normal direction, which hovers weirdly in a no-man's land outside the triangle (Fig. 14.5, middle), or should we just push in the $(\mathbf{x}_p - \mathbf{x}_t)$ direction (Fig. 14.5, right), which does what we want, and just pushes the vertex and face apart? The last one does seem appealing.

The Hessian works out to something grisly-looking:

$$\frac{\partial^2 \Psi_{\sqrt{\text{vf}}}}{\partial \mathbf{x}^2} = k\left[\left(\frac{1}{\mathbf{t}^T\mathbf{t}} - \frac{\|\mathbf{t}\| - \varepsilon}{(\mathbf{t}^T\mathbf{t})^{\frac{3}{2}}}\right)\mathbf{g}\mathbf{g}^T + \frac{\|\mathbf{t}\| - \varepsilon}{\|\mathbf{t}\|}\frac{\partial \mathbf{t}}{\partial \mathbf{x}}^T\frac{\partial \mathbf{t}}{\partial \mathbf{x}}\right]$$

$$\mathbf{g} = \frac{\partial \mathbf{t}}{\partial \mathbf{x}}^T \mathbf{t}.$$

14.5.2 There's A Reversal Problem

The $\Psi_{\sqrt{\text{vf}}}$ energy has a problem that's very similar to Ψ_{ASqrt} from Chapter 8. Since it primarily consists of a $\|\mathbf{t}\| = \sqrt{\mathbf{t}^T\mathbf{t}}$ term, it squares away the direction, and then just assumes that everything should be positive when it then takes the square root. This is not a good assumption!

If the colliding vertex has not yet penetrated the face, then the force's direction is just fine (Fig. 14.6, left), and this direction will push the vertex and face apart. However, if the vertex and face have already collided, and the vertex is inside the volume (Fig. 14.6, middle), then pushing the vertex and face further apart will push the vertex *deeper* into the model, making the collision worse. In this case, you want to *reverse* the direction of the force, and have the vertex and face *attract* each other, until the two are no longer in collision (Fig. 14.6, right). Then you can start repulsing again. All this can be accomplished

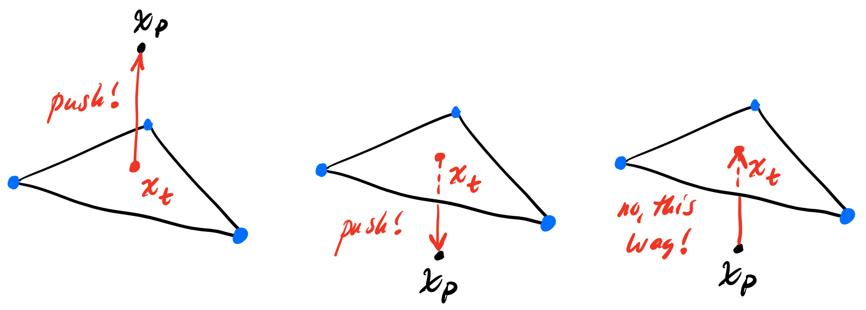


Figure 14.6.: If the collision vertex $\mathbf{x}_p$ has not already passed through the triangle, pushing the vertex and face apart works just fine (left). If $\mathbf{x}_p$ has already penetrated into the triangle though, this will push $\mathbf{x}_p$ deeper into the volume, making the collision worse (middle). In this case, reverse the direction and have the vertex and face *attract* each other until the collision has been untangled.

by switching a minus to a plus in the energy:

$$\Psi^-_{\sqrt{\text{vf}}} = \frac{k}{2}(\|\mathbf{t}\| + \varepsilon)^2 \tag{14.32}$$

$$\frac{\partial \Psi^-_{\sqrt{\text{vf}}}}{\partial \mathbf{x}} = k(\|\mathbf{t}\| + \varepsilon)\frac{\mathbf{t}}{\|\mathbf{t}\|}\frac{\partial \mathbf{t}}{\partial \mathbf{x}} \tag{14.33}$$

$$\frac{\partial^2 \Psi^-_{\sqrt{\text{vf}}}}{\partial \mathbf{x}^2} = k\left[\left(\frac{1}{\mathbf{t}^T\mathbf{t}} - \frac{\|\mathbf{t}\| + \varepsilon}{(\mathbf{t}^T\mathbf{t})^{\frac{3}{2}}}\right)\mathbf{g}\mathbf{g}^T + \frac{\|\mathbf{t}\| + \varepsilon}{\|\mathbf{t}\|}\frac{\partial \mathbf{t}}{\partial \mathbf{x}}^T\frac{\partial \mathbf{t}}{\partial \mathbf{x}}\right] \tag{14.34}$$

The decision to invoke the $\Psi^-_{\sqrt{\text{vf}}}$ or $\Psi_{\sqrt{\text{vf}}}$ energy can be determined using a normal test: compute the normal of the colliding triangle, and see if the colliding vertex has a positive or negative projection onto it. The implementation of this test can be found in `bool VERTEX_FACE_SQRT_COLLISION::reverse`.

The $\Psi_{\sqrt{\text{vf}}}$ energy is definitely *not* equivalent to Ψ_{vf} or Ψ_{Mc}, since there's no normal filtering. The results can be qualitatively similar, but you can see the difference in Fig. 14.7.

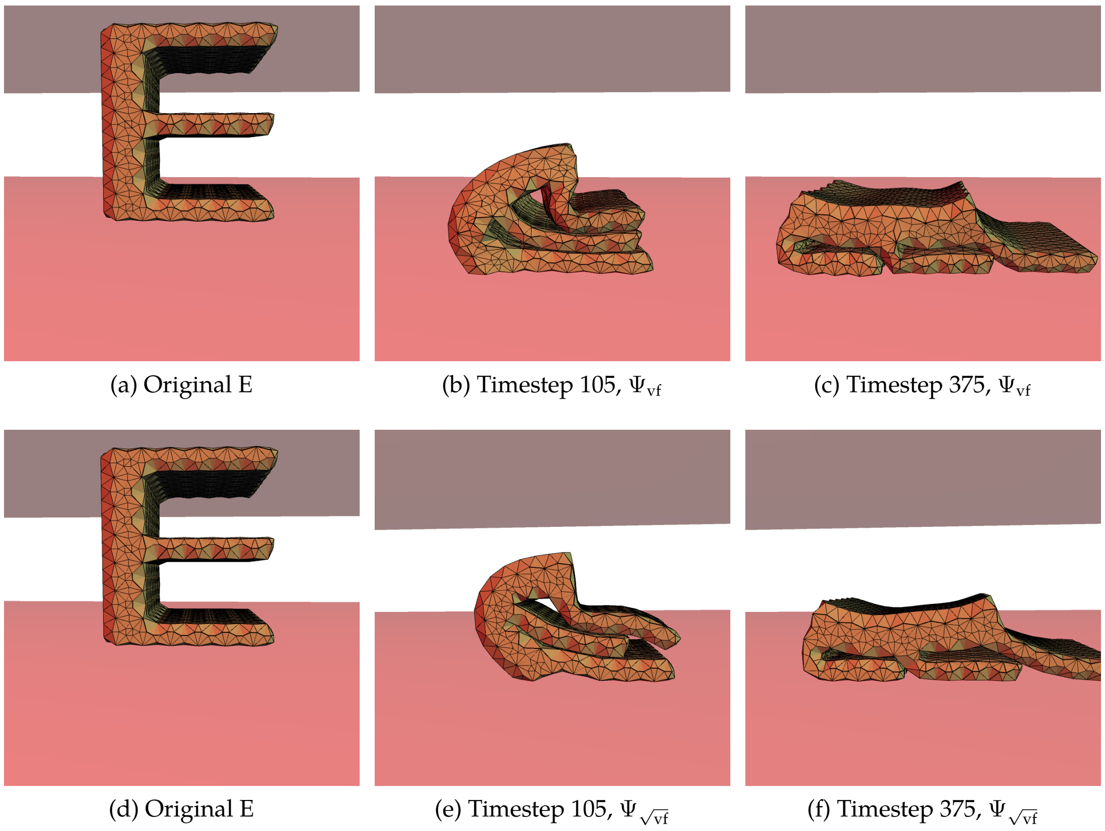

(a) Original E (b) Timestep 105, Ψ_{vf} (c) Timestep 375, Ψ_{vf}
(d) Original E (e) Timestep 105, $\Psi_{\sqrt{\text{vf}}}$ (f) Timestep 375, $\Psi_{\sqrt{\text{vf}}}$

Figure 14.7.: Dropping an E-shaped mesh on the ground, using the vertex-face collision energies Ψ_{vf} and $\Psi_{\sqrt{\text{vf}}}$. The energies are clearly not exactly the same, though they give the same overall behavior.

14.6 An Edge-Edge Energy

14.6.1 The Collision Energy

Vertex-face collisions are great, but they're only half of the story. If you only handle this type of collision, some irritating artifacts will slip on past, particularly when simulating cloth.[7] To get a fuller story, let's take a look at edge-edge collision energies as well.

If you have two close-together edges that you want to push apart, you need to pick a

[7] All that said, edge-edge collisions didn't appear in a Pixar movie until *Coco* (2015), so you can still make movies without it. Your artists might be doing a lot of dirty, expensive, behind-the-scenes clean-up though.

direction to push. There's no triangle involved, so taking its normal and setting that as the direction is out of the question. A bunch of papers (e.g. Provot (1997); Harmon et al. (2008); Wang (2014); Han et al. (2021)) propose taking the *cross product* between the two edges and using that as the more-or-less normal direction. That sounds fine. Let's go with that. This *does* mean that once again our ugly foe the normal derivative will rear its horrid head, but that's okay, the tools we used to tame it before can be used again.

Let's take a look at this cross-product-based edge-edge collision energy:

$$\Psi_{\mathbf{e}\times\mathbf{e}} = k \cdot \left(\varepsilon + (\mathbf{x}_b - \mathbf{x}_a)^T \mathbf{n}\right)^2. \tag{14.35}$$

The variable ε has the same meaning as the non-zero rest-length from §14.4. To make sense of the rest, we need to define a few geometric quantities:

$$\begin{aligned}
\mathbf{x}_a &= a_0\mathbf{x}_0 + a_1\mathbf{x}_1 \\
\mathbf{x}_b &= b_0\mathbf{x}_2 + b_1\mathbf{x}_3 \\
\mathbf{n} &= \mathbf{e}_1 \times \mathbf{e}_0 \\
\mathbf{e}_0 &= \mathbf{x}_1 - \mathbf{x}_0 \\
\mathbf{e}_1 &= \mathbf{x}_3 - \mathbf{x}_2.
\end{aligned}$$

The points $\mathbf{x}_a$ and $\mathbf{x}_b$ are the two closest points between the two edges, computed using the (a_0, a_1) and (b_0, b_1) coordinates returned by the line intersection test of Rhodes (2001).[8] I drew the whole thing out for you in Fig. 14.8.

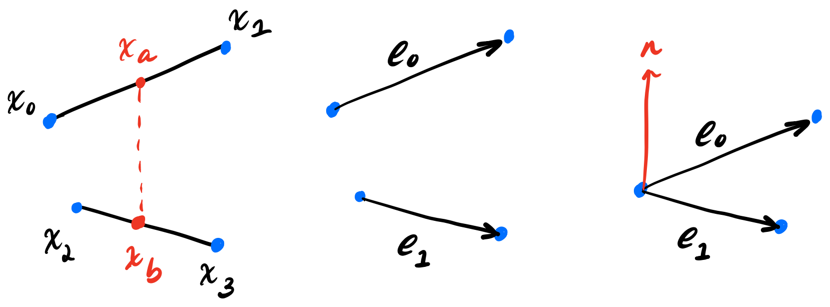


Figure 14.8.: Here's the geometry of edge-edge collisions. We have two edges, $(\mathbf{x}_0, \mathbf{x}_1)$ and $(\mathbf{x}_2, \mathbf{x}_3)$, and the closest points between them are $\mathbf{x}_a$ and $\mathbf{x}_b$ (left). We can define the *directions* of the edges as $\mathbf{e}_0$ and $\mathbf{e}_1$ (middle). The *normal* direction, i.e. the direction in which we want to apply the force, is then $\mathbf{e}_1 \times \mathbf{e}_0 = \mathbf{n}$ (right).

[8] I found the source here:
`https://github.com/UG4/ugcore/blob/master/ugbase/common/math/misc/lineintersect_utils.cpp`.

14.6.2 The Collision Gradient

Let's take the gradient and see what hideousness it reveals:

$$\frac{\partial \Psi_{\mathbf{e}\times\mathbf{e}}}{\partial \mathbf{x}} = 2k\left(\varepsilon + (\mathbf{x}_b - \mathbf{x}_a)^T \mathbf{n}\right)\left[\frac{\partial \mathbf{x}_b - \mathbf{x}_a}{\partial \mathbf{x}}\mathbf{n} + (\mathbf{x}_b - \mathbf{x}_a)\frac{\partial \mathbf{n}}{\partial \mathbf{x}}\right]. \tag{14.36}$$

This is *not bad*. Only two new terms emerged, and they're not that new. First, a difference matrix that looks a whole lot like $\frac{\partial \mathbf{t}}{\partial \mathbf{x}}$ from the vertex-face case:

$$\frac{\partial \mathbf{x}_b - \mathbf{x}_a}{\partial \mathbf{x}} = \begin{bmatrix} -a_0 \mathbf{I}_{3\times 3} & -a_1 \mathbf{I}_{3\times 3} & b_0 \mathbf{I}_{3\times 3} & b_1 \mathbf{I}_{3\times 3} \end{bmatrix}. \tag{14.37}$$

Second, our old frenemy $\frac{\partial \mathbf{n}}{\partial \mathbf{x}}$ also makes an appearance. But that's okay, we already figured this out for the vertex-face case, right? Noooot exactly. In the vertex-face case, we were computing the normal of a triangle, which involved three vertices. In this case, the normal is computed using two independent edges, which involves four vertices. A bunch of the indices change, and some entries that were zero now become non-zero, but nothing fundamentally new shows up if you just follow the same process as before.

At any rate, I worked it all out for you in the back half of Appendix I. Just remember to plug the code from Fig. I.10 into the $\frac{\partial \mathbf{n}}{\partial \mathbf{x}}$ term, **not** the code from Fig. I.4.

14.6.3 The Collision Hessian

Now let's push ahead with the Hessian. Similar to the vertex-face case, it helps to define a gradient term:

$$\mathbf{g}_{\mathbf{e}\times\mathbf{e}} = \frac{\partial (\mathbf{x}_b - \mathbf{x}_a)^T \mathbf{n}}{\partial \mathbf{x}} = \frac{\partial \mathbf{x}_b - \mathbf{x}_a}{\partial \mathbf{x}}^T \mathbf{n} + \frac{\partial \mathbf{n}}{\partial \mathbf{x}}^T (\mathbf{x}_b - \mathbf{x}_a). \tag{14.38}$$

The complete Hessian then becomes:

$$\frac{\partial^2 \Psi_{\mathbf{e}\times\mathbf{e}}}{\partial \mathbf{x}^2} = 2k\left[\mathbf{g}_{\mathbf{e}\times\mathbf{e}}\mathbf{g}_{\mathbf{e}\times\mathbf{e}}^T + \left((\varepsilon + \mathbf{x}_b - \mathbf{x}_a)^T \mathbf{n}\right)\left((\mathbf{x}_b - \mathbf{x}_a)^T \frac{\partial^2 \mathbf{n}}{\partial \mathbf{x}^2} + \frac{\partial (\mathbf{x}_b - \mathbf{x}_a)}{\partial \mathbf{x}}^T \frac{\partial \mathbf{n}}{\partial \mathbf{x}} + \frac{\partial \mathbf{n}}{\partial \mathbf{x}}^T \frac{\partial (\mathbf{x}_b - \mathbf{x}_a)}{\partial \mathbf{x}}\right)\right].$$

In the end, it's not actually that different from the vertex-face Hessians, if you compare them term-by-term. Line it up against Eqn. 14.28, for example. The only complication is the $\frac{\partial^2 \mathbf{n}}{\partial \mathbf{x}^2}$ term, which is slightly different in the edge-edge case, but I've already worked it out for you (again) in Appendix I (again). This time, remember to use the code from Fig. I.12 in the $\frac{\partial^2 \mathbf{n}}{\partial \mathbf{x}^2}$ term, **not** the code from Fig. I.8. All done.

14.6.4 You Have to Reverse It Sometimes

Just like with the $\Psi_{\sqrt{\text{vf}}}$ energy with vertex-face collisions, sometimes you have to reverse the direction of edge-edge collision forces.

In the edge-edge case, if two edges are so close together that they're actually intersecting the surrounding triangles, things can go badly wrong. Similar to the vertex-face case, instead of pushing the two edges *apart* in this situation, you want to push the two edges *together*. Once the triangles are no longer in intersection, *then* switch back to pushing the edges apart. These two scenarios are shown in Fig. 14.9.

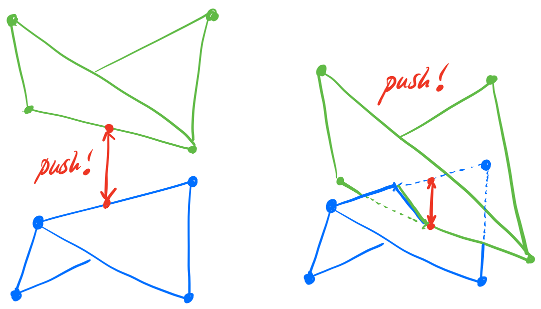


Figure 14.9.: Similar to the vertex-face cases in Fig. 14.6, if the triangles adjacent to the two edges are not in collision, you can just apply a force that *repulses* the two edges (left). You shouldn't do that if the triangles are overlapping (right). In that case, pushing the edges apart will *make things worse*, and cause the triangles to overlap even more. Instead, let's *reverse* the force direction, and have the two edges *attract* each other. Once the triangles stop overlapping, switch back to repulsion.

Fortunately, this can again be accomplished by just flipping the signs in a bunch of places in the energy and its derivatives:

$$
\begin{aligned}
\mathbf{t}_{\mathbf{e}} &= (\mathbf{x}_b - \mathbf{x}_a) \\
\Psi^{-}_{\mathbf{e}\times\mathbf{e}} &= k \cdot \left(\varepsilon - \mathbf{t}_{\mathbf{e}}^T \mathbf{n}\right)^2 \\
\frac{\partial \Psi^{-}_{\mathbf{e}\times\mathbf{e}}}{\partial \mathbf{x}} &= -2k \left(\varepsilon - \mathbf{t}_{\mathbf{e}}^T \mathbf{n}\right) \left[\frac{\partial \mathbf{t}_{\mathbf{e}}}{\partial \mathbf{x}} \mathbf{n} + \mathbf{t}_{\mathbf{e}} \frac{\partial \mathbf{n}}{\partial \mathbf{x}}\right] \\
\frac{\partial^2 \Psi^{-}_{\mathbf{e}\times\mathbf{e}}}{\partial \mathbf{x}^2} &= -2k \left[\mathbf{g}_{\mathbf{e}\times\mathbf{e}} \mathbf{g}^T_{\mathbf{e}\times\mathbf{e}} + \left((\varepsilon - \mathbf{t}_{\mathbf{e}})^T \mathbf{n}\right) \left(\mathbf{t}_{\mathbf{e}}^T \frac{\partial^2 \mathbf{n}}{\partial \mathbf{x}^2} + \frac{\partial \mathbf{t}_{\mathbf{e}}}{\partial \mathbf{x}}^T \frac{\partial \mathbf{n}}{\partial \mathbf{x}} + \frac{\partial \mathbf{n}}{\partial \mathbf{x}}^T \frac{\partial \mathbf{t}_{\mathbf{e}}}{\partial \mathbf{x}}\right)\right].
\end{aligned}
$$

For this reason, the `EDGE_COLLISION` class has both `gradient` and `gradientNegated` functions, as well as `hessian` and `hessianNegated` functions. During collision detection in `TET_MESH::computeEdgeEdgeCollisions`, if two edges are found to be within proximity of each other, edge-triangle intersection tests are also run to see if the triangles are overlapping. When it comes time to compute the forces in `TET_MESH::computeEdgeEdgeCollisionForces()`, the appropriate function is then selected.

14.7 The *Actual* Edge-Edge Energy Used in Fizt

14.7.1 A Fly in the Ointment

You may have already noticed something troubling about $\Psi_{\mathbf{e}\times\mathbf{e}}$. What happens if the two edges are really close to parallel, so close that their cross-product becomes undefined? What if $\mathbf{e}_0$ and $\mathbf{e}_1$ end up right on top of each other in Fig. 14.8? Uuuuuhhh well maybe it doesn't happen that much in practice? Or maybe we're just being obsessive, and in reality unless the two edges are *exactly parallel* to each other, everything works out just fine?

Unfortunately, that is not the world we live in. In fact, the cross product starts going haywire *way* before the two edges are exactly parallel. Other people have had to deal with this. Han et al. (2021) filters off edge-edge contacts that are less than 10 degrees apart. Harmon et al. (2008) ignore parallel edges[9], arguing that there's probably a vertex-face collision happening nearby that will handle things. There's a core irritation at the center of this all though: after all that work making sure we got the normal derivative right, *there's a geometric degeneracy at the center of it all?*

14.7.2 The Fizt Energy

We're going to use an energy here that is *very similar* to the vertex-face energy from §14.5, and *doesn't* contain this geometric degeneracy. Here it is:

$$\Psi_{\sqrt{\mathbf{ee}}} = \mu(\varepsilon - \|\mathbf{t}_{\mathbf{e}}\|)^2 \tag{14.39}$$

$$\mathbf{t}_{\mathbf{e}} = (\mathbf{x}_b - \mathbf{x}_a). \tag{14.40}$$

Instead of computing a normal anywhere, we're just going to push the two closest points apart (Fig. 14.10). You might have the same misgivings here as you had for the energy in §14.5. Don't we want a force that only pushes in some "normal" direction? Won't it introduce some stickiness in the tangent directions? Just like last time, it's fine to have these misgivings, but it practice we haven't found that the stickiness is all that visible.

Since there's no normal, *there's no cross product*. We're just going to push the two edges apart, along the direction connecting their two closest points. It's fine. It's *FINE.* Run the code if you don't believe me!

The gradient works out to:

$$\frac{\partial \Psi_{\sqrt{\mathbf{ee}}}}{\partial \mathbf{x}} = -2k\left(\varepsilon - \|\mathbf{t}_{\mathbf{e}}\|\right)\left(\frac{\partial \mathbf{t}_{\mathbf{e}}}{\partial \mathbf{x}}\right)^T \frac{\mathbf{t}_{\mathbf{e}}}{\|\mathbf{t}_{\mathbf{e}}\|}. \tag{14.41}$$

[9] They don't disclose the threshold at which they consider two edges "parallel". Booooo. That's right, I'm booing a paper. Booooo.

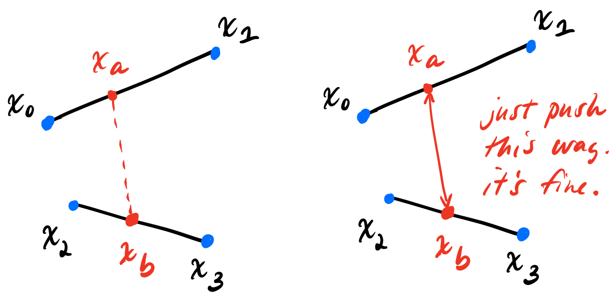


Figure 14.10.: For $\Psi_{\sqrt{\mathrm{ee}}}$, don't even compute a normal. The vector between $\mathbf{x}_a$ and $\mathbf{x}_b$? That's our force direction.

The $\frac{\partial \mathbf{t}_\mathbf{e}}{\partial \mathbf{x}}$ term is exactly the same as the $\frac{\partial \mathbf{x}_b - \mathbf{x}_a}{\partial \mathbf{x}}$ term in Eqn. 14.37. Nothing new here. In fact, there's even less to contend with than usual, since we don't have to deal with a $\frac{\partial \mathbf{n}}{\partial \mathbf{x}}$. Overall, it's actually *simpler.* The Hessian is then:

$$\frac{\partial^2 \Psi_{\sqrt{\mathrm{ee}}}}{\partial \mathbf{x}^2} = -2k(\varepsilon - \|\mathbf{t}_\mathbf{e}\|)\left(\frac{\partial \mathbf{t}_\mathbf{e}}{\partial \mathbf{x}}\right)^T \frac{\partial \mathbf{d}}{\partial \mathbf{x}} + 2k\left(\frac{\partial \|\mathbf{t}_\mathbf{e}\|}{\partial \mathbf{x}}\right)\left(\frac{\partial \mathbf{t}_\mathbf{e}}{\partial \mathbf{x}}\right)^T \mathbf{d}$$
$$\mathbf{d} = \frac{\mathbf{t}_\mathbf{e}}{\|\mathbf{t}_\mathbf{e}\|}.$$

The one new piece that just appeared is $\frac{\partial \mathbf{d}}{\partial \mathbf{x}}$, but that works out to:

$$\frac{\partial \mathbf{d}}{\partial \mathbf{x}} = \frac{1}{\|\mathbf{t}_\mathbf{e}\|}\frac{\partial \mathbf{t}_\mathbf{e}}{\partial \mathbf{x}} - \frac{1}{\|\mathbf{t}_\mathbf{e}\|^3}\mathbf{t}_\mathbf{e}\left(\frac{\partial \mathbf{t}_\mathbf{e}}{\partial \mathbf{x}}^T \mathbf{t}_\mathbf{e}\right)^T. \tag{14.42}$$

All done. The pain in these expressions is minimal, and anyway I implemented it all for you in `EDGE_SQRT_COLLISION`.

One last, *tiny* detail though. **This energy can become degenerate too.** See that $\mathbf{d} = \frac{\mathbf{t}_\mathbf{e}}{\|\mathbf{t}_\mathbf{e}\|}$ term? When happens when $\|\mathbf{t}_\mathbf{e}\| = \|\mathbf{x}_b - \mathbf{x}_a\| \approx 0$? Arrrrrggghhhh, does this endless labyrinth of numerical degeneracies ever end? The answer is no. We just give up if $\|\mathbf{d}\| \approx 10^{-8}$ and return zero force.[10]

[10]You might hold out the hope that *maybe this case doesn't happen that much in practice.* Unfortunately, it's not uncommon. If two edges are colliding with each other *and* a kinematic object, the kinematic constraints will smash the two edges together, making them exactly co-linear, and shooting $\|\mathbf{t}_\mathbf{e}\|$ straight to zero. As long as your object never engages in the simplest, most bargain-basement behavior of **colliding with a plane**, you're safe. You're not safe!

14.7.3 The Negated Version

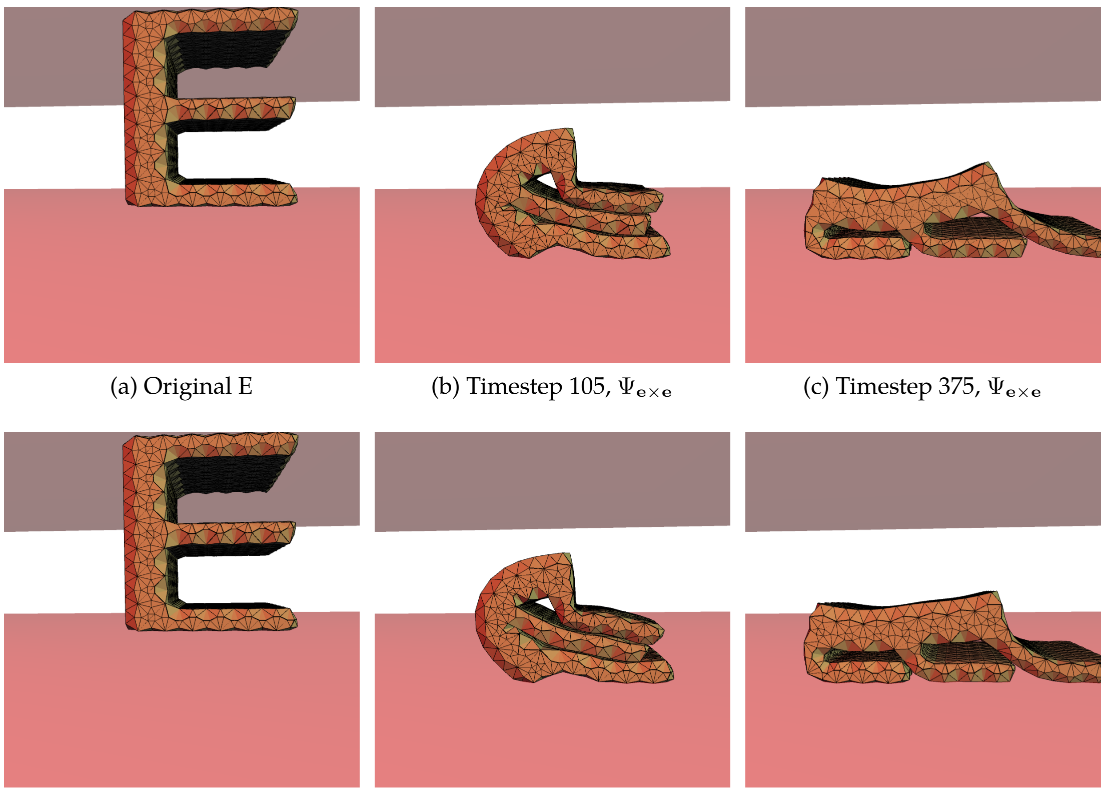
Just like $\Psi_{\mathbf{e}\times\mathbf{e}}$, you need to reverse the direction of $\Psi_{\sqrt{\mathbf{ee}}}$ if the two edges are already in collision. This works out to:

$$\Psi^{-}_{\sqrt{\mathbf{ee}}} = \mu(\varepsilon + \|\mathbf{t_e}\|)^2 \tag{14.43}$$

$$\frac{\partial \Psi^{-}_{\sqrt{\mathbf{ee}}}}{\partial \mathbf{x}} = 2k\left(\varepsilon + \|\mathbf{t_e}\|\right)\left(\frac{\partial \mathbf{t_e}}{\partial \mathbf{x}}\right)^T \frac{\mathbf{t_e}}{\|\mathbf{t_e}\|} \tag{14.44}$$

$$\frac{\partial^2 \Psi^{-}_{\sqrt{\mathbf{ee}}}}{\partial \mathbf{x}^2} = 2k(\varepsilon + \|\mathbf{t_e}\|)\left(\frac{\partial \mathbf{t_e}}{\partial \mathbf{x}}\right)^T \frac{\partial \mathbf{d}}{\partial \mathbf{x}} - 2k\left(\frac{\partial \|\mathbf{t_e}\|}{\partial \mathbf{x}}\right)\left(\frac{\partial \mathbf{t_e}}{\partial \mathbf{x}}\right)^T \mathbf{d}. \tag{14.45}$$

With all this in place, we can compare the two edge-edge collision energies side-by-side in Fig. 14.11.


(a) Original E (b) Timestep 105, $\Psi_{\mathbf{e}\times\mathbf{e}}$ (c) Timestep 375, $\Psi_{\mathbf{e}\times\mathbf{e}}$

(d) Original E (e) Timestep 105, $\Psi_{\sqrt{\mathbf{ee}}}$ (f) Timestep 375, $\Psi_{\sqrt{\mathbf{ee}}}$

Figure 14.11.: Dropping an E-shaped mesh on the ground, using the edge-edge collision energies $\Psi_{\mathbf{e}\times\mathbf{e}}$ and $\Psi_{\sqrt{\mathbf{ee}}}$. The energies are clearly not exactly the same, though they give the same overall behavior.

14.7.4 This Part Isn't in Fizt

You *can* combine the strengths of the previous two energies, and at least provide two lines of defense. Primarily, you could just use $\Psi_{\sqrt{\mathbf{ee}}}$, but if the two edges get too close together, fall back to $\Psi_{\mathbf{e}\times\mathbf{e}}$. I implemented this for you in `EDGE_HYBRID_COLLISION`.[11] What if the two edges are both parallel *and* overlapping? Then you're out of luck. I wasn't kidding. The labyrinth doesn't end.

This case actually *is* relatively rare though, so let's just call it good. A nearby vertex-face collision will probably save us from catastrophe. Probably. After this whole long journey, is this really where I'm going to leave things? Is this really the ending? It is. There's no natural law that says that the development of numerical methods needs to follow a satisfying plot arc. This is the ending. Really.

14.8 What About Eigenvalue Clamping?

One last piece that I still haven't covered yet, but will also end up feeling like an unresolved subplot: what about eigenvalue clamping? I gave you all the messy Hessians, where's the sweet closed-form eigendecomposition that sweeps away all the ugliness? We haven't found them yet. They're probably out there! But, we haven't found them yet.[12] Instead, you have two options.

First, you could take the brute-force numerical eigendecomposition of each Hessian. This isn't so bad when you're dealing with volumes, because the number of collisions is usually relatively small compared to the number of elements, so this isn't actually that expensive. This approach will probably become the bottleneck if you're dealing with cloth though.

Second, you could always do a *Gauss-Newton* approximation to the Hessian, where you only keep the outer product terms. Instead of,

$$\frac{\partial^2 \Psi_{\mathbf{e}\times\mathbf{e}}}{\partial \mathbf{x}^2} = 2k \left[\mathbf{g}_{\mathbf{e}\times\mathbf{e}} \mathbf{g}_{\mathbf{e}\times\mathbf{e}}^T + \left(\left((\varepsilon + \mathbf{t}_{\mathbf{e}})^T \mathbf{n} \right) \mathbf{t}_{\mathbf{e}}^T \frac{\partial^2 \mathbf{n}}{\partial \mathbf{x}^2} + \frac{\partial \mathbf{t}_{\mathbf{e}}}{\partial \mathbf{x}}^T \frac{\partial \mathbf{n}}{\partial \mathbf{x}} + \frac{\partial \mathbf{n}}{\partial \mathbf{x}}^T \frac{\partial \mathbf{t}_{\mathbf{e}}}{\partial \mathbf{x}} \right) \right]$$

just chop off everything that isn't of the form $\mathbf{x}\mathbf{x}^T$:

$$\frac{\partial^2 \Psi_{\mathbf{e}\times\mathbf{e}}}{\partial \mathbf{x}^2} \approx 2k \left[\mathbf{g}_{\mathbf{e}\times\mathbf{e}} \mathbf{g}_{\mathbf{e}\times\mathbf{e}}^T \right].$$

This is always guaranteed to be semi-positive definite as long the coefficient (in this case, $2k$) is greater than or equal to zero. This is what Fizt actually does.

[11] You could arrange the defenses in reverse order too: $\Psi_{\mathbf{e}\times\mathbf{e}}$ most of the time, but fall back to $\Psi_{\sqrt{\mathbf{ee}}}$ when the edges get too parallel.

[12] Once again, sounds like a good project for a graduate student.

Appendix A

Table of Symbols

We adhere as closely as possible to the following convention:

Symbol	Description
a	unbolded, lower-case letter is a *scalar*
A	unbolded, upper-case letter is (still) a *scalar*
$\mathbf{a}$	bold, lower-case letter is a *vector*
$\mathbf{A}$	bold, upper-case letter is a *matrix*
$\mathbf{a} \in \Re^n$	$\mathbf{a}$ is an n-dimensional vector
$\mathbf{A} \in \Re^{m \times n}$	$\mathbf{A}$ is an m by n matrix
$\mathbb{A} \in \Re^{m \times n \times o \times \dots}$	$\mathbb{A}$ is a higher-order (3rdand above) tensor

We use the following operators and special matrices:

Symbol	Description
$\|\cdot\|_2$	2-norm (§C.1.1)
$\|\cdot\|_F$	Frobenius norm (§C.1.2)
$\mathbf{A} : \mathbf{B}$	Double-contraction of matrices $\mathbf{A}$ and $\mathbf{B}$ (§C.3)
$\mathbb{A} : \mathbf{B}$	Double-contraction of higher-order tensor $\mathbb{A}$ and matrix $\mathbf{B}$ (§3.2,§C.3)
$\mathbf{A} \otimes \mathbf{B}$	A *Kronecker product* between two matrices (§C.5)
$\operatorname{tr} \mathbf{A}$	*trace* of matrix $\mathbf{A}$, i.e. the sum of its diagonal entries, $\sum_{i=0}^{n} a_{ii}$ (§C.2)
$\det \mathbf{A}$	*determinant* of matrix $\mathbf{A}$
$\operatorname{vec}(\cdot)$	*flattening* or *vectorization* of a matrix or tensor (§3)
$\mathbf{0}$	The zero matrix, a.k.a. the null matrix. Nothing but zeros
$\mathbf{I}$	The identity matrix. A diagonal matrix of all ones

These symbols are reserved for the following deformation-specific phenomena:

Symbol	Description
$\bar{\mathbf{x}}$	rest vertex, before deformation
$\mathbf{x}$	deformed vertex
$\mathbf{F} = \begin{bmatrix} f_{00} & f_{01} & f_{02} \\ f_{10} & f_{11} & f_{12} \\ f_{20} & f_{21} & f_{22} \end{bmatrix} = \left[\begin{array}{c\|c\|c} \mathbf{f}_0 & \mathbf{f}_1 & \mathbf{f}_2 \end{array} \right]$	deformation gradient, in 3D
$\mathbf{F} = \begin{bmatrix} f_{00} & f_{01} \\ f_{10} & f_{11} \end{bmatrix}$	deformation gradient, in 2D
$\mathbf{C}$	right Cauchy-Green tensor, $\mathbf{C} = \mathbf{F}^T\mathbf{F}$
$\mathbf{E}$	Green's strain, $\mathbf{E} = \mathbf{F}^T\mathbf{F} - \mathbf{I}$
$\mathbf{t}$	translation
$\mathbf{a}$	anisotropic fiber direction from §8
J	$J = \det \mathbf{F}$, the current relative volume of $\mathbf{F}$
$\mathbf{g}_1$	flattened gradient of I_1 invariant from §5.5
$\mathbf{g}_2$	flattened gradient of I_2 invariant from §5.5
$\mathbf{g}_J \equiv \mathbf{g}_3$	flattened gradient of $J \equiv I_3$ from §4.2.2.1
$\mathbf{H}_1$	flattened Hessian of I_1 invariant from §5.5
$\mathbf{H}_2$	flattened Hessian of I_2 invariant from §5.5
$\mathbf{H}_J \equiv \mathbf{H}_3$	flattened Hessian of $J \equiv I_3$ from §4.2.2.2
$\mathbf{R}$	rotation matrix from the polar decomposition $\mathbf{F} = \mathbf{RS}$
$\mathbf{S}$	scaling matrix from the polar decomposition $\mathbf{F} = \mathbf{RS}$
$\mathbf{U}$	left singular vectors of the SVD of $\mathbf{F} = \mathbf{U}\Sigma\mathbf{V}^T$
$\Sigma = \begin{bmatrix} \sigma_x & 0 & 0 \\ 0 & \sigma_y & 0 \\ 0 & 0 & \sigma_z \end{bmatrix}$	singular values of the SVD of $\mathbf{F} = \mathbf{U}\Sigma\mathbf{V}^T$
$\mathbf{V}$	right singular vectors of the SVD of $\mathbf{F} = \mathbf{U}\Sigma\mathbf{V}^T$
$\hat{\mathbf{x}} = \begin{bmatrix} 0 & -x_2 & x_1 \\ x_2 & 0 & -x_0 \\ -x_1 & x_0 & 0 \end{bmatrix}$	the cross product matrix of a vector $\mathbf{x}$
$\hat{\mathbf{n}}$	a normalized vector in §9 and §10

Appendix B

Useful Identities

Invariants:

$$I_{\mathbf{C}} = \|\mathbf{F}\|_F^2 \tag{B.1}$$

$$II_{\mathbf{C}} = \|\mathbf{F}^T\mathbf{F}\|_F^2 \tag{B.2}$$

$$III_{\mathbf{C}} = \det\left(\mathbf{F}^T\mathbf{F}\right) \tag{B.3}$$

$$IV_{\mathbf{C}} = \mathbf{a}^T\mathbf{C}\mathbf{a} \tag{B.4}$$

$$V_{\mathbf{C}} = \mathbf{a}^T\mathbf{C}^T\mathbf{C}\mathbf{a} \tag{B.5}$$

$$I_1 = \operatorname{tr}\left(\mathbf{R}^T\mathbf{F}\right) = \operatorname{tr}\mathbf{S} \tag{B.6}$$

$$I_2 = I_{\mathbf{C}} = \|\mathbf{F}\|_F^2 \tag{B.7}$$

$$I_3 = J = \det\mathbf{F} \tag{B.8}$$

$$I_4 = \mathbf{a}^T\mathbf{S}\mathbf{a} \tag{B.9}$$

$$I_5 = IV_{\mathbf{C}} = \mathbf{a}^T\mathbf{F}^T\mathbf{F}\mathbf{a} \tag{B.10}$$

Trace and Frobenius identities:

$$\operatorname{tr}(\alpha\mathbf{A}) = \alpha\operatorname{tr}\mathbf{A} \tag{B.11}$$

$$\operatorname{tr}(\mathbf{A}+\mathbf{B}) = \operatorname{tr}\mathbf{A} + \operatorname{tr}\mathbf{B} \tag{B.12}$$

$$\operatorname{tr}(\mathbf{A}-\mathbf{B}) = \operatorname{tr}\mathbf{A} - \operatorname{tr}\mathbf{B} \tag{B.13}$$

$$\operatorname{tr}\left(\mathbf{A}^T\mathbf{B}\right) = \mathbf{A}:\mathbf{B} = \operatorname{vec}(\mathbf{A})^T\operatorname{vec}(\mathbf{B}) \tag{B.14}$$

$$\|\mathbf{A}+\mathbf{B}\|_F^2 = \|\mathbf{A}\|_F^2 + \|\mathbf{B}\|_F^2 + 2\operatorname{tr}\mathbf{A}^T\mathbf{B} \tag{B.15}$$

$$\|\mathbf{A}-\mathbf{B}\|_F^2 = \|\mathbf{A}\|_F^2 + \|\mathbf{B}\|_F^2 - 2\operatorname{tr}\mathbf{A}^T\mathbf{B} \tag{B.16}$$

$$\operatorname{tr}\left(\mathbf{B}\mathbf{A}\mathbf{B}^{-1}\right) = \operatorname{tr}\left(\mathbf{A}\mathbf{B}^{-1}\mathbf{B}\right) = \operatorname{tr}(\mathbf{A}) \tag{B.17}$$

$$\operatorname{tr}\left(\mathbf{R}\mathbf{A}\mathbf{R}^T\right) = \operatorname{tr}(\mathbf{A}) \quad \text{(when } \mathbf{R} \text{ is a rotation)} \tag{B.18}$$

Invariant identities and derivatives:

$$\frac{\partial I_1}{\partial \mathbf{F}} = \frac{\partial \operatorname{tr} \mathbf{S}}{\partial \mathbf{F}} = \mathbf{R} \tag{B.19}$$

$$\frac{\partial I_2}{\partial \mathbf{F}} = \frac{\partial I_{\mathbf{C}}}{\partial \mathbf{F}} = \frac{\partial \|\mathbf{F}\|_F^2}{\partial \mathbf{F}} = 2\mathbf{F} \tag{B.20}$$

$$\frac{\partial II_{\mathbf{C}}}{\partial \mathbf{F}} = \frac{\partial \|\mathbf{F}^T\mathbf{F}\|_F^2}{\partial \mathbf{F}} = 4\mathbf{F}\mathbf{F}^T\mathbf{F} \tag{B.21}$$

$$\frac{\partial I_3}{\partial \mathbf{F}} = \frac{\partial J}{\partial \mathbf{F}} = \begin{bmatrix} f_{11} & -f_{10} \\ -f_{01} & f_{00} \end{bmatrix} \quad \text{(In 2D)} \tag{B.22}$$

$$\frac{\partial I_3}{\partial \mathbf{F}} = \frac{\partial J}{\partial \mathbf{F}} = \left[\begin{array}{c|c|c} \mathbf{f}_1 \times \mathbf{f}_2 & \mathbf{f}_2 \times \mathbf{f}_0 & \mathbf{f}_0 \times \mathbf{f}_1 \end{array}\right] \quad \text{(In 3D)} \tag{B.23}$$

$$\frac{\partial I_5}{\partial \mathbf{F}} = \frac{\partial IV_{\mathbf{C}}}{\partial \mathbf{F}} = 2\mathbf{F}\mathbf{a}\mathbf{a}^T \tag{B.24}$$

$$\operatorname{vec}\left(\frac{\partial^2 I_2}{\partial \mathbf{F}^2}\right) = \mathbf{H}_2 = \mathbf{H}_I = 2\mathbf{I}_{9\times 9} \tag{B.25}$$

$$\operatorname{vec}\left(\frac{\partial^2 I_3}{\partial \mathbf{F}^2}\right) = \mathbf{H}_J = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & -1 & 0 \\ 0 & -1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \quad \text{(In 2D)} \tag{B.26}$$

$$\operatorname{vec}\left(\frac{\partial^2 I_3}{\partial \mathbf{F}^2}\right) = \mathbf{H}_J = \begin{bmatrix} \mathbf{0}_{3\times 3} & -\hat{\mathbf{f}}_2 & \hat{\mathbf{f}}_1 \\ \hat{\mathbf{f}}_2 & \mathbf{0}_{3\times 3} & -\hat{\mathbf{f}}_0 \\ -\hat{\mathbf{f}}_1 & \hat{\mathbf{f}}_0 & \mathbf{0}_{3\times 3} \end{bmatrix} \quad \text{(In 3D)} \tag{B.27}$$

$$\operatorname{vec}\left(\frac{\partial^2 I_5}{\partial \mathbf{F}^2}\right) = \operatorname{vec}\left(\frac{\partial^2 IV_{\mathbf{C}}}{\partial \mathbf{F}^2}\right) = 2\mathbf{a}\mathbf{a}^T \otimes \mathbf{I} \tag{B.28}$$

$$\operatorname{vec}\left(\frac{\partial^2 II_{\mathbf{C}}}{\partial \mathbf{F}^2}\right) = \mathbf{H}_{II} = 4\left(\mathbf{I}_{3\times 3} \otimes \mathbf{F}\mathbf{F}^T + \mathbf{F}^T\mathbf{F} \otimes \mathbf{I}_{3\times 3} + \mathbf{D}\right) \tag{B.29}$$

$$\mathbf{D} = \begin{bmatrix} \mathbf{f}_0\mathbf{f}_0^T & \mathbf{f}_1\mathbf{f}_0^T & \mathbf{f}_2\mathbf{f}_0^T \\ \mathbf{f}_0\mathbf{f}_1^T & \mathbf{f}_1\mathbf{f}_1^T & \mathbf{f}_2\mathbf{f}_1^T \\ \mathbf{f}_0\mathbf{f}_2^T & \mathbf{f}_1\mathbf{f}_2^T & \mathbf{f}_2\mathbf{f}_2^T \end{bmatrix} \tag{B.30}$$

$$\frac{\partial \log(I_3)}{\partial \mathbf{F}} = \frac{1}{I_3}\frac{\partial I_3}{\partial \mathbf{F}} = \frac{1}{J}\frac{\partial J}{\partial \mathbf{F}} \tag{B.31}$$

$$\operatorname{tr}\left(\mathbf{F}^T\mathbf{F}\right) = \|\mathbf{F}\|_F^2 = \mathbf{F} : \mathbf{F} = \sum_i \sum_j \mathbf{F}_{ij}^2 \tag{B.32}$$

$$\mathbf{F}^T\mathbf{R} = \mathbf{S} \tag{B.33}$$

$$\frac{\partial \|\mathbf{R}\|_F^2}{\partial \mathbf{F}} = \frac{\partial \operatorname{tr}\left(\mathbf{R}^T\mathbf{R}\right)}{\partial \mathbf{F}} = \frac{\partial \operatorname{tr}\mathbf{I}}{\partial \mathbf{F}} = 0 \tag{B.34}$$

$$\|\mathbf{F}^{-1}\|_F^2 = \frac{I_2}{(I_3)^2} \quad \text{(In 2D)} \tag{B.35}$$

$$\|\mathbf{F}^{-1}\|_F^2 = \frac{1}{4}\left(\frac{I_1^2 - I_2}{I_3}\right) - 2\frac{I_1}{I_3} \quad \text{(In 3D)} \tag{B.36}$$

Appendix C

Notation

C.1 Norms

A *norm* is a way to boil the entries of a vector or matrix down to a single scalar, or score. Often it's impossible to stare at the wall of numbers that constitute a vector or matrix and understand what's going on, so it's useful to somehow distill the "essence" of the vector or matrix into a single number. Often you are looking for different "essences" though, so to capture each of these, there are a variety of different norms.[1]

C.1.1 The 2-Norm

The vector norm we use most often is the *2-norm*. For some vector $\mathbf{x}$,

$$\mathbf{x} = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ x_{n-1} \end{bmatrix} \tag{C.1}$$

the 2-norm $\|\cdot\|_2$ is the square root of the squared sum:

$$\|\mathbf{x}\|_2 = \sqrt{\sum_{i=0}^{n-1} x_i^2}. \tag{C.2}$$

If there are only two entries in $\mathbf{x}$, such as $\mathbf{x} = \begin{bmatrix} x_0 \\ x_1 \end{bmatrix}$, then we get the familiar Euclidean distance:

$$\|\mathbf{x}\|_2 = \sqrt{x_0^2 + x_1^2}. \tag{C.3}$$

[1]Another way to think of a norm is "average over all the members". So when we talk of *societal norms*, it is the overall behavior that we expect from averaging over the members of that society.

The 2-norm is just a generalization of this style of measurement.

C.1.2 The Frobenius Norm

The matrix norm we will use the most often is the *Frobenius norm*. It is exactly the same as the 2-norm, except that it is defined over all the entries of a matrix. For some matrix $\mathbf{A}$,

$$\mathbf{A} = \begin{bmatrix} a_{00} & a_{01} & \cdots & a_{0(n-1)} \\ a_{10} & a_{11} & \cdots & a_{1(n-1)} \\ \vdots & \vdots & \ddots & \vdots \\ a_{(m-1)0} & a_{(m-1)1} & \cdots & a_{(m-1)(n-1)} \end{bmatrix} \tag{C.4}$$

the Frobenius norm $\|\cdot\|_F$ is (again) the square root of the squared sum:

$$\|\mathbf{A}\|_F = \sqrt{\sum_{i=0}^{m-1} \sum_{j=0}^{n-1} a_{ij}^2}. \tag{C.5}$$

It is ugly to have the square root enveloping everything, and it also introduces pesky $1/2$ factors when you take derivatives, so usually we use the *squared* Frobenius norm:

$$\|\mathbf{A}\|_F^2 = \sum_{i=0}^{m-1} \sum_{j=0}^{n-1} a_{ij}^2. \tag{C.6}$$

Notationally, this version only differs from $\|\cdot\|_F$ with the 2 in the super-script, $\|\cdot\|_F^2$, so it can be easy to get confused. Just remember that we're almost always going to use the squared version. In fact, as I write this, I'm having trouble thinking of a single instance where we will need to use the un-squared $\|\cdot\|_F$.

I'll wrap up discussion of this norm with two observations.

- The 2-norm is literally a vector version of the Frobenius norm. You can write $\|\mathbf{x}\|_F$ instead of $\|\mathbf{x}\|_2$ and they mean the *exact same thing*.
- The Frobenius norm is *weird*. With the 2-norm, we have some notion of measuring the magnitude of a vector. With the Frobenius norm, we appear to be arbitrarily smashing all the entries of a matrix together into one scalar.

  Doesn't anybody care which row and column these entries came from? Surely their ordering in the matrix should count for something? Maybe there is a matrix norm out there that cares, but not Frobenius. Despite this apparent blindness, the Frobenius norm gets the job done in many useful scenarios.

C.2 Matrix Trace

The trace of a matrix, $\operatorname{tr} \mathbf{A}$, is the sum of its diagonal entries:

$$\operatorname{tr} \mathbf{A} = \sum_{i=0}^{n-1} a_{ii}. \tag{C.7}$$

This seems like an arbitrary operation to define for a matrix. So if I swap two of the rows, then I get a different trace? How is this at all useful? While this is indeed odd, the interesting and surprising property of the matrix trace is that $\operatorname{tr} \mathbf{A}$ is the *sum of the eigenvalues of* $\mathbf{A}$.

This is not obvious at all! The eigenvalues are supposed to be some fundamental, atomic property that can only be acquired after a perilous journey involving Householder reductions, shifted-QR factorizations, or other exotic numerical analysis weapons. And yet, the *sum* of these buried jewels can somehow be plucked right off of the surface of the matrix. Huh.

C.3 Matrix Double-Contractions

The double-contraction, denotes with a ':', is a generalized version of the dot product for vectors. With vectors, we multiply together the corresponding entries in vector, and then glob them all together in a summation:

$$\mathbf{x}^T\mathbf{y} = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix}^T \begin{bmatrix} y_0 \\ y_1 \\ y_2 \end{bmatrix} = x_0y_0 + x_1y_1 + x_2y_2. \tag{C.8}$$

The double-contraction does the *exact same thing*, but for the entries of two matrices:

$$\mathbf{A} : \mathbf{B} = \begin{bmatrix} a_0 & a_2 \\ a_1 & a_3 \end{bmatrix} \begin{bmatrix} b_0 & b_2 \\ b_1 & b_3 \end{bmatrix} = a_0b_0 + a_1b_1 + a_2b_2 + a_3b_3. \tag{C.9}$$

The dimensions of the two matrices need to be the same, otherwise the operations is undefined (and you screwed up somewhere).

This operation may look odd to you. Similar to the Frobenius norm, it chucks all the entries of the matrices into the same pile, insensitive to whether they have important-looking VIP status like being along the diagonal. It is indeed odd, but this is the definition nevertheless, and despite its status-obliviousness, it is still quite useful.

C.4 Outer Products

You've probably been sent here by §4.2.2.1. We're used to seeing inner products, a.k.a. dot products like $\mathbf{x}^T\mathbf{y} = \alpha$ where two vectors $\mathbf{x}$ and $\mathbf{y}$ are smooshed together in such a way that they're demoted to the lowly scalar α. In other words, inner products take two rank-one quantities (vectors) and bash them down to rank-zero quantity (a scalar).

The outer product goes in the other direction: $\mathbf{x}\mathbf{x}^T = \mathbf{A}$. A pair of rank-one quantities join forces to generate a bigger, rank-two result (a matrix). This product doesn't appear quite as often as the inner product, so while it probably did show up in your linear algebra class at some point, there's a good chance that you haven't seen it since you crammed for the midterm. So, here's a refresher.

If we have two vectors,

$$\mathbf{x} = \begin{bmatrix} a \\ b \\ c \end{bmatrix} \qquad \mathbf{y} = \begin{bmatrix} d \\ e \\ f \end{bmatrix}, \tag{C.10}$$

then their outer product is

$$\mathbf{x}\mathbf{y}^T = \begin{bmatrix} a \\ b \\ c \end{bmatrix} \begin{bmatrix} d \\ e \\ f \end{bmatrix}^T = \begin{bmatrix} ad & ae & af \\ bd & be & bf \\ cd & ce & cf \end{bmatrix} \tag{C.11}$$

We place a copy of $\mathbf{x}$ into each column, and in turn, scale the copy by each entry in $\mathbf{y}$.

Why is this helpful? Inner products have intuitive interpretations like projecting of one vector onto another, but what's the point of this baroque and arbitrary-looking operation?

The outer product form often shows up when you see what's *really* going on inside the primordial morass of numbers that form a matrix. We can look at the eigendecomposition of a symmetric $n \times n$ matrix,

$$\mathbf{A} = \mathbf{Q}\Lambda\mathbf{Q}^T, \tag{C.12}$$

where $\mathbf{Q}$ is the matrix of eigenvectors, $\mathbf{q}_i$, and Λ is a matrix of the eigenvalues, λ_i:

$$\mathbf{Q} = \left[\begin{array}{c|c|c|c} \mathbf{q}_0 & \mathbf{q}_1 & \dots & \mathbf{q}_{n-1} \end{array}\right] \qquad \Lambda = \begin{bmatrix} \lambda_0 & & & \\ & \lambda_1 & & \\ & & \ddots & \\ & & & \lambda_{n-1} \end{bmatrix}. \tag{C.13}$$

We can see a hint of the $\mathbf{x}\mathbf{y}^T$ outer product form by looking at $\mathbf{Q}\Lambda\mathbf{Q}^T$. Both have a transpose all the way on the right hand side. This is not a coincidence, as the eigendecomposition of a matrix can indeed be written as a sum of outer products:

$$\mathbf{A} = \mathbf{Q}\Lambda\mathbf{Q}^T = \lambda_0\mathbf{q}_0\mathbf{q}_0^T + \lambda_1\mathbf{q}_1\mathbf{q}_1^T + \ldots + \lambda_n\mathbf{q}_n\mathbf{q}_n^T \tag{C.14}$$

$$\mathbf{A} = \sum_{i=0}^{n-1} \lambda_i\mathbf{q}_i\mathbf{q}_i^T. \tag{C.15}$$

Thus, one way to look at a matrix $\mathbf{A} \in \Re^{n \times n}$ is as a *scaled sum* of n outer products. What does this mean? Say you went ahead and just lopped off the last ten terms in the sum:

$$\mathbf{A}_{\text{oops}} = \sum_{i=0}^{n-11} \lambda_i \mathbf{q}_i \mathbf{q}_i^T. \tag{C.16}$$

You would still get an $n \times n$ matrix out in the end. However, lurking inside that matrix would be rank-deficiency. You secretly don't need the full complement of n outer products to produce all the entries of $\mathbf{A}_{\text{oops}}$.

In the extreme, you might only need *one* outer product to produce $\mathbf{A}_{\text{oops}}$, making it a rank-one matrix. In that case, you don't need to store the whole matrix $\mathbf{A}_{\text{oops}}$, you can just carry around the one eigenpair, $(\mathbf{q}_0, \lambda_0)$, and generate $\mathbf{A}_{\text{oops}} = \lambda_0 \mathbf{q}_0 \mathbf{q}_0^T$ on-the-fly whenever you need it.

Conversely, imagine you're looking at the eigendecompositions of matrices that appear somewhere in your code. The decomposition keeps telling you that it's a rank-one matrix. In this case, your job as a numerical archeologist just got easier. Instead of trying to excavate the underlying structure of all $n \times n$ entries, you can now just focus on understanding where this one $\mathbf{q}_0 \in \Re^n$ came from. Your task just got quadratically easier.

C.5 Kronecker Products

The Kronecker product is the matrix version of the outer product, though it irritatingly reverses the ordering of the operators. The Kronecker product of $\mathbf{A}$ and $\mathbf{B}$ is:

$$\begin{aligned}\mathbf{A} \otimes \mathbf{B} &= \begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix} \otimes \begin{bmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{bmatrix} = \begin{bmatrix} a_{00}\mathbf{B} & a_{01}\mathbf{B} \\ a_{10}\mathbf{B} & a_{11}\mathbf{B} \end{bmatrix} \\ &= \begin{bmatrix} a_{00}b_{00} & a_{00}b_{01} & a_{01}b_{00} & a_{01}b_{01} \\ a_{00}b_{10} & a_{00}b_{11} & a_{01}b_{00} & a_{01}b_{01} \\ a_{10}b_{00} & a_{10}b_{01} & a_{11}b_{00} & a_{11}b_{01} \\ a_{10}b_{10} & a_{10}b_{11} & a_{11}b_{00} & a_{11}b_{01} \end{bmatrix}\end{aligned}$$

Whereas the outer product stacked together a bunch of scaled copies of $\mathbf{x}$ from the left, the Kronecker product stacks scaled copies of $\mathbf{B}$ from the right. I don't make the rules here. Already I'm wondering where in these notes I messed up the ordering.

This product commonly appears with an identity matrix. If you have a matrix $\mathbf{A}$, and you need to stamp copies of $\mathbf{A}$ down the block-diagonal of a matrix, then the Kronecker product is up to the task:

$$\begin{aligned}
\mathbf{I}_{2\times 2} \otimes \mathbf{A} &= \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \otimes \begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix} = \begin{bmatrix} \mathbf{A} & \mathbf{0}_{2\times 2} \\ \mathbf{0}_{2\times 2} & \mathbf{A} \end{bmatrix} \\
&= \begin{bmatrix} a_{00} & a_{01} & 0 & 0 \\ a_{10} & a_{11} & 0 & 0 \\ 0 & 0 & a_{00} & a_{01} \\ 0 & 0 & a_{10} & a_{11} \end{bmatrix}
\end{aligned}$$

If you need to spread each entry of $\mathbf{A}$ across the block-diagonal of a matrix, reversing the two matrices gets the job done:

$$\begin{aligned}
\mathbf{A} \otimes \mathbf{I}_{2\times 2} &= \begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix} \otimes \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} a_{00}\mathbf{I}_{2\times 2} & a_{01}\mathbf{I}_{2\times 2} \\ a_{10}\mathbf{I}_{2\times 2} & a_{11}\mathbf{I}_{2\times 2} \end{bmatrix} \\
&= \begin{bmatrix} a_{00} & 0 & a_{01} & 0 \\ 0 & a_{00} & 0 & a_{01} \\ a_{10} & 0 & a_{11} & 0 \\ 0 & a_{10} & 0 & a_{11} \end{bmatrix}.
\end{aligned}$$

These sorts of patterns tend to appear when you flatten out higher order tensors. So, we see them in §4.2.3 when computing $\mathbf{H}_{II}$ and in §8.2.2.1 when computing $\mathbf{H}_{IV} \equiv \mathbf{H}_5$.

Appendix D

How to Compute F*, the Deformation Gradient*

First we'll look at an intuitive, geometric, purely linear-algebraic way to compute $\mathbf{F}$ on triangles and tetrahedra. Then we'll look at a slightly more obtuse, calculus-based method that does the exact same thing using finite element basis functions. While this method is more involved and less intuitive, it will point the way to computing $\mathbf{F}$ on other primitives, such as hexahedra.

D.1 Computing F for 2D Triangles

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

We specifically 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. D.1). The new vertices of our origin-centered rest-triangle become:

$$\bar{\mathbf{x}}_0 \rightarrow \qquad \bar{\mathbf{o}}_0 = [0 \quad 0]^T \tag{D.1}$$

$$\bar{\mathbf{x}}_1 \rightarrow \qquad \bar{\mathbf{o}}_1 = \bar{\mathbf{x}}_1 - \bar{\mathbf{x}}_0 \tag{D.2}$$

$$\bar{\mathbf{x}}_2 \rightarrow \qquad \bar{\mathbf{o}}_2 = \bar{\mathbf{x}}_2 - \bar{\mathbf{x}}_0. \tag{D.3}$$

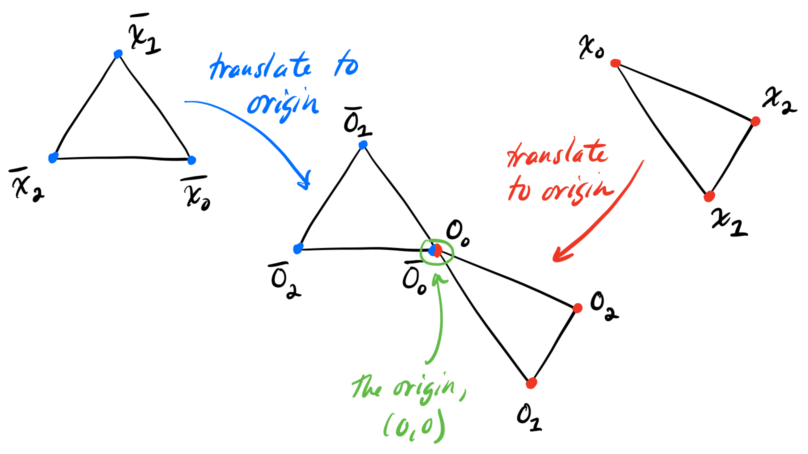


Figure D.1.: Let's eliminate the translation by pulling the rest triangle (left, blue vertices), back to the origin (center left, blue vertices). We do the same thing to the deformed triangle (right, red vertices).

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

$$\mathbf{x}_0 \rightarrow \qquad \mathbf{o}_0 = [0 \quad 0]^T \tag{D.4}$$
$$\mathbf{x}_1 \rightarrow \qquad \mathbf{o}_1 = \mathbf{x}_1 - \mathbf{x}_0 \tag{D.5}$$
$$\mathbf{x}_2 \rightarrow \qquad \mathbf{o}_2 = \mathbf{x}_2 - \mathbf{x}_0. \tag{D.6}$$

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 $\mathbf{o}_i$ vertices. In other words, we want the matrix $\mathbf{F}$ satisfies these three equations:

$$\mathbf{F}\bar{\mathbf{o}}_0 = \mathbf{o}_0 \qquad \mathbf{F}\bar{\mathbf{o}}_1 = \mathbf{o}_1 \qquad \mathbf{F}\bar{\mathbf{o}}_2 = \mathbf{o}_2. \tag{D.7}$$

The first equation, $\mathbf{F}\bar{\mathbf{o}}_0 = \mathbf{o}_0$ 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 $\mathbf{o}_1$ and $\mathbf{o}_2$ when provided with $\bar{\mathbf{o}}_1$ and $\bar{\mathbf{o}}_2$. If we write this desire down explicitly, it's a well-posed linear algebra problem:

$$\mathbf{F}\left[\begin{array}{c|c} \bar{\mathbf{x}}_1 - \bar{\mathbf{x}}_0 & \bar{\mathbf{x}}_2 - \bar{\mathbf{x}}_0 \end{array}\right] = \left[\begin{array}{c|c} \mathbf{x}_1 - \mathbf{x}_0 & \mathbf{x}_2 - \mathbf{x}_0 \end{array}\right] \tag{D.8}$$

$$\mathbf{F}\left[\begin{array}{c|c} \bar{\mathbf{o}}_1 & \bar{\mathbf{o}}_2 \end{array}\right] = \left[\begin{array}{c|c} \mathbf{o}_1 & \mathbf{o}_2 \end{array}\right] \tag{D.9}$$

$$\mathbf{F}\mathbf{D}_m = \mathbf{D}_s \tag{D.10}$$

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

$$\mathbf{F} = \mathbf{D}_s \mathbf{D}_m^{-1}. \tag{D.11}$$

The $\mathbf{D}_m$ matrix is quite small, i.e. 2×2 in 2D, so there's no need to be worried that taking its inverse might be expensive. The $\mathbf{D}_m$ matrix also *only* depends on the rest pose, so you can precompute its inverse once at startup, and then when you need to compute $\mathbf{F}$ at any later time, no new inverse computation is needed.

Here we follow the naming convention in Teran et al. (2005) and abbreviate things to $\mathbf{D}_m$ and $\mathbf{D}_s$, where m denotes *material* coordinates and s denotes *spatial* coordinates[1]. In a perfect world we'd stick with our own naming convention and call these $\mathbf{D}_r$ and $\mathbf{D}_d$, for *rest* and *deformed*, but the m and s subscripts are popular enough in other papers that we're going to stick to them here. Then you won't get confused when you read these other papers (including some of mine).

One thing you may be wondering: we specifically decided to pin $\bar{\mathbf{x}}_0$ to the origin. Could we pick $\bar{\mathbf{x}}_1$ or $\bar{\mathbf{x}}_2$? Does it matter? Will we get the same $\mathbf{F}$? Is one better than the other? It doesn't actually matter, though you should work out a few examples to convince yourself of this fact.

One thing you might worry about is the *conditioning* of these elements, i.e. instead of a nice chubby triangle you have a sadly skinny one that is essentially a line. A poorly-conditioned element is known to ruin the stability of simulations, because divide-by-zeros occur when computing $\mathbf{F}$. This happens because $\mathbf{o}_1$ and $\mathbf{o}_2$ are nearly coincident and $\mathbf{D}_m^{-1}$ becomes rank-deficient. Picking a different $\bar{\mathbf{x}}_i$ to the origin will not to save your bacon.

D.2 Computing F 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:

$$\mathbf{F} \left[\begin{array}{c|c|c} \bar{\mathbf{x}}_1 - \bar{\mathbf{x}}_0 & \bar{\mathbf{x}}_2 - \bar{\mathbf{x}}_0 & \bar{\mathbf{x}}_3 - \bar{\mathbf{x}}_0 \end{array} \right] = \left[\begin{array}{c|c|c} \mathbf{x}_1 - \mathbf{x}_0 & \mathbf{x}_2 - \mathbf{x}_0 & \mathbf{x}_3 - \mathbf{x}_0 \end{array} \right] \tag{D.12}$$

$$\mathbf{F} \left[\begin{array}{c|c|c} \bar{\mathbf{o}}_1 & \bar{\mathbf{o}}_2 & \bar{\mathbf{o}}_3 \end{array} \right] = \left[\begin{array}{c|c|c} \mathbf{o}_1 & \mathbf{o}_2 & \mathbf{o}_3 \end{array} \right] \tag{D.13}$$

$$\mathbf{F}\mathbf{D}_m = \mathbf{D}_s \tag{D.14}$$

$$\mathbf{F} = \mathbf{D}_s \mathbf{D}_m^{-1} \tag{D.15}$$

[1] I think. I'm guessing here, but it seems quite likely.

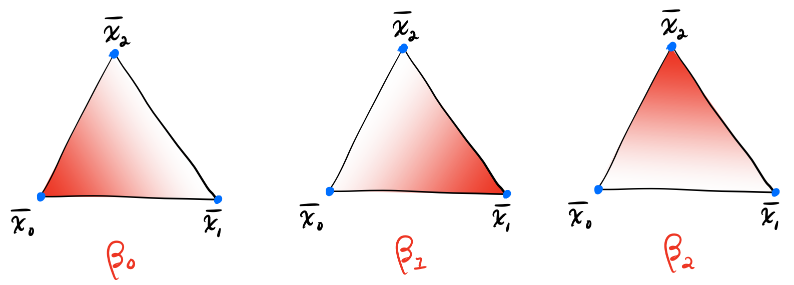


Figure D.2.: The basis functions are linear functions that are equal to one (red) at their corresponding vertices and ramp linearly down to zero (white) as the opposing edge is approached.

D.3 Computing F the Finite Element Way

Now let's do the exact same thing, but in a different and more opaque way. But why? In the end, there will be an obvious way to extend *this* approach to computing **F** on squares (quads) and cubes (hexahedra).

D.3.1 Remember Barycentric Coordinates?

We are going to use the Graphics 101 (see e.g. Marschner and Shirley (2016)) concept of *barycentric* coordinates for this one. If you've done some finite element stuff, you will recognize these as *basis or shape functions* over the triangle, specifically *hat functions*. If you're reading this though, you're probably a graphics person and not a mechanical engineering person, so we'll stick to barycentric coordinates. For our purposes, the two concepts are completely equivalent.

A barycentric coordinate system assigns an easy-to-use 2D coordinate to every point on the interior of a triangle. These are usually denoted u and v, that are easy-to-use because they are always defined over $[0, 1]$. If we have some barycentric coordinate $\mathbf{b} = [u \;\; v]^T$, we can then reconstruct any point $\mathbf{x}(\mathbf{b})$ or $\bar{\mathbf{x}}(\mathbf{b})$ on the interior of the triangle using a linear combination of the basis functions β_i:

$$\mathbf{x}(\mathbf{b}) = \sum_{i=0}^{2} \mathbf{x}_i \beta_i(\mathbf{b}) \qquad \bar{\mathbf{x}}(\mathbf{b}) = \sum_{i=0}^{2} \bar{\mathbf{x}}_i \beta_i(\mathbf{b}) \tag{D.16}$$

The basis functions β_i are:

$$\beta_0(\mathbf{b}) = 1 - u - v \qquad \beta_1(\mathbf{b}) = u \qquad \beta_2(\mathbf{b}) = v \tag{D.17}$$

Each basis function β_i is attached to its corresponding vertex $\mathbf{x}_i$ (see Fig. D.2). It is equal to 1 right on top of the vertex, and then fades off to 0 as you approach the triangle edge that is opposite to that vertex.

D.3.2 Computing F for 2D Triangles, the Basis Function Way

Now let's use these basis functions to compute the deformation gradient. Remember way back in Eqn. 2.2 that the deformed points were defined as

$$\mathbf{x} = \mathbf{F}\bar{\mathbf{x}} + \mathbf{t}, \tag{D.18}$$

so we can retrieve the deformation gradient by taking $\frac{\partial \mathbf{x}}{\partial \bar{\mathbf{x}}}$ (we actually wrote it as $\frac{\partial \phi}{\partial \bar{\mathbf{x}}}$, but it's the same). We can now use the basis functions to compute this in a brute-force, no-intuition-to-cling-to, calculus-based way.

Remember from Eqn. D.16 that we can also write both $\mathbf{x}$ and $\bar{\mathbf{x}}$ in terms of the barycentric coordinate $\mathbf{b}$. We can thus expand $\frac{\partial \mathbf{x}}{\partial \bar{\mathbf{x}}}$ using the chain rule to route it through the $\mathbf{b}$-dependent version of $\mathbf{x}$ and $\bar{\mathbf{x}}$:

$$\begin{aligned}\mathbf{F} = \frac{\partial \mathbf{x}}{\partial \bar{\mathbf{x}}} &= \frac{\partial \mathbf{x}}{\partial \mathbf{b}}\frac{\partial \mathbf{b}}{\partial \bar{\mathbf{x}}} \\ &= \frac{\partial \mathbf{x}}{\partial \mathbf{b}}\left(\frac{\partial \bar{\mathbf{x}}}{\partial \mathbf{b}}\right)^{-1}.\end{aligned} \tag{D.19}$$

If we then plug in our basis-function-based definitions of $\mathbf{x}$ and $\bar{\mathbf{x}}$ (Eqn. D.16), we get:

$$\begin{aligned}\mathbf{F} &= \frac{\partial \mathbf{x}}{\partial \mathbf{b}}\left(\frac{\partial \bar{\mathbf{x}}}{\partial \mathbf{b}}\right)^{-1} \\ &= \frac{\partial}{\partial \mathbf{b}}\left(\sum_{i=0}^{2}\mathbf{x}_i\beta_i(\mathbf{b})\right)\frac{\partial}{\partial \mathbf{b}}\left(\sum_{i=0}^{2}\bar{\mathbf{x}}_i\beta_i(\mathbf{b})\right)^{-1} \\ &= \left(\sum_{i=0}^{2}\mathbf{x}_i\frac{\partial\beta_i(\mathbf{b})}{\partial \mathbf{b}}\right)\left(\sum_{i=0}^{2}\bar{\mathbf{x}}_i\frac{\partial\beta_i(\mathbf{b})}{\partial \mathbf{b}}\right)^{-1}.\end{aligned} \tag{D.20}$$

Since $\mathbf{x}_i$ and $\bar{\mathbf{x}}_i$ don't actually depend on the value of $\mathbf{b}$, we can to write the derivative purely in terms of β_i. This lack of dependence make sense; we're generating $\mathbf{x}(\mathbf{b})$ by interpolating over a set of $\mathbf{x}_i$. No matter what $\mathbf{b}$ we send down the pipe, the set of $\mathbf{x}_i$ points we're trying to interpolate over stays the same.

The next trick is that we can write down Eqn. D.20 in matrix form

$$\begin{aligned}\mathbf{F} &= \left(\sum_{i=0}^{2}\mathbf{x}_i\frac{\partial\beta_i(\mathbf{b})}{\partial \mathbf{b}}\right)\left(\sum_{i=0}^{2}\bar{\mathbf{x}}_i\frac{\partial\beta_i(\mathbf{b})}{\partial \mathbf{b}}\right)^{-1} \\ &= \mathbf{X}\mathbf{H}\left(\bar{\mathbf{X}}\mathbf{H}\right)^{-1}.\end{aligned} \tag{D.21}$$

The matrices $\mathbf{X}$ and $\bar{\mathbf{X}}$ are just the deformed and rest vertices of the triangles packed together:

$$\bar{\mathbf{X}} = \left[\begin{array}{c|c|c} \\ \bar{\mathbf{x}}_0 & \bar{\mathbf{x}}_1 & \bar{\mathbf{x}}_2 \\ \\ \end{array}\right] \in \Re^{2\times 3} \tag{D.22}$$

$$\mathbf{X} = \left[\begin{array}{c|c|c} \\ \mathbf{x}_0 & \mathbf{x}_1 & \mathbf{x}_2 \\ \\ \end{array}\right] \in \Re^{2\times 3}. \tag{D.23}$$

The non-trivial component here is **H**, the matrix of partials $\frac{\partial \beta_i(\mathbf{b})}{\partial \mathbf{b}}$. We can write the matrix of partials out explicitly as:

$$\mathbf{H} = \begin{bmatrix} \frac{\partial \beta_0}{\partial u} & \frac{\partial \beta_0}{\partial v} \\ \frac{\partial \beta_1}{\partial u} & \frac{\partial \beta_1}{\partial v} \\ \frac{\partial \beta_2}{\partial u} & \frac{\partial \beta_2}{\partial v} \end{bmatrix}. \tag{D.24}$$

Fortunately, the β_i functions are quite simple! So, the derivatives are really easy and this matrix works out to:

$$\mathbf{H} = \begin{bmatrix} -1 & -1 \\ 1 & 0 \\ 0 & 1 \end{bmatrix}. \tag{D.25}$$

Finally, and somewhat perversely, we point out that:

$$\bar{\mathbf{X}}\mathbf{H} = \left[\begin{array}{c|c|c} \\ \bar{\mathbf{x}}_0 & \bar{\mathbf{x}}_1 & \bar{\mathbf{x}}_2 \\ \\ \end{array}\right] \begin{bmatrix} -1 & -1 \\ 1 & 0 \\ 0 & 1 \end{bmatrix} = \left[\begin{array}{c|c} \\ \bar{\mathbf{x}}_1 - \bar{\mathbf{x}}_0 & \bar{\mathbf{x}}_2 - \bar{\mathbf{x}}_0 \\ \\ \end{array}\right] = \mathbf{D}_m \tag{D.26}$$

$$\mathbf{X}\mathbf{H} = \left[\begin{array}{c|c|c} \\ \mathbf{x}_0 & \mathbf{x}_1 & \mathbf{x}_2 \\ \\ \end{array}\right] \begin{bmatrix} -1 & -1 \\ 1 & 0 \\ 0 & 1 \end{bmatrix} = \left[\begin{array}{c|c} \\ \mathbf{x}_1 - \mathbf{x}_0 & \mathbf{x}_2 - \mathbf{x}_0 \\ \\ \end{array}\right] = \mathbf{D}_s, \tag{D.27}$$

and therefore

$$\mathbf{F} = \mathbf{X}\mathbf{H}\left(\bar{\mathbf{X}}\mathbf{H}\right)^{-1} \tag{D.28}$$

$$= \mathbf{D}_s \mathbf{D}_m^{-1} \tag{D.29}$$

Argh! It's exactly the same as the previous method we saw, but with a lot of basis function dreck and unnecessary derivatives to slug through! What was the point?!

Relax. You didn't go through a bunch of math of nothing. As I said before, there is indeed a point: once you've gone through this process once, you can use the exact same method to turn the crank and compute **F** for primitives that *aren't* triangles or tetrahedra.

D.3.3 Computing F for 3D Tetrahedra, the Basis Function Way

Let's turn the crank for a 3D tetrahedron. It contains 4 vertices, so we add a new index to $\mathbf{x}(\mathbf{b})$ and $\bar{\mathbf{x}}(\mathbf{b})$,

$$\mathbf{x}(\mathbf{b}) = \sum_{i=0}^{3} \mathbf{x}_i \beta_i(\mathbf{b}) \qquad \bar{\mathbf{x}}(\mathbf{b}) = \sum_{i=0}^{3} \bar{\mathbf{x}}_i \beta_i(\mathbf{b}), \tag{D.30}$$

one more barycentric coordinate to $\mathbf{b} = [u \;\; v \;\; w]^T$ and one more basis function:

$$\beta_0(\mathbf{b}) = 1 - u - v - w \qquad \beta_1(\mathbf{b}) = u \tag{D.31}$$

$$\beta_2(\mathbf{b}) = v \qquad \beta_2(\mathbf{b}) = w. \tag{D.32}$$

We're still looking for

$$\mathbf{F} = \mathbf{X}\mathbf{H}\left(\bar{\mathbf{X}}\mathbf{H}\right)^{-1}, \tag{D.33}$$

where

$$\bar{\mathbf{X}} = \left[\begin{array}{c|c|c|c} \bar{\mathbf{x}}_0 & \bar{\mathbf{x}}_1 & \bar{\mathbf{x}}_2 & \bar{\mathbf{x}}_3 \end{array}\right] \in \Re^{3\times 4} \tag{D.34}$$

$$\mathbf{X} = \left[\begin{array}{c|c|c|c} \mathbf{x}_0 & \mathbf{x}_1 & \mathbf{x}_2 & \mathbf{x}_3 \end{array}\right] \in \Re^{3\times 4}. \tag{D.35}$$

The matrix of partials becomes predictably larger and more unwieldy-looking,

$$\mathbf{H} = \begin{bmatrix} \frac{\partial \beta_0}{\partial u} & \frac{\partial \beta_0}{\partial v} & \frac{\partial \beta_0}{\partial w} \\ \frac{\partial \beta_1}{\partial u} & \frac{\partial \beta_1}{\partial v} & \frac{\partial \beta_1}{\partial w} \\ \frac{\partial \beta_2}{\partial u} & \frac{\partial \beta_2}{\partial v} & \frac{\partial \beta_2}{\partial w} \\ \frac{\partial \beta_3}{\partial u} & \frac{\partial \beta_3}{\partial v} & \frac{\partial \beta_3}{\partial w} \end{bmatrix}, \tag{D.36}$$

but as before, it works out to a simple result:

$$\mathbf{H} = \begin{bmatrix} -1 & -1 & -1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}. \tag{D.37}$$

Then, just like before, we end up with

$$\bar{\mathbf{X}}\mathbf{H} = \left[\begin{array}{c|c|c|c} \\ \bar{\mathbf{x}}_0 & \bar{\mathbf{x}}_1 & \bar{\mathbf{x}}_2 & \bar{\mathbf{x}}_3 \\ \\ \end{array}\right] \begin{bmatrix} -1 & -1 & -1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \left[\begin{array}{c|c|c} \\ \bar{\mathbf{x}}_1 - \bar{\mathbf{x}}_0 & \bar{\mathbf{x}}_2 - \bar{\mathbf{x}}_0 & \mathbf{x}_3 - \bar{\mathbf{x}}_0 \\ \\ \end{array}\right] = \mathbf{D}_m \tag{D.38}$$

$$\mathbf{X}\mathbf{H} = \left[\begin{array}{c|c|c|c} \\ \mathbf{x}_0 & \mathbf{x}_1 & \mathbf{x}_2 & \mathbf{x}_3 \\ \\ \end{array}\right] \begin{bmatrix} -1 & -1 & -1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \left[\begin{array}{c|c|c} \\ \mathbf{x}_1 - \mathbf{x}_0 & \mathbf{x}_2 - \mathbf{x}_0 & \mathbf{x}_3 - \mathbf{x}_0 \\ \\ \end{array}\right] = \mathbf{D}_s, \tag{D.39}$$

and arrive at the same result:

$$\mathbf{F} = \mathbf{X}\mathbf{H}\left(\bar{\mathbf{X}}\mathbf{H}\right)^{-1} \tag{D.40}$$

$$= \mathbf{D}_s\mathbf{D}_m^{-1}. \tag{D.41}$$

D.3.4 Computing F for 2D Quads, the Basis Function Way

For completeness, should really include this.

D.3.5 Computing F for 3D Hexahedra, the Basis Function Way

For a cube element, let's begin by defining everything over a canonical cube that goes from $[1, 1, 1]$ to $[-1, -1, -1]$. We order the vertices of this cube like this:

$$\begin{aligned} a &= [-1, -1, -1] \\ b &= [1, -1, -1] \\ c &= [-1, 1, -1] \\ d &= [1, 1, -1] \\ e &= [-1, -1, 1] \\ f &= [1, -1, 1] \\ g &= [-1, 1, 1] \\ h &= [1, 1, 1] \end{aligned}$$

The vertices are traversed in counter-clockwise order, starting with the square in the $z = -1$ plane, and followed by the one in the $z = +1$ plane. While it's not really "barycentric", we have a coordinate $\mathbf{b} = [u \;\; v \;\; w]^T$ that we use to address every point inside the canonical space. The shape functions using $\mathbf{b}$ are then:

$$\begin{aligned}
\beta_0(\mathbf{b}) &= (1-u)(1-v)(1-w)\\
\beta_1(\mathbf{b}) &= (1+u)(1-v)(1-w)\\
\beta_2(\mathbf{b}) &= (1-u)(1+v)(1-w)\\
\beta_3(\mathbf{b}) &= (1+u)(1+v)(1-w)\\
\beta_4(\mathbf{b}) &= (1-u)(1-v)(1+w)\\
\beta_5(\mathbf{b}) &= (1+u)(1-v)(1+w)\\
\beta_6(\mathbf{b}) &= (1-u)(1+v)(1+w)\\
\beta_7(\mathbf{b}) &= (1+u)(1+v)(1+w)
\end{aligned}$$

The matrix of partials is a real eyesore this time around,

$$\mathbf{H} = \begin{bmatrix}
\frac{\partial\beta_0}{\partial u} & \frac{\partial\beta_0}{\partial v} & \frac{\partial\beta_0}{\partial w}\\
\frac{\partial\beta_1}{\partial u} & \frac{\partial\beta_1}{\partial v} & \frac{\partial\beta_1}{\partial w}\\
\frac{\partial\beta_2}{\partial u} & \frac{\partial\beta_2}{\partial v} & \frac{\partial\beta_2}{\partial w}\\
\frac{\partial\beta_3}{\partial u} & \frac{\partial\beta_3}{\partial v} & \frac{\partial\beta_3}{\partial w}\\
\frac{\partial\beta_4}{\partial u} & \frac{\partial\beta_4}{\partial v} & \frac{\partial\beta_4}{\partial w}\\
\frac{\partial\beta_5}{\partial u} & \frac{\partial\beta_5}{\partial v} & \frac{\partial\beta_5}{\partial w}\\
\frac{\partial\beta_6}{\partial u} & \frac{\partial\beta_6}{\partial v} & \frac{\partial\beta_6}{\partial w}\\
\frac{\partial\beta_7}{\partial u} & \frac{\partial\beta_7}{\partial v} & \frac{\partial\beta_7}{\partial w}
\end{bmatrix}, \tag{D.42}$$

and unlike before, this time it *does not* work out to a matrix of constants:

$$\mathbf{H} = \begin{bmatrix}
-(1-v)(1-w) & -(1-u)(1-w) & -(1-u)(1-v)\\
+(1-v)(1-w) & -(1+u)(1-w) & -(1+u)(1-v)\\
-(1+v)(1-w) & +(1-u)(1-w) & -(1-u)(1+v)\\
+(1+v)(1-w) & +(1+u)(1-w) & -(1+u)(1+v)\\
-(1-v)(1+w) & -(1-u)(1+w) & +(1-u)(1-v)\\
+(1-v)(1+w) & -(1+u)(1+w) & +(1+u)(1-v)\\
-(1+v)(1+w) & +(1-u)(1+w) & +(1-u)(1+v)\\
+(1+v)(1+w) & +(1+u)(1+w) & +(1+u)(1+v)
\end{bmatrix}. \tag{D.43}$$

Typing this equation into C++ is an irritating and error-prone process, so I have listed my implementation of in Fig. D.3. Let me know if you find any typos.

If we want to compute $\mathbf{F} = \mathbf{X}\mathbf{H}\left(\bar{\mathbf{X}}\mathbf{H}\right)^{-1}$, we first build out the $\mathbf{X}$ and $\bar{\mathbf{X}}$ matrices, which now contains all eight vertices of the cube:

$$\mathbf{X} = \left[\begin{array}{c|c|c|c|c|c|c|c} \mathbf{x}_0 & \mathbf{x}_1 & \mathbf{x}_2 & \mathbf{x}_3 & \mathbf{x}_4 & \mathbf{x}_5 & \mathbf{x}_6 & \mathbf{x}_7 \end{array}\right] \in \Re^{3\times 8} \tag{D.44}$$

and

$$\bar{\mathbf{X}} = \left[\begin{array}{c|c|c|c|c|c|c|c} \bar{\mathbf{x}}_0 & \bar{\mathbf{x}}_1 & \bar{\mathbf{x}}_2 & \bar{\mathbf{x}}_3 & \bar{\mathbf{x}}_4 & \bar{\mathbf{x}}_5 & \bar{\mathbf{x}}_6 & \bar{\mathbf{x}}_7 \end{array} \right] \in \Re^{3\times 8}. \tag{D.45}$$

Since $\mathbf{H}$ is not constant, we have to select $\mathbf{b}$, the location of the *quadrature point*. In the case of triangles and tetrahedra (with linear basis functions) that we saw before, $\mathbf{F}$ was uniquely defined over the entire primitive. With a hexahedron, $\mathbf{F}$ is different depending on where you are in the cube, so you need to specify the exact location you want to sample $\mathbf{F}$ at.

Just like with tetrahedra, you can pre-cache $\mathbf{D}_m^{-1}$ at startup, but in this case, you can actually do a little better. In the entire expression $\mathbf{F} = \mathbf{XH}\left(\bar{\mathbf{X}}\mathbf{H}\right)^{-1}$, the only matrix that will vary at runtime is $\mathbf{X}$, so why not go ahead and cache $\mathbf{H}\left(\bar{\mathbf{X}}\mathbf{H}\right)^{-1} = \mathbf{HD}_m^{-1}$? It will save you a matrix multiply every time. An example of what I mean is in Fig. D.4.

```
// Shape function derivative matrix
static Matrix8x3 ComputeH(const Vector3& b)
{
    const Scalar b0Minus = (1.0 - b[0]);
    const Scalar b0Plus  = (1.0 + b[0]);
    const Scalar b1Minus = (1.0 - b[1]);
    const Scalar b1Plus  = (1.0 + b[1]);
    const Scalar b2Minus = (1.0 - b[2]);
    const Scalar b2Plus  = (1.0 + b[2]);

    Matrix8x3 H;

    H(0,0) = -b1Minus * b2Minus;
    H(0,1) = -b0Minus * b2Minus;
    H(0,2) = -b0Minus * b1Minus;

    H(1,0) =  b1Minus * b2Minus;
    H(1,1) = -b0Plus * b2Minus;
    H(1,2) = -b0Plus * b1Minus;

    H(2,0) = -b1Plus * b2Minus;
    H(2,1) =  b0Minus * b2Minus;
    H(2,2) = -b0Minus * b1Plus;

    H(3,0) =  b1Plus * b2Minus;
    H(3,1) =  b0Plus * b2Minus;
    H(3,2) = -b0Plus * b1Plus;

    H(4,0) = -b1Minus * b2Plus;
    H(4,1) = -b0Minus * b2Plus;
    H(4,2) =  b0Minus * b1Minus;

    H(5,0) =  b1Minus * b2Plus;
    H(5,1) = -b0Plus * b2Plus;
    H(5,2) =  b0Plus * b1Minus;

    H(6,0) = -b1Plus * b2Plus;
    H(6,1) =  b0Minus * b2Plus;
    H(6,2) =  b0Minus * b1Plus;

    H(7,0) =  b1Plus * b2Plus;
    H(7,1) =  b0Plus * b2Plus;
    H(7,2) =  b0Plus * b1Plus;

    return H / 8.0;
}
```

Figure D.3.: Shape derivative matrix for Eqn. D.43.

```cpp
// quadrature point locations along the inside
// of a canonical cube
static const Scalar invSqrt3 = 1.0 / std::sqrt(3.0);
static const std::array<Vector3,8> restGaussPoints =
{
    invSqrt3 * Vector3(-1, -1, -1),
    invSqrt3 * Vector3( 1, -1, -1),
    invSqrt3 * Vector3(-1,  1, -1),
    invSqrt3 * Vector3( 1,  1, -1),
    invSqrt3 * Vector3(-1, -1,  1),
    invSqrt3 * Vector3( 1, -1,  1),
    invSqrt3 * Vector3(-1,  1,  1),
    invSqrt3 * Vector3( 1,  1,  1)
};

// cache DmInv at startup
std::array<Matrix8x3,8> ComputeHDmInv(const Matrix3x8& Xbar)
{
    std::array<Matrix8x3,8> HDmInv;

    for (int x = 0; x < 8; x++)
    {
        // compute Dm
        const Matrix8x3 H = ComputeH(restGaussPoints[x]);
        const Matrix3 Dm = Xbar * H;
        const Matrix3 DmInverse = Dm.inverse();

        // cache it left-multiplied by H
        HDmInv[x] = H * DmInverse;
    }

    return HDmInv;
}
```

Figure D.4.: Caching $\mathbf{HD}_m^{-1}$ for hexahedra.

Appendix E

All the Details of $\frac{\partial \mathbf{F}}{\partial \mathbf{x}}$

Following on from §3.4.1 we want to compute

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

This is a 3rd-order tensor containing twelve matrices. First, we will show the form of each of the twelve scalar derivatives $\frac{\partial \mathbf{D}_s}{\partial x_i}$, and then we'll show each combined $\frac{\partial \mathbf{D}_s}{\partial x_i}\mathbf{D}_m^{-1}$.

Again, the entries of $\mathbf{D}_m$ and $\mathbf{D}_s$ are:

$$\mathbf{D}_s = \left[\begin{array}{c|c|c} \mathbf{x}_1 - \mathbf{x}_0 & \mathbf{x}_2 - \mathbf{x}_0 & \mathbf{x}_3 - \mathbf{x}_0 \end{array}\right] \quad \mathbf{D}_m = \left[\begin{array}{c|c|c} \bar{\mathbf{x}}_1 - \bar{\mathbf{x}}_0 & \bar{\mathbf{x}}_2 - \bar{\mathbf{x}}_0 & \bar{\mathbf{x}}_3 - \bar{\mathbf{x}}_0 \end{array}\right], \tag{E.2}$$

and we want the twelve resulting matrices from taking the derivative with respect to $\mathbf{x} = \begin{bmatrix} \mathbf{x}_0 \\ \mathbf{x}_1 \\ \mathbf{x}_2 \\ \mathbf{x}_3 \end{bmatrix} = \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{11} \end{bmatrix}$. These then get arranged like cheese slices on a cutting board:

$$\frac{\partial \mathbf{D}_s}{\partial \mathbf{x}} = \begin{bmatrix} \left[\frac{\partial \mathbf{D}_s}{\partial x_0}\right] \\ \left[\frac{\partial \mathbf{D}_s}{\partial x_1}\right] \\ \vdots \\ \left[\frac{\partial \mathbf{D}_s}{\partial x_{11}}\right] \end{bmatrix} \tag{E.3}$$

Finally, here are the all the actual entries of $\frac{\partial \mathbf{D}_s}{\partial \mathbf{x}}$:

$$
\frac{\partial \mathbf{D}_s}{\partial x_0} = \begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}
\quad
\frac{\partial \mathbf{D}_s}{\partial x_1} = \begin{bmatrix} 0 & 0 & 0 \\ -1 & -1 & -1 \\ 0 & 0 & 0 \end{bmatrix}
\quad
\frac{\partial \mathbf{D}_s}{\partial x_2} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{bmatrix}
$$

$$
\frac{\partial \mathbf{D}_s}{\partial x_3} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}
\quad
\frac{\partial \mathbf{D}_s}{\partial x_4} = \begin{bmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}
\quad
\frac{\partial \mathbf{D}_s}{\partial x_5} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix}
$$

$$
\frac{\partial \mathbf{D}_s}{\partial x_6} = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}
\quad
\frac{\partial \mathbf{D}_s}{\partial x_7} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}
\quad
\frac{\partial \mathbf{D}_s}{\partial x_8} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix}
$$

$$
\frac{\partial \mathbf{D}_s}{\partial x_9} = \begin{bmatrix} 0 & 0 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}
\quad
\frac{\partial \mathbf{D}_s}{\partial x_{10}} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix}
\quad
\frac{\partial \mathbf{D}_s}{\partial x_{11}} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}.
$$

All ones and zeros! There are a lot of them, but they are indeed quite simple. To write down the full $\frac{\partial \mathbf{F}}{\partial \mathbf{x}}$ expression, we now need to label the rows and columns of $\mathbf{D}_m^{-1}$, which we do thusly:

$$
\mathbf{D}_m^{-1} = \left[\begin{array}{c} \mathbf{r}_0 \\ \hline \mathbf{r}_1 \\ \hline \mathbf{r}_2 \end{array}\right] = \left[\begin{array}{c|c|c} \mathbf{c}_0 & \mathbf{c}_1 & \mathbf{c}_2 \end{array}\right]. \tag{E.4}
$$

Now we can list the results of multiplying through each $\frac{\partial \mathbf{F}}{\partial x_i} = \frac{\partial \mathbf{D}_s}{\partial x_i} \mathbf{D}_m^{-1}$:

$$
\frac{\partial \mathbf{F}}{\partial x_0} = \begin{bmatrix} -s_0 & -s_1 & -s_2 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}
\quad
\frac{\partial \mathbf{F}}{\partial x_1} = \begin{bmatrix} 0 & 0 & 0 \\ -s_0 & -s_1 & -s_2 \\ 0 & 0 & 0 \end{bmatrix}
\quad
\frac{\partial \mathbf{F}}{\partial x_2} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ -s_0 & -s_1 & -s_2 \end{bmatrix}
$$

$$
\frac{\partial \mathbf{F}}{\partial x_3} = \begin{bmatrix} & \mathbf{r}_0 & \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}
\quad
\frac{\partial \mathbf{F}}{\partial x_4} = \begin{bmatrix} 0 & 0 & 0 \\ & \mathbf{r}_0 & \\ 0 & 0 & 0 \end{bmatrix}
\quad
\frac{\partial \mathbf{F}}{\partial x_5} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ & \mathbf{r}_0 & \end{bmatrix}
$$

$$
\frac{\partial \mathbf{F}}{\partial x_6} = \begin{bmatrix} & \mathbf{r}_1 & \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}
\quad
\frac{\partial \mathbf{F}}{\partial x_7} = \begin{bmatrix} 0 & 0 & 0 \\ & \mathbf{r}_1 & \\ 0 & 0 & 0 \end{bmatrix}
\quad
\frac{\partial \mathbf{F}}{\partial x_8} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ & \mathbf{r}_1 & \end{bmatrix}
$$

$$
\frac{\partial \mathbf{F}}{\partial x_9} = \begin{bmatrix} & \mathbf{r}_2 & 0 \\ 0 & 0 & \\ 0 & 0 & 0 \end{bmatrix}
\quad
\frac{\partial \mathbf{F}}{\partial x_{10}} = \begin{bmatrix} 0 & 0 & 0 \\ & \mathbf{r}_2 & \\ 0 & 0 & 0 \end{bmatrix}
\quad
\frac{\partial \mathbf{F}}{\partial x_{11}} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ & \mathbf{r}_2 & \end{bmatrix}
$$

In the first row, s_i denotes the sum of the entries in $\mathbf{c}_i$. Again, there's a lot of stuff here, but each individual piece is not very complicated. The final matrix entries are all just rows and columns from $\mathbf{D}_m^{-1}$.

As a big birthday present to you (and Future Me), the code for the flattened version of this tensor, $\text{vec}\left(\frac{\partial \mathbf{F}}{\partial \mathbf{x}}\right)$ is given in Fig. E.1. Finally, it is important to point out that this code *only applies to tetrahedra*. Hexahedra and other polyhedra are another can of worms.

```cpp
static Matrix9x12 ComputePFPx(const Matrix3& DmInv)
{
    const Scalar m = DmInv(0, 0);
    const Scalar n = DmInv(0, 1);
    const Scalar o = DmInv(0, 2);
    const Scalar p = DmInv(1, 0);
    const Scalar q = DmInv(1, 1);
    const Scalar r = DmInv(1, 2);
    const Scalar s = DmInv(2, 0);
    const Scalar t = DmInv(2, 1);
    const Scalar u = DmInv(2, 2);

    const Scalar t1 = - m - p - s;
    const Scalar t2 = - n - q - t;
    const Scalar t3 = - o - r - u;

    Matrix9x12 PFPx;
    PFPx.setZero();
    PFPx(0, 0) = t1;
    PFPx(0, 3) = m;
    PFPx(0, 6) = p;
    PFPx(0, 9) = s;
    PFPx(1, 1) = t1;
    PFPx(1, 4) = m;
    PFPx(1, 7) = p;
    PFPx(1, 10) = s;
    PFPx(2, 2) = t1;
    PFPx(2, 5) = m;
    PFPx(2, 8) = p;
    PFPx(2, 11) = s;
    PFPx(3, 0) = t2;
    PFPx(3, 3) = n;
    PFPx(3, 6) = q;
    PFPx(3, 9) = t;
    PFPx(4, 1) = t2;
    PFPx(4, 4) = n;
    PFPx(4, 7) = q;
    PFPx(4, 10) = t;
    PFPx(5, 2) = t2;
    PFPx(5, 5) = n;
    PFPx(5, 8) = q;
    PFPx(5, 11) = t;
    PFPx(6, 0) = t3;
    PFPx(6, 3) = o;
    PFPx(6, 6) = r;
    PFPx(6, 9) = u;
    PFPx(7, 1) = t3;
    PFPx(7, 4) = o;
    PFPx(7, 7) = r;
    PFPx(7, 10) = u;
    PFPx(8, 2) = t3;
    PFPx(8, 5) = o;
    PFPx(8, 8) = r;
    PFPx(8, 11) = u;

    return PFPx;
}
```

Figure E.1.: C++ code to compute the flattened version of $\frac{\partial \mathbf{F}}{\partial \mathbf{x}}$.

Appendix F

Rotation-Variant SVD and Polar Decomposition

The problem described in §2.3.4.4 is that the traditional polar decomposition, which we will call $\mathbf{F} = \hat{\mathbf{R}}\hat{\mathbf{S}}$, only requires $\hat{\mathbf{R}}$ to be unitary. A troublesome reflection can then lurk in $\hat{\mathbf{R}}$ that prevents it from being a pure rotation. We will now describe a simple way to compute the *rotation variant* of the polar decomposition. Along the way, we will also show how to do the same for the SVD.

A straightforward way to compute the polar decomposition of $\mathbf{F}$ is to first compute its SVD,

$$\mathbf{F} = \hat{\mathbf{U}}\hat{\Sigma}\hat{\mathbf{V}}^T, \tag{F.1}$$

and then recombine its components into $\hat{\mathbf{R}}$ and $\hat{\mathbf{S}}$:

$$\hat{\mathbf{R}} = \hat{\mathbf{U}}\hat{\mathbf{V}}^T \tag{F.2}$$

$$\hat{\mathbf{S}} = \hat{\mathbf{V}}\hat{\Sigma}\hat{\mathbf{V}}^T \tag{F.3}$$

Since $\hat{\mathbf{U}}$ and $\hat{\mathbf{V}}$ are also unitary, these representations are all trivially equivalent:

$$\begin{aligned}\hat{\mathbf{R}}\hat{\mathbf{S}} &= \hat{\mathbf{U}}\hat{\mathbf{V}}^T\hat{\mathbf{V}}\hat{\Sigma}\hat{\mathbf{V}}^T && \text{(F.4)}\\ &= \hat{\mathbf{U}}\mathbf{I}\hat{\Sigma}\hat{\mathbf{V}}^T && \text{(Used unitary property)}\\ &= \hat{\mathbf{U}}\hat{\Sigma}\hat{\mathbf{V}}^T = \mathbf{F}. && \text{(Definition of SVD)}\end{aligned}$$

What we want is to generate purely rotational versions of $\hat{\mathbf{U}}$ and $\hat{\mathbf{V}}$, i.e. $\mathbf{U}$ and $\mathbf{V}$. Then we can combine them into another pure rotation, $\mathbf{R} = \mathbf{U}\mathbf{V}^T$.

One easy way to test whether $\hat{\mathbf{R}}$ is secretly harboring a reflection is to look at the *determinant* of $\hat{\mathbf{R}}$, i.e. $\det \hat{\mathbf{R}}$. The determinant is the product of the singular values of a matrix, so if $\hat{\mathbf{R}}$ was a pure rotation, all its singular values would be $\Sigma = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$, and $\det \hat{\mathbf{R}} = 1$. However, if it contains a reflection, *one* singular value must be -1, $\Sigma = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -1 \end{bmatrix}$, and $\det \hat{\mathbf{R}} = -1$.

There's no such thing as a double reflection, only a single reflection. A double-reflection is the same as *rotating by* π, which is a valid rotation. In that case, the rotation would have been loaded into $\hat{\mathbf{U}}$ or $\hat{\mathbf{V}}$.

With this in hand, we can now build a *reflection matrix*:

$$\mathbf{L} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & \det\left(\hat{\mathbf{U}}\hat{\mathbf{V}}^T\right) \end{bmatrix}. \tag{F.5}$$

Computing the rotation variant of Σ is now straightforward:

$$\Sigma = \hat{\Sigma}\mathbf{L}. \tag{F.6}$$

If there were no reflections lurking anywhere, then $\mathbf{L} = \mathbf{I}$, so no harm done. However, we have to be a little careful when computing the new $\mathbf{U}$ and $\mathbf{V}$. We need to check whether the reflection is lurking in either $\hat{\mathbf{U}}$ or $\hat{\mathbf{V}}$, and only apply $\mathbf{L}$ to that one:

$$\mathbf{U} = \begin{cases} \hat{\mathbf{U}}\mathbf{L} & \text{if } \det \hat{\mathbf{U}} < 0 \text{ and } \det \hat{\mathbf{V}} > 0 \\ \hat{\mathbf{U}} & \text{otherwise} \end{cases} \tag{F.7}$$

$$\mathbf{V} = \begin{cases} \hat{\mathbf{V}}\mathbf{L} & \text{if } \det \hat{\mathbf{U}} > 0 \text{ and } \det \hat{\mathbf{V}} < 0 \\ \hat{\mathbf{V}} & \text{otherwise.} \end{cases} \tag{F.8}$$

With the rotation-variant SVD in hand, computing the rotation-variant polar decomposition is easy:

$$\mathbf{R} = \mathbf{U}\mathbf{V}^T \tag{F.9}$$

$$\mathbf{S} = \mathbf{V}\Sigma\mathbf{V}^T. \tag{F.10}$$

Complete Matlab/Octave code for the 3D case is given in Figs. F.1 and F.2

Historically, the need for the rotation-variant SVD and polar decomposition has been observed by a variety of authors (Twigg and Kačić-Alesić (2010); Higham (2008); Irving et al. (2004)). The version we describe here with the slick determinant/reflection matrix trick is from Sorkine-Hornung and Rabinovich (2017).

```matlab
function [U Sigma V] = svd_rv(F)
  [U Sigma V] = svd(F);

  % reflection matrix
  L = eye(3,3);
  L(3,3) = det(U * V');

  % see where to pull the reflection out of
  detU = det(U);
  detV = det(V);
  if (detU < 0 && detV > 0)
      U = U * L;
  elseif (detU > 0 && detV < 0)
      V = V * L;
  end

  % push the reflection to the diagonal
  Sigma = Sigma * L;
end
```

Figure F.1.: Matlab code to compute the rotation-variant SVD.

```matlab
function [R S] = polar_decomposition_rv(F)
  [U Sigma V] = svd_rv(F);
  R = U * V';
  S = V * Sigma * V';
end
```

Figure F.2.: Matlab code to compute the rotation-variant polar decomposition.

Appendix G

Thin Shell Stretch and Shear Pseudocode

```
void
ClothFace::_ComputeStretchAndShearForcesAndJacobians()
{
    real uvs[3][2]
    _face->GetScaledFaceUVs(uvs)
    const real areaUV = _face->GetAreaUV()
    const real du1 = uvs[1][0] - uvs[0][0]
    const real du2 = uvs[2][0] - uvs[0][0]
    const real dv1 = uvs[1][1] - uvs[0][1]
    const real dv2 = uvs[2][1] - uvs[0][1]

    const real idenom = 1.0 / (du1 * dv2 - du2 * dv1)
    const Vec3& p0 = _particles[0]->GetPos()
    const Vec3 p1 = _particles[1]->GetPos() - p0
    const Vec3 p2 = _particles[2]->GetPos() - p0

    Vec3 wu(p1 * dv2 - p2 * dv1)
    wu *= idenom
    Vec3 wv(p2 * du1 - p1 * du2)
    wv *= idenom

    const real nwu = GetLength(wu)
    const real nwv = GetLength(wv)
    // Make sure we don't blow up from small norms
    const real invLenWu = 1.0 / max(nwu, 1.0E-6)
    const real invLenWv = 1.0 / max(nwv, 1.0E-6)
    const Vec3 wuhat = invLenWu * wu
    const Vec3 wvhat = invLenWv * wv

    Vec3 dwudp
    dwudp[0] = idenom * (uvs[1][1] - uvs[2][1])
    dwudp[1] = idenom * (uvs[2][1] - uvs[0][1])
    dwudp[2] = idenom * (uvs[0][1] - uvs[1][1])

    Vec3 dwvdp;
    dwvdp[0] = idenom * (uvs[2][0] - uvs[1][0])
    dwvdp[1] = idenom * (uvs[0][0] - uvs[2][0])
    dwvdp[2] = idenom * (uvs[1][0] - uvs[0][0])

    Vec3 dCudp[3]
    dCudp[0] = dwudp[0] * wuhat
    dCudp[1] = dwudp[1] * wuhat
    dCudp[2] = dwudp[2] * wuhat

    Vec3 dCvdp[3];
    dCvdp[0] = dwvdp[0] * wvhat
    dCvdp[1] = dwvdp[1] * wvhat
    dCvdp[2] = dwvdp[2] * wvhat

    const real Cu = nwu - 1
    const real Cv = nwv - 1
    const real uKs = _face->GetKsu() * areaUV
    const real vKs = _face->GetKsv() * areaUV
    const real uKd = _face->GetKdu() * areaUV
    const real vKd = _face->GetKdv() * areaUV
    real CuDot = 0.0
    real CvDot = 0.0
    for (i = 0; i < 3; i++) {
        const Vec3& vel = _particles[i]->GetVel()
        CuDot += Dot(dCudp[i], vel)
        CvDot += Dot(dCvdp[i], vel)
    }

    SymBlock33 ScaledProjWuhat
    ScaledProjWuhat.SetScaledComplement(invLenWu, wuhat)
    SymBlock33 ScaledProjWvhat
    ScaledProjWvhat.SetScaledComplement(invLenWv, wvhat)

    const real uGradFullScalar = (-uKs * Cu - uKd * CuDot)
    const real vGradFullScalar = (-vKs * Cv - vKd * CvDot)

    // If we insist on including the second partial term fpr
    // damping, we must make sure it does not become positive or
    // dfdx can become indefinite. If GradFullScalar becomes
    // positive, we drop the term.
    const real uGradScalar = uGradFullScalar > 0.0 ? -uKs * Cu :
   uGradFullScalar
    const real vGradScalar = vGradFullScalar > 0.0 ? -vKs * Cv :
   vGradFullScalar

    // CShear = cosine of angle between wuhat and wvhat
    const real CShear = Dot(wuhat, wvhat)
    // [I-wuhat*wuhat^T]wvhat
    const Vec3 wvhatProjOutWuhat = ProjectOut(wvhat, wuhat)
    // [I-wvhat*wvhat^T]wuhat
    const Vec3 wuhatProjOutWvhat = ProjectOut(wuhat, wvhat)

    Vec3 dCSheardp[3]
    real CShearDot = 0.0
    for (i = 0; i < 3; i++) {
        dCSheardp[i] = invLenWu * dwudp[i] * wvhatProjOutWuhat
        dCSheardp[i] += invLenWv * dwvdp[i] * wuhatProjOutWvhat
        CShearDot += Dot(dCSheardp[i], _particles[i]->GetVel())
    }

    const real shearKs = _face->GetShearKs() * areaUV
    const real shearKd = _face->GetShearKd() * areaUV
    const real shearMag = -shearKs * CShear - shearKd * CShearDot

    Vec3 shearKsddCSheardpi, shearKddCSheardpi
    Vec3 stretchKsdCuDdPi, stretchKsdCvDdPi
    Vec3 stretchKdDCudPi, stretchKdDCvdPi

    const bool CuPositive = Cu >= 0.0
    const bool CvPositive = Cv >= 0.0

    for (i = 0; i < 3; i++)
    {
        // _localPairToArrayIndex maps index pairs to upper diagonal
        // block array index. See pseudocode in Chapter 12.
        blkIdx = _localPairToArrayIndex(i, i);
        _forces[i] += uGradFullScalar * dCudp[i]
        _forces[i] += vGradFullScalar * dCvdp[i]
        _forces[i] += shearMag * dCSheardp[i]
        // Stretch terms
        _dfdx[blkIdx].AddScaledOuterProd(-uKs, dCudp[i])
        _dfdx[blkIdx].AddScaledOuterProd(-vKs, dCvdp[i])

        if (CuPositive) {
            const real scalarU = uGradScalar * dwudp[i] * dwudp[i]
            _dfdx[blkIdx].AddScalarMultiple(scalarU, ScaledProjWuhat)
        }

        if (CvPositive) {
            const real scalarV = vGradScalar * dwvdp[i] * dwvdp[i]
            _dfdx[blkIdx].AddScalarMultiple(scalarV, ScaledProjWvhat)
        }
        // Shear terms
        _dfdx[blkIdx].AddScaledOuterProd(-shearKs, dCSheardp[i])
        _dfdv[blkIdx].AddScaledOuterProd(-shearKd, dCSheardp[i])
        _dfdv[blkIdx].AddScaledOuterProd(-uKd, dCudp[i])
        _dfdv[blkIdx].AddScaledOuterProd(-vKd, dCvdp[i])

        const Vec3 stretchKsdCuDdPi = -uKs * dCudp[i]
        const Vec3 stretchKdDCudPi = -uKd * dCudp[i]
        const Vec3 stretchKsdCvDdPi = -vKs * dCvdp[i]
        const Vec3 stretchKdDCvdPi = -vKd * dCvdp[i]
        const Vec3 shearKsddCSheardpi = -shearKs * dCSheardp[i]
        const Vec3 shearKddCSheardpi = -shearKd * dCSheardp[i]

        for (j = i + 1; j < 3; j++)
        {
            blkIdx = _localPairToArrayIndex(i, j)
            // Stretch terms
            _dfdx[blkIdx].AddOuterProd(stretchKsdCuDdPi, dCudp[j])
            _dfdx[blkIdx].AddOuterProd(stretchKsdCvDdPi, dCvdp[j])
            _dfdv[blkIdx].AddOuterProd(stretchKdDCudPi, dCudp[j])
            _dfdv[blkIdx].AddOuterProd(stretchKdDCvdPi, dCvdp[j])

            if (CuPositive) {
                const real scalarU = uGradScalar * dwudp[i] * dwudp[j]
                _dfdx[blkIdx].AddScalarMultiple(scalarU, ScaledProjWuhat)
            }
            if (CvPositive) {
                const real scalarV = vGradScalar * dwvdp[i] * dwvdp[j]
                _dfdx[blkIdx].AddScalarMultiple(scalarV, ScaledProjWvhat)
            }
            // Shear terms
            // Off diagonal local blocks will need to be examined
            // for transpose before being accumulated into the global
            // system block entries, because off-digaonl shear blocks
            // are NOT symmetric like the onese from stretch.
            _dfdx[blkIdx].AddOuterProd(dCSheardp[j], shearKsddCSheardpi)
            _dfdv[blkIdx].AddOuterProd(dCSheardp[j], shearKddCSheardpi)
        }
    }
}
```

Appendix H

Thin Shell Bending Pseudocode

```
////////////////////////////////////////////////////////
//         p2
//        / | \
//       /  |  \
//      /   |   \
//     p0 n0|n1 p3
//      \   |   /
//       \  |  /
//        \ | /
//         p1
//
// e = p2 - p1 == p12
// n0 = p01 X p02
// n1 = p32 X p31
//
void DihedralEdge::ComputeBendForcesAndJacobians()
{
    Face* f0 = _edge->GetFace(0)
    Face* f1 = _edge->GetFace(1)

    const Vec3 p[4] = {_particles[0]->GetPos(),
                       _particles[1]->GetPos(),
                       _particles[2]->GetPos(),
                       _particles[3]->GetPos()}

    const Vec3 p12 = p2 - p1
    const Vec3 p02 = p2 - p0
    const Vec3 p32 = p2 - p3
    const Vec3 p01 = p1 - p0
    const Vec3 p31 = p1 - p3

    const Vec3 n0hat = f0->GetNormal()
    const Vec3 n1hat = f1->GetNormal()
    const Vec3 ehat = GetNormalized(p12)

    const real sinTheta = Clamp(Dot(Cross(n0hat, n1hat), ehat), -1.0, 1.0)
    const real cosTheta = Clamp(Dot(n0hat, n1hat), -1.0, 1.0)

    // One needs to clamp the inverse of the normal length from
    // becoming too small to prevent numerical issues including poor
    // conditioning of the linear system.
    const real invN0Mag = 1.0 / max(f0->_nMag, 0.1 * f0->GetAreaUV())
    const real invN1Mag = 1.0 / max(f1->_nMag, 0.1 * f1->GetAreaUV())

    // Effect of [I-n0hat*n0hat^T] on n1hat
    const Vec3 projectN0OutFromN1 = ProjectOut(n1hat, n0hat)
    // Effect of [I-n1hat*n1hat^T] on n0hat
    const Vec3 projectN1OutFromN0 = ProjectOut(n0hat, n1hat)

    Vec3 dCosThetadP[4]
    dCosThetadP[0] = -invN0Mag * Cross(p12, projectN0OutFromN1)

    dCosThetadP[1] = -invN0Mag * Cross(-p02, projectN0OutFromN1)
    dCosThetadP[1] += -invN1Mag * Cross(p32, projectN1OutFromN0)

    dCosThetadP[2] = -invN0Mag * Cross(p01, projectN0OutFromN1)
    dCosThetadP[2] += -invN1Mag * Cross(-p31, projectN1OutFromN0)

    dCosThetadP[3] = -invN1Mag * Cross(-p12, projectN1OutFromN0)

    const Vec3 n0hatXehat = Cross(n0hat, ehat)
    const Vec3 n1hatXehat = Cross(n1hat, ehat)
    // Effect of [I-n1hat*n1hat^T] on n0hatXehat
    const Vec3 projOutN1Dir = ProjectOut(n0hatXehat, n1hat)
    // Effect of [I-n0hat*n0hat^T] on n1hatXehat
    const Vec3 projOutN0Dir = ProjectOut(n1hatXehat, n0hat)

    Vec3 dSinThetadP[4]
    dSinThetadP[0] = -invN0Mag * Cross(p12, projOutN0Dir)

    dSinThetadP[1] = invN1Mag * Cross(p32, projOutN1Dir)
    dSinThetadP[1] += -invN0Mag * Cross(-p02, projOutN0Dir)

    dSinThetadP[2] = invN1Mag * Cross(-p31, projOutN1Dir)
    dSinThetadP[2] += -invN0Mag * Cross(p01, projOutN0Dir)

    dSinThetadP[3] = invN1Mag * Cross(-p12, projOutN1Dir)

    Vec3 dThetadP[4]
    real thetaDot = 0.0

    for (i = 0; i < 4; i++)
    {
        dThetadP[i] = dSinThetadP[i]
        dThetadP[i] *= cosTheta
        dThetadP[i] -= sinTheta * dCosThetadP[i]
        thetaDot += Dot(dThetadP[i], _particles[i]->GetVel())
    }

    const real theta = atan2(sinTheta, cosTheta)
    const real thetaRest = _edge->GetThetaRest()

    // Note: thetaDist does not account for windings
    const real thetaDist = theta - thetaRest
    // Compute scaling factor based on a factor of the average
    // height and area of incident faces in material space.
    const real scaleFactor = ComputeScaleFactor(f0,f1)
    const ksTilde = scalefactor * _edge->GetBendStiffness()
    const kdTilde = scalefactor * _edge->GetBendDamping()
    const real mag = -ksTilde * thetaDist + -kdTilde * thetaDot

    Vec3 spatialK_dThetadPi
    Vec3 kd_dThetadPi
    int blockIdxII, blockIdxIJ

    for (i = 0; i < 4; i++)
    {
        _force[i] += mag * dThetadP[i];
        spatialK_dThetadPi = -ksTilde * dThetadP[i];
        kd_dThetadPi = -kdTilde * dThetadP[i];
        // _localPairToArrayIndex maps index pairs to upper diagonal
        // block array index. See pseudocode in Chapter 12.
        blockIdxII = _localPairToArrayIndex(i,i)
        _dfdx[blockIdxII].AddOuterProd(spatialK_dThetadPi, dThetadP[i])
        _dfdv[blockIdxII].AddOuterProd(kd_dThetadPi, dThetadP[i])

        for (j = i + 1; j < 4; j++)
        {
            blockIdxIJ = _localPairToArrayIndex(i,j)
            // Off diagonal local blocks will need to be examined
            // for transpose before being accumulated into the global
            // system block entries.
            _dfdx[blockIdxIJ].AddOuterProd(spatialK_dThetadPi, dThetadP[j])
            _dfdv[blockIdxIJ].AddOuterProd(kd_dThetadPi, dThetadP[j])
        }
    }
}
```

Appendix I

Computing the Derivatives of a Triangle (and Edge) Normal

I.1 The Triangle Normal Gradient

You've probably been sent here by Chapter 14, after being told some horror story about how computing $\frac{\partial \mathbf{n}}{\partial \mathbf{x}}$ has broken the will of many a brave student, leaving a trail thick with shattered ambitions, like the ending of *The Lady From Shanghai* (1947).

Don't worry. Whoever told you that is just trying to act like they're the only one who can pull the $\frac{\partial \mathbf{n}}{\partial \mathbf{x}}$ Excalibur from the stone. It's not a big deal once you see that it's really a 3rdorder tensor, *sneakily disguised as a matrix*.

Just to review, we're looking at a triangle with vertices $\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3$, arranged in clockwise order (see Fig. 14.1 for a refresher). There's also a vertex $\mathbf{x}_0$ defined, which is the vertex that is colliding with the current triangle. Thus, there's a big caveat here: everything you'll see here is arranged for vertex-face collision force computation. If you're looking to compute the normal gradient for some other purpose, you'll need to figure out how to tweak the formulas here to suit your own needs. Once you see through the 3rdorder tensor disguise though, hopefully this will be much easier.

Anyway, back to the show. The normal is then defined as:

$$\mathbf{e}_0 = \mathbf{x}_3 - \mathbf{x}_2 \qquad \mathbf{e}_1 = \mathbf{x}_0 - \mathbf{x}_2 \qquad \mathbf{e}_2 = \mathbf{x}_1 - \mathbf{x}_2 \tag{I.1}$$

$$\mathbf{n} = \frac{\mathbf{e}_2 \times \mathbf{e}_0}{\|\mathbf{e}_2 \times \mathbf{e}_0\|}. \tag{I.2}$$

Just like in Chapter 14, let's go ahead and stack all three vertices into a vector $\mathbf{x}$:

$$\mathbf{x} = \begin{bmatrix} \mathbf{x}_0 \\ \mathbf{x}_1 \\ \mathbf{x}_2 \\ \mathbf{x}_3 \end{bmatrix} \in \Re^{12}. \tag{I.3}$$

Keeping with our existing convention, every scalar entry in $\mathbf{x}$ can then be written as x_i. As usual, taking the derivative of *anything* with respect to a scalar is straightforward, and $\mathbf{n}$ is no exception. Applying the chain rule, we get:

$$\frac{\partial \mathbf{n}}{\partial x_i} = \frac{1}{\|\mathbf{z}\|}\frac{\partial \mathbf{z}}{\partial x_i} - \frac{\mathbf{z}^T \frac{\partial \mathbf{z}}{\partial x_i}}{\left(\mathbf{z}^T\mathbf{z}\right)^{\frac{3}{2}}}\mathbf{z} \tag{I.4}$$

$$\mathbf{z} = \mathbf{e}_2 \times \mathbf{e}_0. \tag{I.5}$$

For convenience, we've abbreviated the cross product as $\mathbf{z}$. This isn't so bad, right? We can see $\frac{\partial \mathbf{n}}{\partial x_i} \in \Re^3$, so it's a friendly-looking 3-vector. There's a few pieces we haven't defined yet, like $\frac{\partial \mathbf{z}}{\partial x_i}$, but otherwise, this looks fairly tame. What's the big deal? Nothing here to drive us into the mouth of madness.

The difficulty appears if you try to write down some some master expression for the whole thing, $\frac{\partial \mathbf{n}}{\partial \mathbf{x}}$. Don't do it. It looks like it should yield a nice matrix $\frac{\partial \mathbf{n}}{\partial \mathbf{x}} \in \Re^{3\times 12}$, but don't be fooled. *It's actually a 3rdorder tensor,* $\frac{\partial \mathbf{n}}{\partial \mathbf{x}} \in \Re^{3\times 1\times 12}$. Writing down each individual $\Re^{3\times 1}$ entry is straightforward, but trying to write down something simple and compact for the whole $\Re^{3\times 12}$ matrix will drive you nuts. The structure lives in the $\Re^{3\times 1}$ blocks, not the untamed $\Re^{3\times 12}$ monster.

If we can write down an expression for $\frac{\partial \mathbf{z}}{\partial x_i}$, then we're home free. With that in hand, you can compute each $\frac{\partial \mathbf{n}}{\partial x_i}$, and stack all the results together into $\frac{\partial \mathbf{n}}{\partial \mathbf{x}}$. Getting each $\frac{\partial \mathbf{z}}{\partial x_i}$ is tedious, but straightforward. As an example, let's look at:

$$\frac{\partial \mathbf{z}}{\partial x_0} = \frac{\partial \mathbf{e}_2 \times \mathbf{e}_0}{\partial x_0} = \begin{bmatrix} 0 \\ \mathbf{e}_{0,z} - \mathbf{e}_{2,z} \\ -\mathbf{e}_{0,y} + \mathbf{e}_{2,y} \end{bmatrix}. \tag{I.6}$$

I know. I abused the notation here, and wrote junk like $\mathbf{e}_{0,z}$, which means the z component of $\mathbf{e}_0$, and $\mathbf{e}_{2,y}$, which means the y component of $\mathbf{e}_2$. I promise I won't make a habit of it. The point is, each $\frac{\partial \mathbf{z}}{\partial x_i}$ should work out to some simple expression. In fact, we can go ahead and get rid of the abusive notation, because $\mathbf{e}_0 = \mathbf{x}_3 - \mathbf{x}_2$, so we should be able to replace $\mathbf{e}_{0,z}$ with $\mathbf{x}_{3,z} - \mathbf{x}_{2,z} = x_{11} - x_8$.

Chugging through this way, we can get expressions for all $\frac{\partial \mathbf{z}}{\partial x_i}$:

$$\frac{\partial \mathbf{z}}{\partial x_0} = \begin{bmatrix} 0 \\ x_5 - x_{11} \\ x_{10} - x_4 \end{bmatrix} \qquad \frac{\partial \mathbf{z}}{\partial x_1} = \begin{bmatrix} x_{11} - x_5 \\ 0 \\ x_3 - x_9 \end{bmatrix} \qquad \frac{\partial \mathbf{z}}{\partial x_2} = \begin{bmatrix} x_4 - x_{10} \\ x_9 - x_3 \\ 0 \end{bmatrix} \tag{I.7}$$

$$\frac{\partial \mathbf{z}}{\partial x_3} = \begin{bmatrix} 0 \\ x_{11} - x_2 \\ x_1 - x_{10} \end{bmatrix} \qquad \frac{\partial \mathbf{z}}{\partial x_4} = \begin{bmatrix} x_2 - x_{11} \\ 0 \\ x_9 - x_0 \end{bmatrix} \qquad \frac{\partial \mathbf{z}}{\partial x_5} = \begin{bmatrix} x_{10} - x_1 \\ x_0 - x_9 \\ 0 \end{bmatrix} \tag{I.8}$$

$$\frac{\partial \mathbf{z}}{\partial x_9} = \begin{bmatrix} 0 \\ x_2 - x_5 \\ x_4 - x_1 \end{bmatrix} \qquad \frac{\partial \mathbf{z}}{\partial x_{10}} = \begin{bmatrix} x_5 - x_2 \\ 0 \\ x_0 - x_3 \end{bmatrix} \qquad \frac{\partial \mathbf{z}}{\partial x_{11}} = \begin{bmatrix} x_1 - x_4 \\ x_3 - x_0 \\ 0 \end{bmatrix}. \tag{I.9}$$

$$\frac{\partial \mathbf{z}}{\partial x_6} = \frac{\partial \mathbf{z}}{\partial x_7} = \frac{\partial \mathbf{z}}{\partial x_8} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}. \tag{I.10}$$

This sure is a lot of tedious data entry. If you're hoping that I already typed this all up and debugged it for you, you're in luck! A Matlab implementation is in Fig. I.1, and a C++ implementation is in Fig. I.2. These are written in terms of the $\mathbf{e}$ edges, not the x_i entries in $\mathbf{x}$, so there might be some code micro-optimizations that can be introduced here, if you're so inclined.

Once you have these in hand, you can compute the complete normal gradient $\frac{\partial \mathbf{n}}{\partial \mathbf{x}}$ by stacking together all the individual $\frac{\partial \mathbf{n}}{\partial x_i}$ terms. At that point, you're welcome to treat the result as just a vanilla matrix, such as when computing the $\frac{\partial \mathbf{n}}{\partial \mathbf{x}}^T \mathbf{t}$ term in Eqn. 14.20. Matlab and C++ code for the full $\frac{\partial \mathbf{n}}{\partial \mathbf{x}}$ computation is available in Figs. I.3 and I.4.

There. All done. No need to hide in the basement just because $\frac{\partial \mathbf{n}}{\partial \mathbf{x}}$ appeared at your front door.

I.2 The Triangle Normal Hessian

Just kidding, this horror movie has a sequel. Just when you thought you were safe from $\frac{\partial \mathbf{n}}{\partial \mathbf{x}}$, its bigger, meaner cousin, $\frac{\partial^2 \mathbf{n}}{\partial \mathbf{x}^2}$, is needed in Hessians like Eqn. 14.15. Fortunately, we're not looking at an *Alien* (1979) to *Aliens* (1986) jump in horror here, necessitating the deployment of innovative new techniques like nuking it from orbit. We're more in the realm of *Final Destination* (2000) to *Final Destination 5* (2011), where once you know the basic premise, the rest is a tedious retread of more-or-less the same thing.

This time around, $\frac{\partial^2 \mathbf{n}}{\partial \mathbf{x}^2}$ looks like it should be a 3rdorder tensor, i.e. $\Re^{3\times 12\times 12}$. But just like last time, it's sneakily an order larger, $\Re^{3\times 1\times 12\times 12}$. More importantly, all the basic

```matlab
function [final] = cross_gradient(x)
  v0 = x(1:3);
  v1 = x(4:6);
  v2 = x(7:9);
  v3 = x(10:12);

  e0 = v1 - v0;
  e1 = v2 - v0;
  e2 = v3 - v0;

  crossMatrix = zeros(3, 12);
  e0x = e0(1);
  e0y = e0(2);
  e0z = e0(3);

  e2x = e2(1);
  e2y = e2(2);
  e2z = e2(3);

  crossMatrix(:,1)  = [0;
                       (e0z - e2z);
                       (-e0y + e2y)];
  crossMatrix(:,2)  = [(-e0z + e2z);
                       0;
                       (e0x - e2x)];
  crossMatrix(:,3)  = [(e0y - e2y);
                       (-e0x + e2x);
                       0];
  crossMatrix(:,4)  = [0;
                       e2z;
                       -e2y];
  crossMatrix(:,5)  = [-e2z;
                       0;
                       e2x];
  crossMatrix(:,6)  = [e2y;
                       -e2x;
                       0];
  crossMatrix(:,7)  = [0 0 0]';
  crossMatrix(:,8)  = [0 0 0]';
  crossMatrix(:,9)  = [0 0 0]';
  crossMatrix(:,10) = [0;
                       -e0z;
                      e0y];
  crossMatrix(:,11) = [e0z;
                       0;
                       -e0x];
  crossMatrix(:,12) = [-e0y;
                       e0x;
                       0];

  final = crossMatrix;
end
```

Figure I.1.: Matlab implementation of the $\frac{\partial \mathbf{z}}{\partial \mathbf{x}} = \frac{\partial \mathbf{e}_2 \times \mathbf{e}_0}{\partial \mathbf{x}}$ term needed by $\frac{\partial \mathbf{n}}{\partial \mathbf{x}}$, which is needed in Eqn. I.4.

structure still just lives in each $\Re^3$ entry. We can write down each $\Re^3$ entry as follows:

$$\frac{\partial^2 \mathbf{n}}{\partial x_i \partial x_j} = \frac{-1}{\left(\mathbf{z}^T \mathbf{z}\right)^{\frac{3}{2}}} \left(\frac{\partial \mathbf{z}}{\partial x_j}^T \mathbf{z} \right) \frac{\partial \mathbf{z}}{\partial x_i} + \frac{1}{\|\mathbf{z}\|} \frac{\partial^2 \mathbf{z}}{\partial x_i \partial x_j} - \alpha \frac{\partial \mathbf{z}}{\partial x_j} - \frac{\partial \alpha}{\partial x_j} \mathbf{z} \tag{I.11}$$

```
///////////////////////////////////////////////////////////////////////
// gradient of the cross product used to compute the triangle normal,
// vertex-face case
///////////////////////////////////////////////////////////////////////
MATRIX3x12 crossGradientVF(const std::vector<VECTOR3>& e)
{
  MATRIX3x12 crossMatrix;

  const REAL e0x = e[0][0];
  const REAL e0y = e[0][1];
  const REAL e0z = e[0][2];

  const REAL e2x = e[2][0];
  const REAL e2y = e[2][1];
  const REAL e2z = e[2][2];

  crossMatrix.col(0) = VECTOR3(0, 0, 0);
  crossMatrix.col(1) = VECTOR3(0, 0, 0);
  crossMatrix.col(2) = VECTOR3(0, 0, 0);
  crossMatrix.col(3) = VECTOR3(0, -e0z, e0y);
  crossMatrix.col(4) = VECTOR3(e0z, 0, -e0x);
  crossMatrix.col(5) = VECTOR3(-e0y, e0x, 0);
  crossMatrix.col(6) = VECTOR3(0, (e0z - e2z), (-e0y + e2y));
  crossMatrix.col(7) = VECTOR3((-e0z + e2z), 0, (e0x - e2x));
  crossMatrix.col(8) = VECTOR3((e0y - e2y), (-e0x + e2x), 0);
  crossMatrix.col(9) = VECTOR3(0, e2z, -e2y);
  crossMatrix.col(10) = VECTOR3(-e2z, 0, e2x);
  crossMatrix.col(11) = VECTOR3(e2y, -e2x, 0);

  return crossMatrix;
}
```

Figure I.2.: C++ implementation of the $\frac{\partial \mathbf{z}}{\partial \mathbf{x}} = \frac{\partial \mathbf{e}_2 \times \mathbf{e}_0}{\partial \mathbf{x}}$ term needed by $\frac{\partial \mathbf{n}}{\partial \mathbf{x}}$, which is needed in Eqn. I.4.

Just to make things a little tidier later on, I've pulled off the following factor:

$$\alpha = \frac{\mathbf{z}^T \frac{\partial \mathbf{z}}{\partial x_i}}{\left(\mathbf{z}^T \mathbf{z}\right)^{\frac{3}{2}}}. \tag{I.12}$$

Two terms have now appeared that we don't already have in hand from before: $\frac{\partial \alpha}{\partial x_j}$ and $\frac{\partial^2 \mathbf{z}}{\partial x_i \partial x_j}$. The first one's just a scalar, so we can derive it directly:

$$\frac{\partial \alpha}{\partial x_j} = \frac{\frac{\partial \mathbf{z}}{\partial x_j}^T \frac{\partial \mathbf{z}}{\partial x_i}}{\left(\mathbf{z}^T \mathbf{z}\right)^{\frac{3}{2}}} + \frac{\mathbf{z}^T \frac{\partial^2 \mathbf{z}}{\partial x_i \partial x_j}}{\left(\mathbf{z}^T \mathbf{z}\right)^{\frac{3}{2}}} - 3 \frac{\mathbf{z}^T \frac{\partial \mathbf{z}}{\partial x_i}}{\left(\mathbf{z}^T \mathbf{z}\right)^{\frac{5}{2}}} \left(\frac{\partial \mathbf{z}}{\partial x_j}^T \mathbf{z} \right). \tag{I.13}$$

The second term, $\frac{\partial^2 \mathbf{z}}{\partial x_i \partial x_j}$, is where all the haven't-we-see-this-all-before tedium comes in. We have all the first derivatives of $\mathbf{z}$ in Eqns. I.7-I.9, so now we need to chug through and

```matlab
function [final] = normal_gradient(x)
  v0 = x(1:3);
  v1 = x(4:6);
  v2 = x(7:9);
  v3 = x(10:12);

  e0 = v1 - v0;
  e1 = v2 - v0;
  e2 = v3 - v0;

  crossed = cross(e2, e0);
  crossNorm = norm(crossed);
  crossNormCubed = (crossed' * crossed)^(1.5);
  crossMatrix = cross_gradient(x);

  final = zeros(3,12);
  for i = 1:12
    crossColumn = crossMatrix(:,i);
    final(:,i) = (1 / crossNorm) * crossColumn - ((crossed' * crossColumn) / crossNormCubed) * crossed;
  end
end
```

Figure I.3.: Matlab implementation of $\frac{\partial \mathbf{n}}{\partial \mathbf{x}}$. The cross product gradient, $\frac{\partial \mathbf{z}}{\partial \mathbf{x}} = \frac{\partial \mathbf{e}_2 \times \mathbf{e}_0}{\partial \mathbf{x}}$, called via `cross_gradient`, is given in Fig. I.1.

```cpp
///////////////////////////////////////////////////////////////////////////
// gradient of the triangle normal, vertex-face case
///////////////////////////////////////////////////////////////////////////
MATRIX3x12 normalGradientVF(const std::vector<VECTOR3>& e)
{
  VECTOR3 crossed = e[2].cross(e[0]);
  REAL crossNorm = crossed.norm();
  const REAL crossNormCubedInv = 1.0 / pow(crossed.dot(crossed), 1.5);
  MATRIX3x12 crossMatrix = crossGradientVF(e);

  MATRIX3x12 result;
  for (int i = 0; i < 12; i++)
  {
    const VECTOR3 crossColumn = crossMatrix.col(i);
    result.col(i) = (1.0 / crossNorm) * crossColumn -
                    ((crossed.dot(crossColumn)) * crossNormCubedInv) * crossed;
  }
  return result;
}
```

Figure I.4.: C++ implementation of $\frac{\partial \mathbf{n}}{\partial \mathbf{x}}$. The cross product gradient, $\frac{\partial \mathbf{z}}{\partial \mathbf{x}} = \frac{\partial \mathbf{e}_2 \times \mathbf{e}_0}{\partial \mathbf{x}}$, called via `crossGradientVF`, is given in Fig. I.2. Equivalent lines from the Matlab implementation are inlined in the comments.

take the scalar derivatives of all of those, again, with respect to the second variable.

$$\frac{\partial^2 \mathbf{z}}{\partial x_0 \partial x_5} = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} \quad \frac{\partial^2 \mathbf{z}}{\partial x_0 \partial x_4} = \begin{bmatrix} 0 \\ 0 \\ -1 \end{bmatrix} \quad \frac{\partial^2 \mathbf{z}}{\partial x_0 \partial x_{10}} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \quad \frac{\partial^2 \mathbf{z}}{\partial x_0 \partial x_{11}} = \begin{bmatrix} 0 \\ -1 \\ 0 \end{bmatrix} \tag{I.14}$$

$$\frac{\partial^2 \mathbf{z}}{\partial x_1 \partial x_3} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \quad \frac{\partial^2 \mathbf{z}}{\partial x_1 \partial x_5} = \begin{bmatrix} -1 \\ 0 \\ 0 \end{bmatrix} \quad \frac{\partial^2 \mathbf{z}}{\partial x_1 \partial x_9} = \begin{bmatrix} 0 \\ 0 \\ -1 \end{bmatrix} \quad \frac{\partial^2 \mathbf{z}}{\partial x_1 \partial x_{11}} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \tag{I.15}$$

$$\frac{\partial^2 \mathbf{z}}{\partial x_2 \partial x_3} = \begin{bmatrix} 0 \\ -1 \\ 0 \end{bmatrix} \quad \frac{\partial^2 \mathbf{z}}{\partial x_2 \partial x_4} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \quad \frac{240\partial^2 \mathbf{z}}{\partial x_2 \partial x_9} = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} \quad \frac{\partial^2 \mathbf{z}}{\partial x_2 \partial x_{10}} = \begin{bmatrix} -1 \\ 0 \\ 0 \end{bmatrix} \tag{I.16}$$

$$\frac{\partial^2\mathbf{z}}{\partial x_3\partial x_1} = \begin{bmatrix}0\\0\\1\end{bmatrix} \quad \frac{\partial^2\mathbf{z}}{\partial x_3\partial x_2} = \begin{bmatrix}0\\-1\\0\end{bmatrix} \quad \frac{\partial^2\mathbf{z}}{\partial x_3\partial x_{10}} = \begin{bmatrix}0\\0\\-1\end{bmatrix} \quad \frac{\partial^2\mathbf{z}}{\partial x_3\partial x_{11}} = \begin{bmatrix}0\\1\\0\end{bmatrix} \tag{I.17}$$

$$\frac{\partial^2\mathbf{z}}{\partial x_4\partial x_0} = \begin{bmatrix}0\\0\\-1\end{bmatrix} \quad \frac{\partial^2\mathbf{z}}{\partial x_4\partial x_2} = \begin{bmatrix}1\\0\\0\end{bmatrix} \quad \frac{\partial^2\mathbf{z}}{\partial x_4\partial x_9} = \begin{bmatrix}0\\0\\1\end{bmatrix} \quad \frac{\partial^2\mathbf{z}}{\partial x_4\partial x_{11}} = \begin{bmatrix}-1\\0\\0\end{bmatrix} \tag{I.18}$$

$$\frac{\partial^2\mathbf{z}}{\partial x_5\partial x_0} = \begin{bmatrix}0\\1\\0\end{bmatrix} \quad \frac{\partial^2\mathbf{z}}{\partial x_5\partial x_1} = \begin{bmatrix}-1\\0\\0\end{bmatrix} \quad \frac{\partial^2\mathbf{z}}{\partial x_5\partial x_9} = \begin{bmatrix}0\\-1\\0\end{bmatrix} \quad \frac{\partial^2\mathbf{z}}{\partial x_5\partial x_{10}} = \begin{bmatrix}1\\0\\0\end{bmatrix} \tag{I.19}$$

$$\frac{\partial^2\mathbf{z}}{\partial x_9\partial x_1} = \begin{bmatrix}0\\0\\-1\end{bmatrix} \quad \frac{\partial^2\mathbf{z}}{\partial x_9\partial x_2} = \begin{bmatrix}0\\1\\0\end{bmatrix} \quad \frac{\partial^2\mathbf{z}}{\partial x_9\partial x_4} = \begin{bmatrix}0\\0\\1\end{bmatrix} \quad \frac{\partial^2\mathbf{z}}{\partial x_9\partial x_5} = \begin{bmatrix}0\\-1\\0\end{bmatrix} \tag{I.20}$$

$$\frac{\partial^2\mathbf{z}}{\partial x_{10}\partial x_0} = \begin{bmatrix}0\\0\\1\end{bmatrix} \quad \frac{\partial^2\mathbf{z}}{\partial x_{10}\partial x_2} = \begin{bmatrix}-1\\0\\0\end{bmatrix} \quad \frac{\partial^2\mathbf{z}}{\partial x_{10}\partial x_3} = \begin{bmatrix}0\\0\\-1\end{bmatrix} \quad \frac{\partial^2\mathbf{z}}{\partial x_{10}\partial x_5} = \begin{bmatrix}1\\0\\0\end{bmatrix} \tag{I.21}$$

$$\frac{\partial^2\mathbf{z}}{\partial x_{11}\partial x_0} = \begin{bmatrix}0\\1\\0\end{bmatrix} \quad \frac{\partial^2\mathbf{z}}{\partial x_{11}\partial x_1} = \begin{bmatrix}1\\0\\0\end{bmatrix} \quad \frac{\partial^2\mathbf{z}}{\partial x_{11}\partial x_3} = \begin{bmatrix}0\\1\\0\end{bmatrix} \quad \frac{\partial^2\mathbf{z}}{\partial x_{11}\partial x_4} = \begin{bmatrix}-1\\0\\0\end{bmatrix}. \tag{I.22}$$

Ugh, like a said, it's all just a tedious retread. There's a lot of redundancy in these results though, so you can boil it down to a few mildly less ugly-looking cases:

$$\frac{\partial^2\mathbf{z}}{\partial x_1\partial x_{11}} = \frac{\partial^2\mathbf{z}}{\partial x_2\partial x_4} = \frac{\partial^2\mathbf{z}}{\partial x_5\partial x_{10}} = \begin{bmatrix}1\\0\\0\end{bmatrix} \tag{I.23}$$

$$\frac{\partial^2\mathbf{z}}{\partial x_0\partial x_5} = \frac{\partial^2\mathbf{z}}{\partial x_2\partial x_9} = \frac{\partial^2\mathbf{z}}{\partial x_3\partial x_{11}} = \begin{bmatrix}0\\1\\0\end{bmatrix} \tag{I.24}$$

$$\frac{\partial^2\mathbf{z}}{\partial x_0\partial x_{10}} = \frac{\partial^2\mathbf{z}}{\partial x_1\partial x_3} = \frac{\partial^2\mathbf{z}}{\partial x_4\partial x_9} = \begin{bmatrix}0\\0\\1\end{bmatrix} \tag{I.25}$$

$$\frac{\partial^2\mathbf{z}}{\partial x_1\partial x_5} = \frac{\partial^2\mathbf{z}}{\partial x_2\partial x_{10}} = \frac{\partial^2\mathbf{z}}{\partial x_4\partial x_{11}} = \begin{bmatrix}-1\\0\\0\end{bmatrix} \tag{I.26}$$

$$\frac{\partial^2\mathbf{z}}{\partial x_0\partial x_{11}} = \frac{\partial^2\mathbf{z}}{\partial x_2\partial x_3} = \frac{\partial^2\mathbf{z}}{\partial x_5\partial x_9} = \begin{bmatrix}0\\-1\\0\end{bmatrix} \tag{I.27}$$

$$\frac{\partial^2\mathbf{z}}{\partial x_0\partial x_4} = \frac{\partial^2\mathbf{z}}{\partial x_1\partial x_9} = \frac{\partial^2\mathbf{z}}{\partial x_3\partial x_{10}} = \begin{bmatrix}0\\0\\-1\end{bmatrix}. \tag{I.28}$$

Once again, to rescue you from data entry errors, I provide Matlab and C++ implementations for $\frac{\partial^2 \mathbf{z}}{\partial \mathbf{x}^2}$ in Figs. I.5 and I.6.

```
function [final] = cross_hessian(i,j)
  if (i > j)
    temp = j;
    j = i;
    i = temp;
  end

  % use zero indexing, because we're not cretins
  i = i - 1;
  j = j - 1;

  if ((i == 1 && j == 11) || (i == 2 && j == 4) || (i == 5 && j == 10))
    final = [1 0 0]';
    return;
  end

  if ((i == 0 && j == 5) || (i == 2 && j == 9) || (i == 3 && j == 11))
    final = [0 1 0]';
    return;
  end

  if ((i == 0 && j == 10) || (i == 1 && j == 3) || (i == 4 && j == 9))
    final = [0 0 1]';
    return;
  end

  if ((i == 1 && j == 5) || (i == 2 && j == 10) || (i == 4 && j == 11))
    final = [-1 0 0]';
    return;
  end

  if ((i == 0 && j == 11) || (i == 2 && j == 3) || (i == 5 && j == 9))
    final = [0 -1 0]';
    return;
  end

  if ((i == 0 && j == 4) || (i == 1 && j == 9) || (i == 3 && j == 10))
    final = [0 0 -1]';
    return;
  end

  final = [0 0 0]';
end
```

Figure I.5.: Matlab implementation of the cross product Hessian, $\frac{\partial^2 \mathbf{z}}{\partial \mathbf{x}^2} = \frac{\partial^2 \mathbf{e}_2 \times \mathbf{e}_0}{\partial \mathbf{x}^2}$, needed by the normal Hessian in Fig. I.7. This encodes all the cases specified in Eqns. I.23-I.28.

With that in hand, we have all the terms needed for the full normal Hessian, Eqn. I.11. The Matlab and C++ code is in Figs. I.7 and I.8. There. I sat through the terrible sequel so you didn't have to. Will this ever turn into a trilogy? Will you ever need $\frac{\partial^3 \mathbf{n}}{\partial \mathbf{x}^3}$? We seem safe for now, but you never know when some relative of Jason Voorhees might bubble up again from the bottom of the lake ...

```
///////////////////////////////////////////////////////////////////////
// one entry of the rank-3 hessian of the cross product used to compute
// the triangle normal, vertex-face case
///////////////////////////////////////////////////////////////////////
VECTOR3 crossHessianVF(const int iIn, const int jIn)
{
  int i = iIn;
  int j = jIn;

  if (i > j)
  {
    int temp = j;
    j = i;
    i = temp;
  }

  if ((i == 5 && j == 7)  || (i == 8 && j == 10) || (i == 4 && j == 11))
    return VECTOR3(1, 0, 0);

  if ((i == 6 && j == 11) || (i == 3 && j == 8) || (i == 5 && j == 9))
    return VECTOR3(0, 1, 0);

  if ((i == 4 && j == 6)  || (i == 7 && j == 9) || (i == 3 && j == 10))
    return VECTOR3(0, 0, 1);

  if ((i == 7 && j == 11) || (i == 4 && j == 8) || (i == 5 && j == 10))
    return VECTOR3(-1, 0, 0);

  if ((i == 5 && j == 6)  || (i == 8 && j == 9) || (i == 3 && j == 11))
    return VECTOR3(0, -1, 0);

  if ((i == 6 && j == 10) || (i == 3 && j == 7) || (i == 4 && j == 9))
    return VECTOR3(0, 0, -1);

  return VECTOR3(0, 0, 0);
}
```

Figure I.6.: C++ implementation of the cross product Hessian, $\frac{\partial^2 \mathbf{z}}{\partial \mathbf{x}^2} = \frac{\partial^2 \mathbf{e}_2 \times \mathbf{e}_0}{\partial \mathbf{x}^2}$, needed by the normal Hessian in Fig. I.8. This encodes all the cases specified in Eqns. I.23-I.28.

I.3 The Edge-Edge Normal Gradient

What're you doing back here? Probably sent here by §14.6 because it turns out the normal gradients for edge-edge collisions are not exactly the same as the vertex-face case. Again, it's not an *Alien* (1979) to *Aliens* (1986) level jump in difficulty. This time it's more of a return to comfortable familiarity, like *Blade* (1998) to *Blade 2* (2002).[1]

[1]For some reason, this Guillermo del Toro film doesn't get mentioned in the same breath as other improbably good sequels like *Spiderman 2* (2004) or *The Godfather Part II* (1974), but I'd watch it over *Pan's Labyrinth* (2006) any day.

```
function [xSlab ySlab zSlab] = normal_hessian(x)
  v0 = x(1:3);
  v1 = x(4:6);
  v2 = x(7:9);
  v3 = x(10:12);

  e0 = v1 - v0;
  e1 = v2 - v0;
  e2 = v3 - v0;

  crossed = cross(e2, e0);
  crossNorm = norm(crossed);
  crossGradient = cross_gradient(x);

  z = crossed;
  denom15 = (z' * z) ^ (1.5);

  zGrad = cross_gradient(x);

  xSlab = zeros(12,12);
  ySlab = zeros(12,12);
  zSlab = zeros(12,12);

  for j = 1:12
    for i = 1:12
      zGradi = crossGradient(:,i);
      zGradj = crossGradient(:,j);
      zHessianij = cross_hessian(i,j);
      alpha = alpha_Psi(x,i);
      alphaGradj = alpha_gradient(x,i,j);

      entry = -((zGradj' * z) / denom15) * zGradi + ...
              1 / norm(z) * zHessianij - alpha * zGradj - alphaGradj * z;

      xSlab(i,j) = entry(1);
      ySlab(i,j) = entry(2);
      zSlab(i,j) = entry(3);
    end
  end
end
```

Figure I.7.: Matlab implementation of the normal Hessian, $\frac{\partial^2 \mathbf{n}}{\partial \mathbf{x}^2}$, as described in Eqn. I.11. The corresponding cross-product Hessian is provided in Fig. I.5.

This time, the normal is defined as:

$$\mathbf{e}_0 = \mathbf{x}_1 - \mathbf{x}_0 \tag{I.29}$$

$$\mathbf{e}_1 = \mathbf{x}_3 - \mathbf{x}_2 \tag{I.30}$$

$$\mathbf{n} = \mathbf{e}_1 \times \mathbf{e}_0. \tag{I.31}$$

Then we have the same gradient as before,

$$\frac{\partial \mathbf{n}}{\partial x_i} = \frac{1}{\|\mathbf{z}\|} \frac{\partial \mathbf{z}}{\partial x_i} - \frac{\mathbf{z}^T \frac{\partial \mathbf{z}}{\partial x_i}}{\left(\mathbf{z}^T \mathbf{z}\right)^{\frac{3}{2}}} \mathbf{z}, \tag{I.32}$$

```cpp
//////////////////////////////////////////////////////////////////////
// hessian of the triangle normal, vertex-face case
//////////////////////////////////////////////////////////////////////
std::vector<MATRIX12> normalHessianVF(const std::vector<VECTOR3>& e)
{
  using namespace std;

  vector<MATRIX12> H(3);
  for (int i = 0; i < 3; i++)
    H[i].setZero();

  //crossed = cross(e2, e0);
  //crossNorm = norm(crossed);
  //crossGradient = cross_gradient(x);
  VECTOR3 crossed = e[2].cross(e[0]);
  MATRIX3x12 crossGrad = crossGradientVF(e);
  const VECTOR3& z = crossed;

  //denom15 = (z' * z) ^ (1.5);
  REAL denom15 = pow(crossed.dot(crossed), 1.5);
  REAL denom25 = pow(crossed.dot(crossed), 2.5);

  for (int j = 0; j < 12; j++)
    for (int i = 0; i < 12; i++)
    {
      VECTOR3 zGradi = crossGrad.col(i);
      VECTOR3 zGradj = crossGrad.col(j);
      VECTOR3 zHessianij = crossHessianVF(i,j);

      // z = cross(e2, e0);
      // zGrad = crossGradientVF(:,i);
      // alpha= (z' * zGrad) / (z' * z) ^ (1.5);
      REAL a = z.dot(crossGrad.col(i)) / denom15;

      // final = (zGradj' * zGradi) / denom15 + (z' * cross_hessian(i,j)) / denom15;
      // final = final - 3 * ((z' * zGradi) / denom25) * (zGradj' * z);
      REAL aGrad = (zGradj.dot(zGradi)) / denom15 +
                   z.dot(crossHessianVF(i,j)) / denom15;
      aGrad -= 3.0 * (z.dot(zGradi) / denom25) * zGradj.dot(z);

      //entry = -((zGradj' * z) / denom15) * zGradi + 1 / norm(z) * zHessianij -
      // alpha * zGradj - alphaGradj * z;
      VECTOR3 entry = -((zGradj.dot(z)) / denom15) * zGradi +
                        1.0 / z.norm() * zHessianij - a * zGradj - aGrad * z;

      H[0](i,j) = entry[0];
      H[1](i,j) = entry[1];
      H[2](i,j) = entry[2];
    }
  return H;
}
```

Figure I.8.: C++ implementation of the normal Hessian, $\frac{\partial^2 \mathbf{n}}{\partial \mathbf{x}^2}$, as described in Eqn. I.11. The corresponding cross-product Hessian is provided in Fig. I.6.

but this time around, $\mathbf{z} = \mathbf{e}_1 \times \mathbf{e}_0$, and $\mathbf{e}_1$ and $\mathbf{e}_0$ don't even mean the same thing as before. But, even with this slightly new expression, if we can get $\frac{\partial \mathbf{z}}{\partial x_i} = \frac{\partial \mathbf{e}_1 \times \mathbf{e}_0}{\partial x_i}$ squared away, then we're done.

Demonstrating the sort of reckless, irresponsible, gonna-get-us-all-killed behavior that I would never condone in a graduate student, I just went ahead and implemented this gradient in C++ first, without verifying things in Matlab. The C++ version of the edge-edge $\frac{\partial \mathbf{e}_1 \times \mathbf{e}_0}{\partial x_i}$ term is given in Fig. I.9, and the full edge-edge $\frac{\partial \mathbf{n}}{\partial \mathbf{x}}$ is available in Fig. I.10.

```
///////////////////////////////////////////////////////////////////////
// gradient of the cross product used to compute the normal,
// edge-edge case
///////////////////////////////////////////////////////////////////////
MATRIX3x12 crossGradientEE(const std::vector<VECTOR3>& e)
{
  MATRIX3x12 crossMatrix;

  const REAL e0x = e[0][0];
  const REAL e0y = e[0][1];
  const REAL e0z = e[0][2];

  const REAL e1x = e[1][0];
  const REAL e1y = e[1][1];
  const REAL e1z = e[1][2];

  crossMatrix.col(0) = VECTOR3(0, -e1z, e1y);
  crossMatrix.col(1) = VECTOR3(e1z, 0, -e1x);
  crossMatrix.col(2) = VECTOR3(-e1y, e1x, 0);

  crossMatrix.col(3) = VECTOR3(0, e1z, -e1y);
  crossMatrix.col(4) = VECTOR3(-e1z, 0, e1x);
  crossMatrix.col(5) = VECTOR3(e1y, -e1x, 0);

  crossMatrix.col(6) = VECTOR3(0, e0z, -e0y);
  crossMatrix.col(7) = VECTOR3(-e0z, 0, e0x);
  crossMatrix.col(8) = VECTOR3(e0y, -e0x, 0);

  crossMatrix.col(9)  = VECTOR3(0, -e0z, e0y);
  crossMatrix.col(10) = VECTOR3(e0z, 0, -e0x);
  crossMatrix.col(11) = VECTOR3(-e0y, e0x, 0);

  return crossMatrix;
}
```

Figure I.9.: C++ implementation of the $\frac{\partial \mathbf{z}}{\partial \mathbf{x}} = \frac{\partial \mathbf{e}_1 \times \mathbf{e}_0}{\partial \mathbf{x}}$ term needed by $\frac{\partial \mathbf{n}}{\partial \mathbf{x}}$ for the **edge-edge** case. If you compare this code to Fig. I.2, you see that there's no `VECTOR3(0, 0, 0);` entries in this one, because all four vertices are involved in computing the cross-product this time around.

```cpp
///////////////////////////////////////////////////////////////////////
// gradient of the normal, edge-edge case
///////////////////////////////////////////////////////////////////////
MATRIX3x12 normalGradientEE(const std::vector<VECTOR3>& e)
{
  VECTOR3 crossed = e[1].cross(e[0]);
  const REAL crossNorm = crossed.norm();
  const REAL crossNormInv = (crossNorm > 1e-8) ? 1.0 / crossed.norm() : 0.0;
  const REAL crossNormCubedInv = (crossNorm > 1e-8) ? 1.0 / pow(crossed.dot(crossed),
    1.5) : 0.0;
  MATRIX3x12 crossMatrix = crossGradientEE(e);

  MATRIX3x12 result;
  for (int i = 0; i < 12; i++)
  {
    const VECTOR3 crossColumn = crossMatrix.col(i);
    result.col(i) = crossNormInv * crossColumn -
                    ((crossed.dot(crossColumn)) * crossNormCubedInv) * crossed;
  }
  return result;
}
```

Figure I.10.: C++ implementation of the $\frac{\partial \mathbf{n}}{\partial \mathbf{x}}$ term needed for the **edge-edge** case. I put a bunch of guards in there in case the cross product is degenerate, but the disciplined thing to do would be to work through the cases in Wang (2014). Instead, this'll probably stick around until the code hits some cases that forces me to come back and revise these lines, cursing my past self for not doing the right thing in the first place.

I.4 The Edge-Edge Normal Hessian

The Hessian rolls forward analogous to the gradient. Once again, if the normal is defined slightly differently

$$\mathbf{e}_0 = \mathbf{x}_1 - \mathbf{x}_0 \tag{I.33}$$

$$\mathbf{e}_1 = \mathbf{x}_3 - \mathbf{x}_2 \tag{I.34}$$

$$\mathbf{n} = \mathbf{e}_1 \times \mathbf{e}_0, \tag{I.35}$$

then the $\frac{\partial^2 \mathbf{n}}{\partial \mathbf{x}^2}$ works out slightly differently. It's all still ones and zeros arranged in specific places, but the entries move around since the indices in $\mathbf{x}$ move around. Once again, with a reckless sense of abandon, I implemented the $\frac{\partial^2 \mathbf{e}_1 \times \mathbf{e}_0}{\partial \mathbf{x}}$ and $\frac{\partial^2 \mathbf{n}}{\partial \mathbf{x}}$ terms directly in C++, and listed them directly in Figs. I.11 and I.12.

```cpp
///////////////////////////////////////////////////////////////////////
// one entry of the rank-3 hessian of the cross product used to compute
// the triangle normal, edge-edge case
///////////////////////////////////////////////////////////////////////
VECTOR3 crossHessianEE(const int iIn, const int jIn)
{
  int i = iIn;
  int j = jIn;

  if (i > j)
  {
    int temp = j;
    j = i;
    i = temp;
  }

  if ((i == 1 && j == 11)  || (i == 2 && j == 7) ||
      (i == 4 && j == 8) || (i == 5 && j == 10))
    return VECTOR3(1, 0, 0);

  if ((i == 0 && j == 8) || (i == 2 && j == 9) ||
      (i == 3 && j == 11) || (i == 5 && j == 6))
    return VECTOR3(0, 1, 0);

  if ((i == 0 && j == 10)  || (i == 1 && j == 6) ||
      (i == 3 && j == 7) || (i == 4 && j == 9))
    return VECTOR3(0, 0, 1);

  if ((i == 1 && j == 8) || (i == 2 && j == 10) ||
      (i == 4 && j == 11) || (i == 5 && j == 7))
    return VECTOR3(-1, 0, 0);

  if ((i == 0 && j == 11) || (i == 2 && j == 6) ||
      (i == 3 && j == 8) || (i == 5 && j == 9))
    return VECTOR3(0, -1, 0);

  if ((i == 0 && j == 7) || (i == 1 && j == 9) ||
      (i == 3 && j == 10) || (i == 4 && j == 6))
    return VECTOR3(0, 0, -1);

  return VECTOR3(0, 0, 0);
}
```

Figure I.11.: C++ implementation of the $\frac{\partial^2 \mathbf{z}}{\partial \mathbf{x}^2} = \frac{\partial^2 \mathbf{e}_1 \times \mathbf{e}_0}{\partial \mathbf{x}^2}$ term needed by $\frac{\partial^2 \mathbf{n}}{\partial \mathbf{x}^2}$ for the **edge-edge** case.

```cpp
///////////////////////////////////////////////////////////////////////
// hessian of the triangle normal, edge-edge case
///////////////////////////////////////////////////////////////////////
std::vector<MATRIX12> normalHessianEE(const std::vector<VECTOR3>& e)
{
  using namespace std;

  vector<MATRIX12> H(3);
  for (int i = 0; i < 3; i++)
    H[i].setZero();

  VECTOR3 crossed = e[1].cross(e[0]);
  MATRIX3x12 crossGrad = crossGradientEE(e);
  const VECTOR3& z = crossed;

  //denom15 = (z' * z) ^ (1.5);
  REAL denom15 = pow(crossed.dot(crossed), 1.5);
  REAL denom25 = pow(crossed.dot(crossed), 2.5);

  for (int j = 0; j < 12; j++)
    for (int i = 0; i < 12; i++)
    {
      VECTOR3 zGradi = crossGrad.col(i);
      VECTOR3 zGradj = crossGrad.col(j);
      VECTOR3 zHessianij = crossHessianEE(i,j);

      // z = cross(e2, e0);
      // zGrad = crossGradientVF(:,i);
      // alpha= (z' * zGrad) / (z' * z) ^ (1.5);
      REAL a = z.dot(crossGrad.col(i)) / denom15;

      // final = (zGradj' * zGradi) / denom15 + (z' * cross_hessian(i,j)) / denom15;
      // final = final - 3 * ((z' * zGradi) / denom25) * (zGradj' * z);
      REAL aGrad = (zGradj.dot(zGradi)) / denom15 +
                   z.dot(crossHessianEE(i,j)) / denom15;
      aGrad -= 3.0 * (z.dot(zGradi) / denom25) * zGradj.dot(z);

      //entry = -((zGradj' * z) / denom15) * zGradi +
      //          1 / norm(z) * zHessianij -
      //          alpha * zGradj - alphaGradj * z;
      VECTOR3 entry = -((zGradj.dot(z)) / denom15) * zGradi +
                        1.0 / z.norm() * zHessianij -
                        a * zGradj - aGrad * z;

      H[0](i,j) = entry[0];
      H[1](i,j) = entry[1];
      H[2](i,j) = entry[2];
    }
  return H;
}
```

Figure I.12.: C++ implementation of the $\frac{\partial^2 \mathbf{n}}{\partial \mathbf{x}^2}$ term needed for the **edge-edge** case.

Appendix J

The `HOBAK` C++ Code

The `HOBAK` library was written entirely by Theodore Kim[1]. It is intended to illustrate many of the concepts from the course as clearly as possible, and as a consequence is currently *not optimized for performance*. Readers beware: anybody claiming a 5X speed win over this code is not actually claiming that much.

Some optimizations that it would be painful to live without have been implemented in child classes named `*_FASTER`. For example, there are the `TET_MESH` and `TET_MESH_FASTER` classes. If you want to see readable version of stiffness matrix assembly, look at:

`TET_MESH::computeHyperelasticClampedHessian().`

If you want to see a version that's not as painfully slow, take a look at:

`TET_MESH_FASTER::computeHyperelasticClampedHessian().`

`HOBAK` is Korean for *squash* or *pumpkin*. This is library for squashing things.

J.1 Coding Conventions

- All class names follow `MACRO_CASE`, i.e. all-caps separated by underscores.
- All variables follow camel case, e.g. `tetMesh`. Function names also follow this convention, e.g. `computeSurfaceTriangles()`.
- All member variables begin with a leading underscore, followed by the camel case variable name, e.g. `_triangles` and `_vertices`.
- Instead of `float` or `double`, we use `REAL`, which is defined in `SETTINGS.h` at the topmost directory. I supposed I could have done this with templates instead, but that seemed so very ugly.

[1]In part to avoid the rights issues surrounding code that David Eberle writes.

- All tabs are two spaces, not actual tabs.

J.2 Quick Start

`HOBAK` is designed to build and run with the smallest number of dependencies possible. All the necessary external libraries are already included in the `HOBAK/ext` directory. Hopefully, you will not need to download anything else.

The fastest way to get up and running is to pick a build architecture, and call 'make' on it. From the top level `HOBAK` directory, to build on Linux, just call `make linux`. It should put everything where it needs to go, build the `simulateScenes` project, and run a *Bunny Drop* simulation.

If you want to build on a Mac, just call `make mac`. This one might give you a build error saying it can't find GLUT, in which case you need to install XQuartz, because apparently Mac is just toooo goooood to have GLUT installed by default.

If you have a version of `g++` installed that supports OpenMP (the default Mac compiler, `clang`, does not), you can alternatively build everything on the Mac by calling `make mac_omp`. It will expect that `g++` already points to an OpenMP-capable compiler.

J.3 Building Other Projects

To build a project in `HOBAK` , you first need to pick an architecture. If you called `make linux` or `make mac`, then it already picked for you.

If you want to pick by hand, go to `HOBAK/projects/`, and copy one of the `include_top` files to `include_top.mk`. For example, to build on Linux, call `cp include_top.linux include_top.mk`. Again, if you already ran `make linux`, this was already done for you.

From there, if you want to build a specific project, go to its directory and call `make depend`, followed by `make`. For example, to build the regression test suite, do:

```
cd projects/regressionTests
make depend
make
```

You only need to call `make depend` once. It will build the dependency files, i.e. which files depend on which headers. After calling it once, you should only need to call `make` from then on. You should only need to run `make depend` again if you change the dependencies, such as adding a new file to the `Makefile`, or changing a header.[2]

J.4 Running the Code

Everything should be run from the `./bin` directory. To build and then run `regressionTests` on Linux, we would do:

[2]I am aware that there are these thingies called `scons` and `cmake` and I have strong opinions:

```
cd projects/regressionTests
make depend
make
cd ../../bin
./regressionTests
```

Projects that pop up an OpenGL window, like `simulateScenes` or `replayScenes` support the following keyboard commands:

- `a` will start and stop the animation.
- `q` will quit, and write out a JSON file of simulation data and a QuickTime movie of the animation. You need to have FFMPEG installed for the movie write to succeed.
- `Q` or `Esc` will quit without writing anything out.

J.5 Unit Testing

Some basic unit testing has been implemented using the Catch2 library. The following should build and run the unit tests:

```
cd projects/unitTests
make depend
make
cd ../../bin
./unitTests
```

Note: On the Mac, `clang` currently doesn't seem to like Catch2. However, building with `g++` installed through Homebrew seems to work just fine.

J.6 What's Missing

A bunch of features are missing, mostly because my coding time is limited. If you want to see these features implemented, either add them yourself or give me a grant.

- *Continuous Collision Detection (CCD)* and *Global Intersection Analysis (GIA)*: If something weird happens under violent collisions, e.g. triangles get snagged, this is probably why.
- *Line Search*: If the simulation explodes, this is probably why.
- *Shells and Strands*: For now, you can fake it with really thin tet meshes.

Theodore Kim
@_TheodoreKim

It is a truth universally acknowledged that if cmake doesn't work on the first try you are totally screwed.

12:00 PM · Jul 22, 2021 · Twitter Web App

J.7 External Libraries

HOBAK uses the following external libraries, all of which are included in the download, so that we don't run into versioning issues a year from now.

- Eigen
- Catch2
- RapidJSON
- Spectra
- GLVU

J.8 The tobj File Format

Several tet meshes are provided for you in ./data/ directory, such as a simple cube at different resolutions, and the bunny at several resolutions, etc. These are stored in a flat, text-based *.tobj format, which is similar to the popular `*.obj` format.

Vertices are specified the same way, and are numbered according to the order that they appear in the file

```
v 1.0 2.0 1.0
```

The only new addition is a tetrahedron tag has also been added, denoted `t`, which indicates with vertices compose a tetrahedron;

```
t 100 341 82 900
```

I sure hope the `t` tag wasn't already taken in the `*.obj` specification, because that will probably cause somebody some headaches later, and that person could be Future Me.

To make your life slightly easier, I have also included a Gmsh to Tobj converter, under the project `gmshToTobj`. It uses the Gmsh loader from Qingnan Zhou's PyMesh library. If you want to use some sweet new meshing library like fTetWild by Hu et al. (2020), you can still go nuts, but then convert its output into a form that HOBAK can ingest when you're all done.

Bibliography

Alastrué, V., E. Peña, M. Martínez, and M. Doblaré (2008). Experimental study and constitutive modelling of the passive mechanical properties of the ovine infrarenal vena cava tissue. *Journal of biomechanics 41*(14), 3038–3045.

Arruda, E. M. and M. C. Boyce (1993). A three-dimensional constitutive model for the large stretch behavior of rubber elastic materials. *Journal of Mechanics Physics of Solids 41*, 389–412.

Ascher, U. M. and E. Boxerman (2003). On the modified conjugate gradient method in cloth simulation. *The Visual Computer 19*(7-8), 526–531.

Autonne, L. (1902). Sur les groupes linéaires, réels et orthogonaux. *Bulletin de la société mathématique de France 30*, 121–134.

Baraff, D. and A. Witkin (1998). Large steps in cloth simulation. In *Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '98. ACM.

Baraff, D. and A. Witkin (2003). Untangling cloth. In *ACM SIGGRAPH 2003 Papers*, SIGGRAPH '03. ACM.

Barbic, J. (2012). Exact corotational linear fem stiffness matrix. Technical report, University of Southern California. Department of Computer Science.

Bargteil, A. and T. Shinar (2018). An introduction to physics-based animation. In *ACM SIGGRAPH Courses*, New York, NY, USA. ACM.

Belytschko, T., W. K. Liu, B. Moran, and K. Elkhodary (2013). *Nonlinear finite elements for continua and structures.* John wiley & sons.

Bergou, M., M. Wardetzky, D. Harmon, D. Zorin, and E. Grinspun (2002). A quadratic bending model for inextensible surfaces. In *Eurographics Symposium on Geometry Processing*, pp. 49–54.

Blair, P. (2003). *Animation 1: Learn to animate cartoons step by step.* Walter Foster Publishing.

Blemker, S. S., P. M. Pinsky, and S. L. Delp (2005). A 3d model of muscle reveals the causes of nonuniform strains in the biceps brachii. *Journal of biomechanics 38*(4), 657–665.

Bonet, J. and R. D. Wood (2008). *Nonlinear continuum mechanics for finite element analysis*. Cambridge university press.

Bower, A. F. (2009). *Applied mechanics of solids*. CRC press.

Bridson, R., R. Fedkiw, and J. Anderson (2002). Robust treatment of collisions, contact and friction for cloth animation. In *Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '02, pp. 594–603. ACM.

Brochu, T., E. Edwards, and R. Bridson (2012). Efficient geometrically exact continuous collision detection. *ACM Trans. Graph. 31*(4).

Bunch, J. R., C. P. Nielsen, and D. C. Sorensen (1978). Rank-one modification of the symmetric eigenproblem. *Numerische Mathematik 31*(1), 31–48.

Chagnon, G., M. Rebouah, and D. Favier (2015). Hyperelastic energy densities for soft biological tissues: a review. *Journal of Elasticity 120*(2), 129–160.

Chao, I., U. Pinkall, P. Sanan, and P. Schröder (2010, July). A simple geometric model for elastic deformations. *ACM Trans. Graph. 29*(4).

Choi, K.-J. and H.-S. Ko (2002, 07). Stable but responsive cloth. Volume 21. ACM.

Chow, E. and A. Patel (2015). Fine-grained parallel incomplete lu factorization. *SIAM journal on Scientific Computing 37*(2), C169–C193.

Espinosa, H. D., P. D. Zavattieri, and G. L. Emore (1998). Adaptive fem computation of geometric and material nonlinearities with application to brittle failure. *Mechanics of Materials 29*(3-4), 275–305.

Etzmuss, O., M. Keckeisen, and W. Strasser (2003, Oct). A fast finite element solution for cloth modelling. In *Proceedings of Pacific Graphics*, pp. 244–251.

Fung, Y.-c. (2013). *Biomechanics: mechanical properties of living tissues*. Springer Science & Business Media.

Gao, Z., T. Kim, D. L. James, and J. P. Desai (2009, Aug). Semi-automated soft-tissue acquisition and modeling for surgical simulation. In *IEEE International Conference on Automation Science and Engineering*, pp. 268–273.

Garg, A., E. Grinspun, M. Wardetzky, and D. Zorin (2007). Cubic shells. In *Symposium on Computer Animation*.

Golub, G. and W. Kahan (1965). Calculating the singular values and pseudo-inverse of a matrix. *Journal of the Society for Industrial and Applied Mathematics, Series B: Numerical Analysis 2*(2), 205–224.

Golub, G. H. and C. F. Van Loan (2013). Matrix computations. *The Johns Hopkins University Press*.

Greaves, G. N., A. Greer, R. S. Lakes, and T. Rouxel (2011). Poisson's ratio and modern materials. *Nature materials 10*(11), 823–837.

Gribbin, J. (2011). *In search of Schrodinger's cat: Quantum physics and reality*. Bantam.

Grinspun, E., A. Hirani, M. Desbrun, and P. Schröeder (2003). Discrete shells. In *SCA '03: Proceedings of the 2003 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*.

Guennebaud, G., B. Jacob, et al. (2010). Eigen v3. http://eigen.tuxfamily.org.

Han, H., M. Sun, S. Zhang, D. Liu, and T. Liu (2021). Gpu cloth simulation pipeline in lightchaser animation studio. In *SIGGRAPH Asia Technical Communications*.

Harmon, D., E. Vouga, R. Tamstorf, and E. Grinspun (2008). Robust treatment of simultaneous collisions. Volume 27. ACM.

Hauth, M. and O. Etzmuss (2001). A high performance solver for the animation of deformable objects using advanced numerical methods. *Eurographics 20*(3), 319–328.

Higham, N. J. (1986). Computing the polar decomposition—with applications. *SIAM Journal on Scientific and Statistical Computing 7*(4), 1160–1174.

Higham, N. J. (2008). *Functions of matrices: theory and computation*. SIAM.

Holzapfel, G. (2005). Similarities between soft biological tissues and rubberlike materials. In *Constitutive Models for Rubber IV*, pp. 607–617.

Hu, Y., T. Schneider, B. Wang, D. Zorin, and D. Panozzo (2020, July). Fast tetrahedral meshing in the wild. *ACM Trans. Graph. 39*(4).

Irving, G., J. Teran, and R. Fedkiw (2004). Invertible finite elements for robust simulation of large deformation. In *SIGGRAPH/Eurog. Symp. on Comp. Anim.*, pp. 131–140.

Jenkins, M. A. and J. F. Traub (1970). A three-stage variable-shift iteration for polynomial zeros and its relation to generalized rayleigh iteration. *Numerische Mathematik 14*(3), 252–263.

Johnston, O. and F. Thomas (1981). *The illusion of life: Disney animation*. Disney Editions New York.

Kautzman, R., G. Cameron, and T. Kim (2018). Robust skin simulation in incredibles 2. In *ACM SIGGRAPH Talks*, pp. 1–2.

Kim, T. (2020). A finite element formulation of baraff-witkin cloth. In *SCA '03: Proceedings of the 2020 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*.

Kim, T., F. De Goes, and H. Iben (2019, July). Anisotropic elasticity for inversion-safety and element rehabilitation. *ACM Trans. Graph. 38*(4).

Kim, T. and D. L. James (2012, Aug). Physics-based character skinning using multidomain subspace deformations. *IEEE Transactions on Visualization and Computer Graphics 18*(8), 1228–1240.

Kolda, T. G. and B. W. Bader (2009). Tensor decompositions and applications. *SIAM review 51*(3), 455–500.

Liu, T., S. Bouaziz, and L. Kavan (2017). Quasi-newton methods for real-time simulation of hyperelastic materials. *ACM Trans. Graph. (TOG) 36*(3), 1–16.

Löschner, F., A. Longva, S. Jeske, T. Kugelstadt, and J. Bender (2020). Higher-order time integration for deformable solids. In *Computer Graphics Forum*, Volume 39, pp. 157–169. Wiley Online Library.

Marschner, S. and P. Shirley (2016). *Fundamentals of Computer Graphics* (4th ed.). USA: CRC Presss.

Marsden, J. E. and T. J. Hughes (1994). *Mathematical foundations of elasticity*. Dover Publications.

McAdams, A., Y. Zhu, A. Selle, M. Empey, R. Tamstorf, J. Teran, and E. Sifakis (2011, 07). Efficient elasticity for character skinning with contact and collisions. *ACM Trans. Graph. 30*, 37.

Möller, T., E. Haines, and N. Hoffman (2008). *Real-Time Rendering* (3 ed.). A.K Peters Ltd.

Mooney, M. (1940). A theory of large elastic deformation. *Journal of Applied Physics 11*(9), 582–592.

Moore, M. and J. Wilhelms (1988). Collision detection and response for computer animation. In *ACM SIGGRAPH computer graphics*, Volume 22. ACM.

Müller, M., J. Dorsey, L. McMillan, R. Jagnow, and B. Cutler (2002). Stable real-time deformations. In *ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pp. 49–54.

Nocedal, J. and S. Wright (2006). *Numerical optimization*. Springer Science & Business Media.

Ogden, R. W. (1997). *Non-linear elastic deformations*. Dover Publications.

Papadopoulo, T. and M. I. Lourakis (2000). Estimating the jacobian of the singular value decomposition: Theory and applications. In *European Conference on Computer Vision*, pp. 554–570. Springer.

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

Pritchard, D. (2002). Implementing baraff witkin's cloth simulation.

Provot, X. (1995). Collision and self-collision handling in cloth model dedicated to design garment. *Graphics Interface*.

Provot, X. (1997). Collision and self-collision handling in cloth model dedicated to design garments. In *Computer Animation and Simulation*, pp. 177–189. Springer.

Rankin, C. C. and F. A. Brogan (1986, 05). An Element Independent Corotational Procedure for the Treatment of Large Rotations. *Journal of Pressure Vessel Technology 108*(2), 165–174.

Rhodes, G. (2001). Fast, robust intersection of 3d line segments. *Game Programming Gems 2*, 191–204.

Rivlin, R. (1948). Large elastic deformations of isotropic materials iv. further developments of the general theory. *Philosophical Transactions of the Royal Society of London. Series A, Mathematical and Physical Sciences 241*(835), 379–397.

Rupp, K., P. Tillet, F. Rudolf, J. Weinbub, A. Morhammer, T. Grasser, A. Jüngel, and S. Selberherr (2016). Viennacl—linear algebra library for multi-and many-core architectures. *SIAM Journal on Scientific Computing 38*(5), S412–S439.

Schneider, P. and D. Eberly (2003). *Geometric Tools for Computer Graphics*. Morgan Kaufmann Publishers.

Shtengel, A., R. Poranne, O. Sorkine-Hornung, S. Z. Kovalsky, and Y. Lipman (2017). Geometric optimization via composite majorization. *ACM Trans. Graph. 36*(4).

Sifakis, E. and J. Barbic (2012). Fem simulation of 3d deformable solids: A practitioner's guide to theory, discretization and model reduction. In *SIGGRAPH Courses*, pp. 20:1–20:50.

Simmonds, J. G. (2012). *A brief on tensor analysis*. Springer Science & Business Media.

Smith, B., F. D. Goes, and T. Kim (2018, March). Stable neo-hookean flesh simulation. *ACM Trans. Graph. 37*(2).

Smith, B., F. D. Goes, and T. Kim (2019, February). Analytic eigensystems for isotropic distortion energies. *ACM Trans. Graph. 38*(1).

Smith, J. and S. Schaefer (2015a, July). Bijective parameterization with free boundaries. *ACM Trans. Graph. 34*(4).

Smith, J. and S. Schaefer (2015b). Bijective parameterization with free boundaries. *ACM Trans. Graph. 34*(4).

Smith, O. K. (1961). Eigenvalues of a symmetric $3\times$ 3 matrix. *Communications of the ACM 4*(4), 168.

Sorkine, O. and M. Alexa (2007). As-rigid-as-possible surface modeling. In *Eurog. Symposium on Geometry processing*, Volume 4.

Sorkine-Hornung, O. and M. Rabinovich (2017). Least-squares rigid motion using svd. Technical report, ETH Zurich, Department of Computer Science.

Tamstorf, R. and E. Grinspun (2013). Discrete bending forces and their jacobians. In *Graphical Models Issue 75*.

Tamstorf, R., T. Jones, and S. McCormick (2015). Smoothed aggregation multigrid for cloth simulation. *ACM Trans. Graph. 34*(6).

Tang, M., R. Tong, Z. Wang, and D. Manocha (2014). Fast and exact continuous collision detection with bernstein sign classification. Volume 33, pp. 186:1–186:8.

Teran, J., E. Sifakis, S. S. Blemker, V. Ng-Thow-Hing, C. Lau, and R. Fedkiw (2005, May). Creating and simulating skeletal muscle from the visible human data set. *IEEE Transactions on Visualization and Computer Graphics 11*(3), 317–328.

Twigg, C. D. and Z. Kačić-Alesić (2010). Point cloud glue: Constraining simulations using the procrustes transform. In *ACM SIGGRAPH/Eurog. Symp. on Comp. Anim.*, pp. 45–54.

Wang, H. (2014). Defending continuous collision detection against errors. *ACM Trans. Graph. 33*(4).

Wang, H. and Y. Yang (2016). Descent methods for elastic body simulation on the gpu. *ACM Trans. Graph. 35*(6).

Witkin, A. and D. Baraff (1997). Physically based modeling: Principles and practice. In *SIGGRAPH Courses*.

Wong, A., D. Eberle, and T. Kim (2018). Clean cloth inputs: Removing character self-intersections with volume simulation. In *ACM SIGGRAPH Talks*, pp. 1–2.

Ye, J. (2009). A reduced unconstrained system for he cloth dynamics solver. *The Visual Computer. 25*(10).