

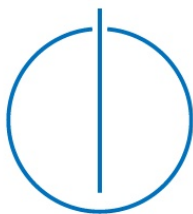
Technische Universität München

Fakultät für Informatik

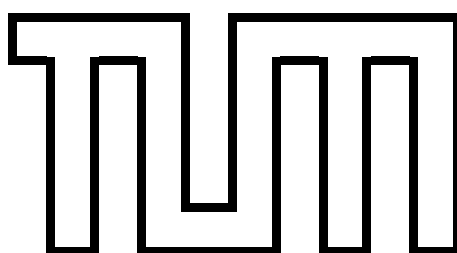
Master's Thesis in Informatics

# Physically-based Action Reconstruction

Sebastian Weiß







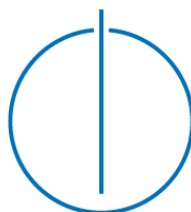
Technische Universität München

Fakultät für Informatik

Master's Thesis in Informatics

**Physically-based Action Reconstruction**  
Physikbasierte Rekonstruktion von Handlungen

Author: Sebastian Weiß  
1<sup>st</sup> Supervisor: Prof. Dr. Daniel Cremers  
2<sup>nd</sup> Supervisor: Prof. Dr. Nils Thuerey  
Submission Date: October 15th, 2018







I hereby declare that this thesis is entirely the result of my own work except where otherwise indicated. I have only used the resources given in the list of references.

Garching,

Sebastian Weiß



---

## Acknowledgments

First and foremost, I would like to thank my supervisors Prof. Dr. Nils Thuerey and Prof. Dr. Daniel Cremers for their helpful advice. I also wish to express my gratitude to Marvin Eisenberger and Dr. Dan Koschier for their feedback in several discussions.

I am thankful for the office and the computational resources provided at the chair of computer graphics. The pleasant working atmosphere and the exchange with members of the chair assisted me in writing this thesis.

---

*"To me, mathematics, computer science, and the arts are insanely related.  
They're all creative expressions."*

*- Sebastian Thrun*

---

## Abstract

In medical imaging or animation control the parameters of elasticity simulations are often unknown and need to be estimated. We present a method for reconstructing the physical material parameters of elastic bodies, the Young's Modulus, Poisson Ratio and mass, Rayleigh damping parameters, as well as ground collision configurations from observed images.

To achieve that goal, we first introduce a new discretization scheme of the linear elasticity PDE based on a regular grid with partially filled FE cells. This method was already successfully used for fluid simulations under the name 'cutFEM' or 'Immersed Boundaries' and is related to 'exFEM' as used for simulating cuts on tetrahedral meshes. We also show how to handle corotation and collisions in this framework. Next, we present an efficient implementation in CUDA that allows us to achieve interactive framerates on large objects. We then embed the simulation in an adjoint framework to reconstruct the unknown parameters. Furthermore, we introduce a cost function that uses sparse points of the object surface as observations. Therefore, feature tracking to obtain vertex displacements of the reference configuration is avoided.

Our method allows us to use signed distance fields as obtained e.g. by 3D reconstruction algorithms directly as the reference configuration of the soft body simulation. Therefore, we avoid to triangulate the object into a tetrahedral mesh. The computational speed of the grid discretization is comparable to existing methods on tetrahedral mesh. We furthermore demonstrate how the sparse point cost function can be used with observations from (simulated) RGB-D cameras to reconstruct the physical parameters of the simulated object.



# Contents

<b>Acknowledgements</b>	<b>vii</b>
<b>Abstract</b>	<b>ix</b>
<b>1. Introduction</b>	<b>1</b>
<b>2. Soft Body Simulation</b>	<b>3</b>
2.1. Basic Notation . . . . .	3
2.2. Strain and Stress . . . . .	3
2.3. PDE: Strong Form . . . . .	5
2.4. PDE: Weak Form . . . . .	6
2.4.1. Boundaries: Neumann . . . . .	7
2.4.2. Boundaries: Hardly Enforced Dirichlet . . . . .	7
2.4.3. Boundaries: Weakly Enforced Dirichlet (Nietsche) . . . . .	7
2.5. Discretization . . . . .	7
2.5.1. 2D Mesh . . . . .	9
2.5.2. 2D Grid . . . . .	11
2.6. Time Integration - Dynamic Elasticity . . . . .	22
2.7. Corotational Formulation . . . . .	24
2.8. Collision . . . . .	26
2.9. Conclusion . . . . .	30
<b>3. cuMat - Linear Algebra Library for CUDA</b>	<b>33</b>
3.1. Related Work . . . . .	34
3.2. A (very short) review of CUDA . . . . .	34
3.3. Batched Evaluation and the Matrix class . . . . .	35
3.4. Expression Templates and Kernel Merging . . . . .	36
3.5. Benchmarks . . . . .	37
3.6. Conclusion . . . . .	38
<b>4. 3D Implementation with CUDA</b>	<b>41</b>
4.1. Related Work . . . . .	41
4.2. Basis Functions and Partial Integrals . . . . .	41
4.3. Datastructures and Sparsity Pattern . . . . .	42
4.3.1. World Grid . . . . .	43
4.3.2. Input Data per Node and Element . . . . .	43
	<b>xi</b>

4.3.3.	Blocked CSR for the Stiffness Matrix . . . . .	43
4.4.	Matrix Assembly and Solving . . . . .	45
4.4.1.	Stiffness Matrix . . . . .	45
4.4.2.	Mass and Force Vector . . . . .	48
4.4.3.	Newmark Time Integration . . . . .	50
4.4.4.	Conjugate Gradient Solver . . . . .	51
4.4.5.	Levelset Advection . . . . .	51
4.5.	Rendering . . . . .	51
4.5.1.	Marching Cubes . . . . .	52
4.5.2.	Slices . . . . .	52
4.5.3.	Volume Raycasting . . . . .	53
4.6.	Benchmarks . . . . .	53
<b>5.</b>	<b>Inverse Simulation</b> . . . . .	<b>57</b>
5.1.	Related Work . . . . .	57
5.2.	General Adjoint Method . . . . .	57
5.2.1.	Problem Statement . . . . .	58
5.2.2.	Gradient Evaluation . . . . .	58
5.2.3.	Computing the Adjoint of the Problem . . . . .	59
5.2.4.	Examples . . . . .	62
5.2.5.	On Testing Adjoint Code . . . . .	64
5.3.	Adjoint Method for Soft Body Simulations . . . . .	66
5.4.	Cost Function . . . . .	66
5.4.1.	Differences on the Displacements . . . . .	67
5.4.2.	Differences on the Levelset . . . . .	67
5.4.3.	Distance to Point Clouds . . . . .	68
5.5.	Adjoint of the Individual Steps . . . . .	71
5.5.1.	Adjoint: Levelset Advection . . . . .	72
5.5.2.	Adjoint: Displacement Diffusion . . . . .	73
5.5.3.	Adjoint: Newmark Time Integration . . . . .	73
5.5.4.	Adjoint: Stiffness Matrix . . . . .	74
5.5.5.	Adjoint: Collision Forces . . . . .	76
5.5.6.	Adjoint: Body Forces . . . . .	78
5.6.	Optimization Algorithm . . . . .	78
5.7.	Memory Consumption . . . . .	78
5.8.	Analysis of the Gradients . . . . .	79
5.9.	Benchmarks and Examples . . . . .	83
5.10.	Stability . . . . .	88
<b>6.</b>	<b>Conclusion and Future Work</b> . . . . .	<b>91</b>



<b>Appendix</b>	<b>93</b>
<b>A. Additional Algorithms</b>	<b>93</b>
<b>B. Tables</b>	<b>97</b>
<b>C. Additional Equations</b>	<b>99</b>
<b>Bibliography</b>	<b>101</b>



# 1. Introduction

The field of elasticity or soft body simulation tries to model how objects deform under external and internal forces as physically accurate as possible. The physical properties of the objects range from stiff materials, which do not bend much, to materials that behave more like viscous fluids. This range gives rise to many algorithms tailored to specific material properties. For completely stiff materials (e.g. idealized wood, stone or steel in computer games), rigid body simulations are used [5, 35]. The other extrema, viscous materials (e.g. honey or wax) are most often better solved with a fluid solver based on the Navier-Stokes equations combined with a viscosity solver [12]. The middle range of elasticity deals with materials like rubber, flesh or tissue. Simulating this class of materials is covered by this thesis.

For the soft body simulation of medium stiff materials as listed above, different algorithms were developed for different targets. Simple mass-spring systems [73] approximate the model fairly well and their performance makes them applicable for real-time scenarios like computer games. However, they are not physically accurate. For physically accurate simulations, one typically uses the Finite Element Method (FEM). The standard approach is to discretize the object into tetrahedra [72], less frequently into cuboids [25], and assume that the boundary of the object is properly represented by the boundaries of the finite elements. This approach is used frequently in scientific simulations and the filming industry. In recent years, methods that allow the boundary of the object to lie inside the finite elements were developed under various names: exFEM, Immersed Boundaries, cutFEM, to name a few. In computer science, these methods are primarily used on tetrahedra meshes to simulate cuts in the mesh. In this thesis, we introduce a new discretization scheme for the elasticity simulation on a regular grid where the boundary is implicitly defined by a Signed Distance Function (SDF) and is allowed to lie within a grid cell. This method was already used successfully for fluid simulations in mechanical engineering. To our best knowledge, this method hasn't been applied to soft body simulations in the computer graphic community.

Furthermore, in many applications, the exact material properties and forces are not known. To overcome this difficulty, inverse problems try to reconstruct these properties from given input, e.g. camera observations, CT scans in medical applications, or keyframes for computer animation. We show how the new discretization technique on a regular grid can be used in the inverse problem using the Adjoint Method to estimate the material properties and external forces. We also describe a cost function that acts directly on sparse noisy observations of the target surface, as obtained e.g. by RGB-D cameras.

The major contributions of this thesis are:

- A new discretization scheme for soft body FEM simulations based on a regular grid with implicit object boundaries represented by a signed distance function (SDF)
- An efficient algorithm to parallelize the matrix assembly for the soft body simulation on the GPU with CUDA
- A demonstration on how this new discretization scheme can be used in the inverse problem using the Adjoint Method
- A cost function for the inverse problem that directly uses the sparse observations from RGB-D cameras as input.

The thesis is structured as follow: First, in Section 2 we introduce the mathematical background of soft body simulations and present the discretization into a standard tetrahedra-mesh and the new regular grid in 2D. Section 3 introduces *cuMat*, a linear algebra library for CUDA that is used in Section 4, the efficient implementation of the soft body simulation in 3D on the GPU. The inverse problem using the Adjoint Method is described in Section 5. Last, we provide results and comparisons in Section 6.

## 2. Soft Body Simulation

In this part we introduce the equations for a Soft Body / Elasticity simulation using the Finite Element Method as presented by Dick [25] and describe the discretization on a triangular mesh and on a regular grid. For simplicity, the discretizations are performed in 2D. The changes required in 3D and an efficient implementation on the GPU is described in Sec. 4. For a detailed introduction into the elasticity theory, we refer to the work of Dick [25].

### 2.1. Basic Notation

Notational conventions:

- Matrices: uppercase letters, e.g.  $A, F$ .
- Identity matrix:  $\mathbf{1}^d$  for the d-dimension identity. The dimension is dropped if it is clear from the context.
- Block access:  $A(i : j, m : n)$  denotes the sub-matrix of rows  $i$  to  $j$  and columns  $m$  to  $n$ . As a special case, a single subscript  $A_j$  denotes the j-th column.
- Component-wise product: The operator  $\odot$  is used to indicate the component-wise product, also called Hadamard or Schur product on matrices. If a division  $/$  is used on matrix expressions, it is always the component-wise division.
- Vectors: by default always column vectors, lowercase bold letters, e.g.  $\mathbf{f}, \mathbf{x}$ .
- Scalars: lowercase greek or roman letters, e.g.  $\alpha, \beta, i, j$ .
- Sets: uppercase greek letters, e.g.  $\Omega, \Gamma$ .
- Functions: lowercase roman letters, e.g.  $u : \Omega \rightarrow \mathbb{R}^d$ .

### 2.2. Strain and Stress

Assume that the object that should be simulated is given in the *reference configuration*  $\Omega^r$ . The reference configuration is the undeformed state of the object, i.e. no forces act on the object. Then,  $u : \Omega^r \rightarrow \Omega^d$  describes the *displacement vector* from the reference configuration

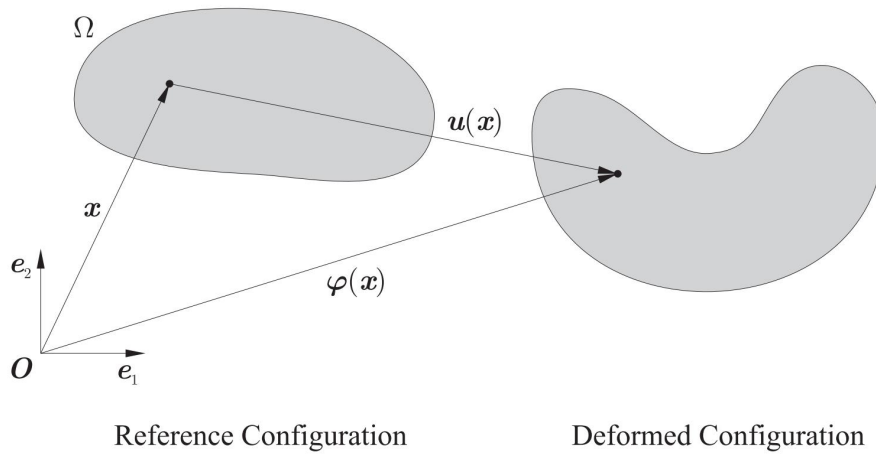


Figure 2.1.: Deformation from the reference configuration to the deformed configuration with the deformation  $u$ . Highlighted is a single position  $x$ . [25]

into the *deformed configuration*  $\Omega^d$ , see Fig. 2.1. The goal of elasticity simulations is to find this displacement vector from given external and internal forces. It holds that

$$\Omega^d = \{\mathbf{x} + u(\mathbf{x}) : \mathbf{x} \in \Omega^r\}. \quad (2.2.1)$$

We further use the *deformation*  $\varphi$  defined as

$$\varphi(\mathbf{x}) := \mathbf{x} + u(\mathbf{x}). \quad (2.2.2)$$

The deformation gradient is defined as

$$F := \nabla\varphi = \nabla u + \mathbf{1}. \quad (2.2.3)$$

Note that the gradient of a vector-valued function (the Jacobian) is the matrix:

$$\nabla\varphi := \begin{pmatrix} \frac{\partial\varphi_1}{\partial x_1} & \dots & \frac{\partial\varphi_1}{\partial x_d} \\ \vdots & \ddots & \vdots \\ \frac{\partial\varphi_d}{\partial x_1} & \dots & \frac{\partial\varphi_d}{\partial x_d} \end{pmatrix}. \quad (2.2.4)$$

From the deformation gradient, we derive the strain tensor field that describes the material's local changes of shape

$$E(u) := \frac{1}{2} (\nabla u + (\nabla u)^T + (\nabla u)^T \nabla u) \in \mathbb{R}^{d \times d}. \quad (2.2.5)$$

This strain tensor is called *St. Venant-Kirchoff strain*. Note that  $E(\cdot)$  is a non-linear function. For simplicity, one typically uses the *Green strain tensor*

$$\epsilon(u) := \frac{1}{2} (\nabla u + (\nabla u)^T) \in \mathbb{R}^{d \times d} \quad (2.2.6)$$

by dropping the nonlinear terms.

The stress tensor describes the internal forces depending on the strain acting on the body. Assuming a hyperelastic material, the *Piola-Kirchoff stress tensor* is given by

$$P(u) := 2\mu E(u) + \lambda \operatorname{tr}(E(u)) \mathbf{1}. \quad (2.2.7)$$

In order to obtain a linear version, we can use  $\epsilon(u)$  instead of  $E(u)$ . For a detailed derivations of the elastic material, see [72] and [25].

The two constants  $\mu$  and  $\lambda$  are called the *Lamé coefficients* and are derived from the following two material properties:

- Young's modulus  $k > 0$ : the higher the value, the stiffer the material.
- Poisson's ratio  $\rho \in (0, \frac{1}{2})$ : the higher, the more incompressible is the material. Typically, values between 0.3 and 0.45 are used.

The Lamé coefficients are defined as:

$$\mu := \frac{k}{2(1 + \rho)}, \quad (2.2.8a)$$

$$\lambda := \frac{k\rho}{(1 + \rho)(1 - 2\rho)}. \quad (2.2.8b)$$

## 2.3. PDE: Strong Form

We can now define the strong form of the PDE describing the soft body physics.

Let  $f_B$  be the body forces within the object (e.g. the gravity),  $\Gamma_D^r$  the Dirichlet boundaries,  $\Gamma_N^r$  the Neumann boundaries (such that  $\Gamma_D^r \cup \Gamma_N^r = \partial\Omega$ ),  $u_D : \Gamma_D^r \rightarrow \mathbb{R}^d$  the fixed displacements at the Dirichlet boundaries, and  $f_S : \Gamma_N^r \rightarrow \mathbb{R}^d$  boundary forces at the Neumann boundaries. Then the static elasticity problem is given by: Find  $u : \bar{\Omega}^r \rightarrow \mathbb{R}^d$  such that

$$-\operatorname{div} P(u) = f_B \text{ in } \Omega^r \quad (2.3.1a)$$

$$u = u_D \text{ on } \Gamma_D^r \quad (2.3.1b)$$

$$P(u) \cdot \mathbf{n} = f_S \text{ in } \Gamma_N^r. \quad (2.3.1c)$$

Note that  $P(u)$  gives a matrix, hence  $P(u) \cdot \mathbf{n}$  is the extension of the dot product to matrices by performing the dot product column-wise. The result is a vector in  $\mathbb{R}^d$ .

This model can be simply extended to the dynamic elasticity problem: Find  $u : \Omega^r \times \mathbb{R}_0^+ \rightarrow \mathbb{R}^d$  such that

$$m\ddot{u} - \operatorname{div} P(u) = f_B \quad \text{in } \Omega^r \times \mathbb{R}_0^+ \quad (2.3.2a)$$

$$u = u_D \quad \text{on } \Gamma_D^r \times \mathbb{R}_0^+ \quad (2.3.2b)$$

$$P(u) \cdot \mathbf{n} = f_S \quad \text{in } \Gamma_N^r \times \mathbb{R}_0^+ \quad (2.3.2c)$$

$$u = u^0 \quad \text{in } \Omega^r \times \{0\} \quad (2.3.2d)$$

$$\dot{u} = \dot{u}^0 \quad \text{in } \Omega^r \times \{0\}, \quad (2.3.2e)$$

where  $m : \Omega^r \rightarrow \mathbb{R}^+$  is the mass at that position,  $u^0$  and  $\dot{u}^0$  is the initial displacements and velocities, and  $\ddot{u}$  is the second time derivative of the current displacement (the acceleration).

## 2.4. PDE: Weak Form

We will now derive the weak form of the static elasticity problem (2.3.1). The term  $m\ddot{u}$  from the dynamic elasticity problem does not contain any derivatives and can therefore be trivially added in later stages when solving the resulting linear system.

Let  $V := H^1(\bar{\Omega}^r \rightarrow \mathbb{R}^d)$  be the space of test functions and let  $U := H^1(\bar{\Omega}^r \rightarrow \mathbb{R}^d)$  be the space of trial functions. Multiplying the test function gives: Find  $u \in U$  such that  $\forall v \in V$

$$\int_{\Omega} -\operatorname{div} P(u) \cdot v \, dx = \int_{\Omega} f_B \cdot v \, dx \quad \text{in } \Omega^r \quad (2.4.1a)$$

$$u = u^0 \quad \text{on } \Gamma_D^r \quad (2.4.1b)$$

$$P(u) \cdot \mathbf{n} = f_S \quad \text{in } \Gamma_N^r. \quad (2.4.1c)$$

We now assume that the linear Green strain  $\epsilon(u) := \frac{1}{2} (\nabla u + (\nabla u)^T) \in \mathbb{R}^{d \times d}$  given in Eq. (2.2.6) is used. This does not allow large rotations, but this can be corrected with the help of the corotational formulation, see Sec. 2.7.

Starting from the right hand side, we obtain (using the generalized Divergence Theorem on matrix fields):

$$\begin{aligned} \int_{\Omega} f_B \cdot v \, dx &= \int_{\Omega} -\operatorname{div} P(u) \cdot v \, dx \\ &= \int_{\Omega} P(u) : \nabla v \, dx - \int_{\partial\Omega} P(u) \cdot \mathbf{n} \cdot v \, ds \\ &= \int_{\Omega} \sum_{j=1}^d \left( \mu \left( \nabla u_j + \frac{\partial}{\partial x_j} u \right) + \lambda \sum_{i=1}^d \frac{\partial u_i}{\partial u_j} \mathbf{1}_j \right) \cdot \nabla v_j \, dx \\ &\quad - \int_{\partial\Omega} \sum_{j=1}^d \left( \left( \mu \left( \nabla u_j + \frac{\partial}{\partial x_j} u \right) + \lambda \sum_{i=1}^d \frac{\partial u_i}{\partial u_j} \mathbf{1}_j \right) \cdot \mathbf{n} \right) v_j \, ds. \end{aligned} \quad (2.4.2)$$



This formulation does not include boundary conditions yet, they are handled now. In general, the boundaries are split between Dirichlet and Neumann boundaries:

$$\int_{\partial\Omega} P(u) \cdot \mathbf{n} \cdot v \, ds = \int_{\Gamma_D^r} P(u) \cdot \mathbf{n} \cdot v \, ds + \int_{\Gamma_N^r} P(u) \cdot \mathbf{n} \cdot v \, ds. \quad (2.4.3)$$

#### 2.4.1. Boundaries: Neumann

Neumann boundaries are defined as  $P(u) \cdot \mathbf{n} = f_S$  in  $\Gamma_N^r$ , see (2.3.1c). Hence

$$\int_{\Gamma_N^r} P(u) \cdot \mathbf{n} \cdot v \, ds = \int_{\Gamma_N^r} f_S \cdot v \, ds. \quad (2.4.4)$$

#### 2.4.2. Boundaries: Hardly Enforced Dirichlet

Dirichlet boundaries are defined as  $u = u_D$  on  $\Gamma_D^r$ , see (2.3.1b). The simplest way is to enforce these boundaries strongly. They are directly included into the function spaces. The new test space is  $V_0 := \{v \in V : v = \mathbf{0} \text{ on } \Gamma_D^r\}$  and the new trial space is  $U_0 := \{u \in U : u = u_D \text{ on } \Gamma_D^r\}$ .

These boundaries are only applicable later in the discretization (Sec. 2.5) if the nodes coincide exactly with the Dirichlet boundaries. Then degrees of freedom of those nodes are removed, the matrix shrinks.

If this is not possible, e.g. as in the cutFEM method, Sec. 2.5.2, weak Dirichlet boundaries have to be used.

#### 2.4.3. Boundaries: Weakly Enforced Dirichlet (Nietsche)

Dirichlet boundaries can also be enforced weakly using Nietsche's method [8, 27, 44]. The weak form is extended using productive zeros  $u - u_0$ :

$$- \int_{\Gamma_N^r} P(u) \cdot \mathbf{n} \cdot v \, ds - \int_{\Gamma_N^r} P(v) \cdot \mathbf{n} \cdot (u - u_0) \, ds - \eta \int_{\Gamma_N^r} (u - u_0) \cdot v \, ds. \quad (2.4.5)$$

The first term makes the resulting linear system symmetric. The second term enforces the Dirichlet boundaries. The third term acts as a regularizer and the parameter  $\eta$  has to be chosen as  $\eta > ch^{-1}$  with  $h$  being the grid size later in the discretization and  $c$  a sufficient large constant. In our experiments, we chose  $10^5$  for stable results.

## 2.5. Discretization

The next step is to discretize the weak form. We provide and compare two different discretizations, a triangle mesh that moves with the object and a regular fixed grid with the

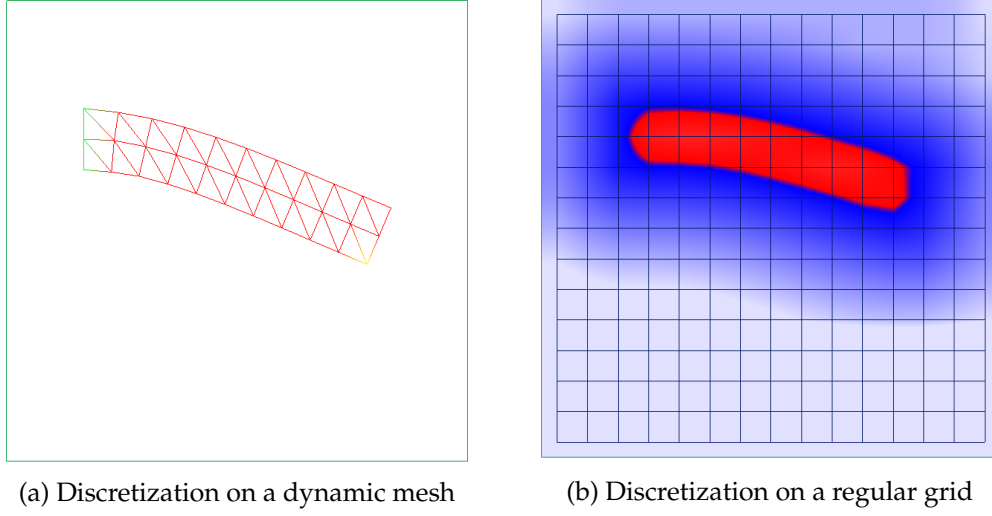


Figure 2.2.: Different representations / discretizations of the same object

object represented as a levelset. For simplicity, we assume  $d = 2$ , the 3D version is presented later in Sec. 4. A comparison of the two different methods, mesh and grid, can be seen in Fig. 2.2.

We are going to use the following notation:

- $N_v$  is the number of nodes / degrees of freedom.
- $N_e$  is the number of elements (triangles or quads).
- $N_{ev}$  is the number of nodes per element (3 for triangles, 4 for quads).
- $\Omega^e \subset \mathbb{R}^d$  is the area covered by element  $e$ .
- $\phi_i^e : \mathbb{R}^d \rightarrow \mathbb{R}^d$  is the linear local basis of node  $i$  in element  $e$ .
- $\mathbf{u}_i \in \mathbb{R}^d$  is the value of node  $i$ ;  $\mathbf{f}_{B,i} \in \mathbb{R}^d$  is the body force at node  $i$ .
- $\underline{\mathbf{u}} = (\mathbf{u}_{x,1}, \mathbf{u}_{y,1}, \dots, \mathbf{u}_{x,N_v}, \mathbf{u}_{y,N_v})$  is the linearized version of  $\mathbf{u}$ , similarly  $\underline{\mathbf{f}}$ .
- $s(e, i) \in \{1, \dots, N_v\}$  is the mapping of the local index  $i \in \{1, \dots, N_{ev}\}$  in element  $e$  to the global index.
- $V_h := \left\{ v_h \in V : v_h|_{\Omega^e}(x) = \sum_{i=1}^{N_{ev}} v_{s(e,i)} \phi_i^e(x) \right\}$  is the discrete function space of  $V$ .

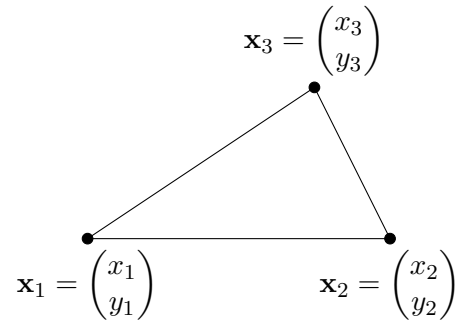


Figure 2.3.: Local coordinates of a triangle

### 2.5.1. 2D Mesh

A 2D mesh is defined by the positions of  $N_v$  vertices  $\mathbf{x}_1, \dots, \mathbf{x}_{N_v}$  combined to  $N_e$  triangles by the mapping  $s(e, i)$ . The local coordinates of a triangle  $e$  are given by  $\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3$  as visualized in Fig. 2.3.

The local (linear) basis functions are defined using barycentric coordinates: Let

$$\text{area}(\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3) = \frac{1}{2}(x_2 - x_1)(y_3 - y_1) - (x_3 - x_1)(y_2 - y_1) \quad (2.5.1)$$

be the signed area of the triangle  $\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3$ . Let

$$\phi_1^e(\mathbf{x}) = \frac{\text{area}(\mathbf{x}, \mathbf{x}_2, \mathbf{x}_3)}{\text{area}(\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3)}, \quad (2.5.2a)$$

$$\phi_2^e(\mathbf{x}) = \frac{\text{area}(\mathbf{x}_1, \mathbf{x}, \mathbf{x}_3)}{\text{area}(\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3)}, \quad (2.5.2b)$$

$$\phi_3^e(\mathbf{x}) = \frac{\text{area}(\mathbf{x}_1, \mathbf{x}_2, \mathbf{x})}{\text{area}(\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3)} \quad (2.5.2c)$$

be the local basis functions of the triangle. Note that we don't need to use absolute values to get the actual areas of the triangles, because the winding orders in the nominator and denominator are consistent. These basis functions are assembled into the basis matrix  $\Phi^e$

$$\Phi^e(\mathbf{x}) := \begin{pmatrix} \phi_1^e(\mathbf{x}) & 0 & \phi_2^e(\mathbf{x}) & 0 & \phi_3^e(\mathbf{x}) & 0 \\ 0 & \phi_1^e(\mathbf{x}) & 0 & \phi_2^e(\mathbf{x}) & 0 & \phi_3^e(\mathbf{x}) \end{pmatrix} \in \mathbb{R}^{2 \times 6} \quad (2.5.3)$$

such that  $u_h|_{\Omega_e}(\mathbf{x}) = \Phi^e(\mathbf{x})\underline{\mathbf{u}}^e$  with  $\underline{\mathbf{u}}^e = \{u_{s(e,1)}^e{}^T; u_{s(e,2)}^e{}^T; u_{s(e,3)}^e{}^T\} \in \mathbb{R}^6$ .

## 2. Soft Body Simulation

---

The derivatives are assembled in the matrix  $B^e$

$$\begin{aligned}
 B^e &= \begin{pmatrix} \frac{\partial \phi_1^e(\mathbf{x})}{\partial x_1} & 0 & \frac{\partial \phi_2^e(\mathbf{x})}{\partial x_1} & 0 & \frac{\partial \phi_3^e(\mathbf{x})}{\partial x_1} & 0 \\ 0 & \frac{\partial \phi_1^e(\mathbf{x})}{\partial x_2} & 0 & \frac{\partial \phi_2^e(\mathbf{x})}{\partial x_2} & 0 & \frac{\partial \phi_3^e(\mathbf{x})}{\partial x_2} \\ \frac{\partial \phi_1^e(\mathbf{x})}{\partial x_2} & \frac{\partial \phi_1^e(\mathbf{x})}{\partial x_1} & \frac{\partial \phi_2^e(\mathbf{x})}{\partial x_2} & \frac{\partial \phi_2^e(\mathbf{x})}{\partial x_1} & \frac{\partial \phi_3^e(\mathbf{x})}{\partial x_2} & \frac{\partial \phi_3^e(\mathbf{x})}{\partial x_1} \end{pmatrix} \\
 &= \frac{1}{\text{area}(\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3)} \begin{pmatrix} y_2 - y_3 & 0 & y_3 - y_1 & 0 & y_1 - y_2 & 0 \\ 0 & x_3 - x_2 & 0 & x_1 - x_3 & 0 & x_2 - x_1 \\ x_3 - x_2 & y_2 - y_3 & x_1 - x_3 & y_3 - y_1 & x_2 - x_1 & y_1 - y_2 \end{pmatrix}.
 \end{aligned} \tag{2.5.4}$$

Note that the matrix  $B^e$  does not depend on  $\mathbf{x}$  unlike  $\Phi^e(\mathbf{x})$ .

The integrals in the weak form (2.4.2) can be decomposed into per-element integrals. This gives rise to the following matrix representation:

$$\begin{aligned}
 &\int_{\Omega^e} \sum_{j=1}^d \left( \mu \left( \nabla u_j + \frac{\partial}{\partial x_j} u \right) + \lambda \sum_{i=1}^d \frac{\partial u_i}{\partial u_j} \mathbf{1}_j \right) \cdot \nabla v_j \, dx \\
 &= \int_{\Omega^e} \begin{pmatrix} \frac{\partial u_1}{\partial x_1} \\ \frac{\partial u_1}{\partial x_2} \\ \frac{\partial u_1}{\partial x_2} + \frac{\partial u_2}{\partial x_1} \end{pmatrix} \underline{\mathbf{v}}^e \cdot \begin{pmatrix} 2\mu + \lambda & \lambda & \lambda \\ \lambda & 2\mu + \lambda & \lambda \\ \mu & & \mu \end{pmatrix} \begin{pmatrix} \frac{\partial u_1}{\partial x_1} \\ \frac{\partial u_1}{\partial x_2} \\ \frac{\partial u_1}{\partial x_2} + \frac{\partial u_2}{\partial x_1} \end{pmatrix} \underline{\mathbf{u}}^e \, dx \\
 &= (\underline{\mathbf{v}}^e)^T \int_{\Omega^e} (B^e)^T C B^e \, dx \, \underline{\mathbf{u}}^e \\
 &= (\underline{\mathbf{v}}^e)^T \underbrace{(|\text{area}(\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3)|) (B^e)^T C B^e}_{=: K^e \in \mathbb{R}^{6 \times 6} = \mathbb{R}^{d_{\text{Neu}} \times d_{\text{Neu}}}} \underline{\mathbf{u}}^e.
 \end{aligned} \tag{2.5.5}$$

The matrix  $C$  is called the *material matrix* and contains the Lamé coefficients as introduced in Eq. (2.2.8). The matrix  $K^e$  is the *stiffness matrix* of element  $e$ .

The inertia term (see Eq. (2.3.2)) is discretized as

$$\begin{aligned}
 \int_{\Omega^e} m \ddot{\mathbf{u}} \cdot \mathbf{v} \, dx &= \int_{\Omega^e} m (\Phi^e(x) \underline{\mathbf{v}}^e)^T \Phi^e(x) \ddot{\mathbf{u}} \, dx \\
 &= (\underline{\mathbf{v}}^e)^T \int_{\Omega^e} m (\Phi^e(x))^T \Phi^e(x) \, dx \, \ddot{\mathbf{u}} \\
 &= (\underline{\mathbf{v}}^e)^T M^e \ddot{\mathbf{u}}.
 \end{aligned} \tag{2.5.6}$$

The computation of the *mass matrix*  $M^e$  can be simplified by assuming lumped mass, i.e. the whole mass of the triangle is centered at the vertices. The mass matrix then becomes diagonal:

$$M^e := \frac{m}{3 * \text{area}(\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3)} \mathbf{1}_6 \in \mathbb{R}^{6 \times 6}. \tag{2.5.7}$$

Neumann boundaries are discretized using

$$\int_{\Gamma_N^e} \mathbf{v} \cdot \mathbf{f}_N \, dx = (\underline{\mathbf{v}}^e)^T \int_{\Gamma_N^e} (\Phi^e(x))^T \mathbf{f}_N \, dx. \tag{2.5.8}$$

If we assume that the Neumann forces act only on the vertices, the computation can be simplified as

$$\int_{\Gamma_N^e} v \cdot f_N \, dx = (\underline{v}^e)^T \underline{\mathbf{f}}_N^e \, dx. \quad (2.5.9)$$

with  $\underline{\mathbf{f}}_N^e = (l_1 f_{Nx1}, l_1 f_{Ny1}, l_2 f_{Nx2}, l_2 f_{Ny2}, l_3 f_{Nx1}, l_3 f_{Ny1})^T$  where  $l_1, l_2, l_3$  are the lengths of the edges incident to vertices 1,2,3 that lie on the boundary of the object.

All the local element matrices and vectors  $K^e, M^e, \underline{\mathbf{f}}_B^e, \underline{\mathbf{f}}_N^e$  are finally assembled into global matrices  $K, M \in \mathbb{R}^{2N_v \times 2N_v}, \underline{\mathbf{f}}_B, \underline{\mathbf{f}}_N \in \mathbb{R}^{2N_v}$  (see Alg. A.1 in the Appendix). Because it is sufficient to test the weak form only for all basis functions of  $v$ , this gives rise to a linear system of  $2N_v$  equations

$$K \underline{\mathbf{u}} = \underline{\mathbf{f}}_B - \underline{\mathbf{f}}_N \quad (2.5.10)$$

for the static case and

$$M \ddot{\underline{\mathbf{u}}} + K \underline{\mathbf{u}} = \underline{\mathbf{f}}_B - \underline{\mathbf{f}}_N \quad (2.5.11)$$

for the dynamic case. Observe that this system is symmetric and positive definite. The dynamic case needs a time integration scheme that is described in detail in Sec. 2.6.

Up to now, the static case can not be solved as the matrix  $K$  is not of full rank. Each matrix  $K^e$  has rank 3 and the global matrix  $K$  has also only rank  $2N_v - 3$ . These three degrees of freedom correspond exactly to the three degrees of freedom in affine transformations: translation along  $x$ , translation along  $y$ , rotation. It is therefore necessary to fix at least two vertices to specific coordinates using Dirichlet boundaries  $u = u_D$ . This is done by plugging in the fixed value  $u_D$  for every Dirichlet node into the linear system  $K \underline{\mathbf{u}} = \underline{\mathbf{f}}$ , thus reducing the size of  $K$  and  $\underline{\mathbf{f}}$  by two (rows and columns in  $K$ ).

In the computer graphics literature, the above matrices are typically used directly in the simulation, without considering the underlying PDE. See e.g. [77] for a work that uses these matrices and extend them with locomotion terms.

## 2.5.2. 2D Grid

We now apply the ideas from the discretization on the mesh in a new setting, the discretization on a regular grid. The grid cell boundaries don't need to coincide with the object boundaries any longer, as it was the case with the tetrahedra mesh. The individual steps, however, require a bit more computation.

### 2.5.2.1. Levelset representation

In the case of a static grid, the object in its reference configuration  $\Omega^r$  (and all other configurations) is enclosed within a larger rectangular region  $\Omega$ .

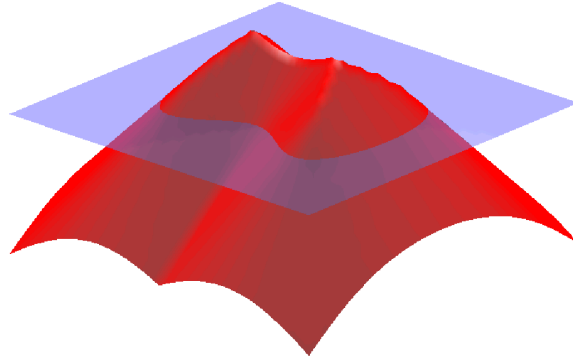


Figure 2.4.: Example of a signed distance function<sup>1</sup>

With a slight abuse of notation, an object  $X \subset \Omega$  (e.g.  $\Omega^r$ ) is implicitly described using a signed distance function (SDF)  $\phi : \mathbb{R}^d \rightarrow \mathbb{R}$  such that

$$\phi(\mathbf{x}) := \begin{cases} -d(\mathbf{x}, \partial X) & \mathbf{x} \in X \\ d(\mathbf{x}, \partial X) & \mathbf{x} \notin \bar{X} \\ 0 & \mathbf{x} \in \partial X \end{cases} \quad (2.5.12)$$

where  $d(\mathbf{x}, \partial X)$  is the shortest distance from  $\mathbf{x}$  to the boundary of  $X$  in the Euclidean norm. The signed distance function is negative inside the object, positive outside and zero on the boundary. Note the abuse of notation:  $\phi$  now denotes the level set function, no longer the basis functions. Local basis functions are now called  $N_i$ . An example of a SDF is visualization in Fig. 2.4.

For every SDF, it holds that  $|\nabla\phi| = 1$  where  $\nabla\phi$  is the outward normal along  $\partial X$ .

In the discretization, the values of  $\phi$  are given at every grid point  $(x, y)$  and then bilinearly interpolated. An example of this representation was already shown in Fig. 2.2b.

Notation:

- Every discrete grid point where the SDF is stored is called a *node*.
- Nodes that are inside the object  $\phi \leq 0$  are called *inner nodes*.
- Cells in the grid that have at least one inner node as vertex are called *active cells*.
- Nodes that are outside the object but incident to an active cell are called *boundary nodes*.
- Nodes that are either inner nodes or boundary nodes are called *active nodes*.
- Nodes that are not active nodes are called *outer nodes*.
- Cells that are not active cells are called *outer cells*.

---

<sup>1</sup>[https://upload.wikimedia.org/wikipedia/commons/d/d4/Signed\\_distance2.png](https://upload.wikimedia.org/wikipedia/commons/d/d4/Signed_distance2.png)

### 2.5.2.2. Basis functions

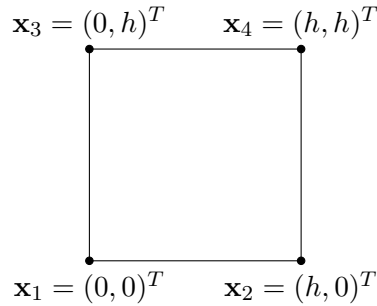


Figure 2.5.: Local coordinates of the standard quad

Let  $h$  be the side length of each cell in the regular grid. For the computation of the basis functions, we can assume that the cell starts at the origin as shown in Fig. 2.5.

The basis functions and their derivatives are given by

$$N_1(\mathbf{x}) = \left(1 - \frac{x}{h}\right) \left(1 - \frac{y}{h}\right), \quad \frac{\partial N_1}{\partial x}(\mathbf{x}) = \frac{y-h}{h^2}, \quad \frac{\partial N_1}{\partial y}(\mathbf{x}) = \frac{x-h}{h^2}, \quad (2.5.13a)$$

$$N_2(\mathbf{x}) = \frac{x}{h} \left(1 - \frac{y}{h}\right), \quad \frac{\partial N_2}{\partial x}(\mathbf{x}) = \frac{h-y}{h^2}, \quad \frac{\partial N_2}{\partial y}(\mathbf{x}) = \frac{-x}{h^2}, \quad (2.5.13b)$$

$$N_3(\mathbf{x}) = \left(1 - \frac{x}{h}\right) \frac{y}{h}, \quad \frac{\partial N_3}{\partial x}(\mathbf{x}) = \frac{-y}{h^2}, \quad \frac{\partial N_3}{\partial y}(\mathbf{x}) = \frac{h-x}{h^2}, \quad (2.5.13c)$$

$$N_4(\mathbf{x}) = \frac{xy}{h^2}, \quad \frac{\partial N_4}{\partial x}(\mathbf{x}) = \frac{y}{h^2}, \quad \frac{\partial N_4}{\partial y}(\mathbf{x}) = \frac{x}{h^2}. \quad (2.5.13d)$$

Analogously to the triangle basis in Eq. (2.5.3) and Eq. (2.5.4), these basis functions are combined into the matrices  $\Phi^e(\mathbf{x}) \in \mathbb{R}^{2 \times 8}$  and  $B^e(\mathbf{x}) \in \mathbb{R}^{3 \times 8}$  that now contain 8 columns instead of 6 because we now have four vertices per element instead of three. Further note that the derivatives are now dependent of the position  $\mathbf{x}$ . This was not the case for the triangles as shown in Sec. 2.5.1.

### 2.5.2.3. Integration over parts of a cell

Since the object does not coincide with the cell boundaries, the integrals in the elasticity simulation (see Eq. (2.4.1)) have to be evaluated over a part of each cell. This method is called cutFEM in the math community [39, 15, 14, 38] or Immersed Boundary Methods in the engineering community [52]. To our knowledge, this method has so far being used for Navier-Stokes fluid simulations (citations above), but has not yet been attempted for soft body elasticity.

Table 2.1 summarizes the five base cases for the integration. All other cases of the in total  $2^4 = 16$  possibilities are rotations of those five base cases.

## 2. Soft Body Simulation

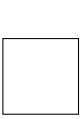

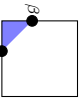

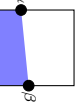

			Let $v_i$ be the value to integrate over $\Omega^e$ and $\phi_i$ be the SDF value at node $i$ . $\Omega^e$ is the part of the cell that is within the object described by the SDF	
Configuration	Area integral $v := \int_{\Omega^e} \sum_{i=1}^4 N_i(x) v_i \, dx$	Boundary integral $v := \int_{\partial\Omega^e} \sum_{i=1}^4 N_i(x) v_i \, ds$		
$\text{all}(\phi_i) > 0$	 $v = 0$	$v = 0$		
$\text{all}(\phi_i) \leq 0$	 $v = \frac{h^2}{4} (v_1 + v_2 + v_3 + v_4)$	$v = hc$ $c = \mathbf{1}_{\phi_1=0 \wedge \phi_2=0} + \mathbf{1}_{\phi_1=0 \wedge \phi_3=0} + \mathbf{1}_{\phi_2=0 \wedge \phi_4=0} + \mathbf{1}_{\phi_3=0 \wedge \phi_4=0}$		
$\phi_1 \leq 0 \wedge \phi_2 > 0$ $\wedge \phi_3 > 0 \wedge \phi_4 > 0$	 $\alpha = \phi_1 / (\phi_1 - \phi_2)$ $\beta = \phi_1 / (\phi_1 - \phi_3)$ $c_1 = (\alpha\beta(\alpha(\beta - 4) - 4(\beta - 3))) / 24$ $c_2 = -(\alpha^2\beta(\beta - 4)) / 24$ $c_3 = -(\alpha\beta^2(\alpha - 4)) / 24$ $c_4 = (\alpha^2\beta^2) / 24$ $v = h^2(c_1 v_1 + c_2 v_2 + c_3 v_3 + c_4 v_4)$	$\alpha = \phi_1 / (\phi_1 - \phi_2)$ $\beta = \phi_1 / (\phi_1 - \phi_3)$ $c_1 = (6 + \alpha(\beta - 3) - 3\beta) / 6$ $c_2 = -(\alpha(\beta - 3)) / 6$ $c_3 = -(\beta(\alpha - 3)) / 6$ $c_4 = \alpha\beta / 6$ $v = h\sqrt{\alpha^2 + \beta^2}(c_1 v_1 + c_2 v_2 + c_3 v_3 + c_4 v_4)$		
$\phi_1 > 00 \wedge \phi_2 \leq 0$ $\wedge \phi_3 \leq 0 \wedge \phi_4 \leq 0$	 $\alpha = \phi_1 / (\phi_1 - \phi_2)$ $\beta = \phi_1 / (\phi_1 - \phi_3)$ $c_1 = 0.25 - (\alpha\beta(\alpha(\beta - 4) - 4(\beta - 3))) / 24$ $c_2 = 0.25 + (\alpha^2\beta(\beta - 4)) / 24$ $c_3 = 0.25 + (\alpha\beta^2(\alpha - 4)) / 24$ $c_4 = 0.25 - (\alpha^2\beta^2) / 24$ $v = h^2(c_1 v_1 + c_2 v_2 + c_3 v_3 + c_4 v_4)$	$\alpha = \phi_1 / (\phi_1 - \phi_2)$ $\beta = \phi_1 / (\phi_1 - \phi_3)$ $c_1 = (6 + \alpha(\beta - 3) - 3\beta) / 6$ $c_2 = -(\alpha(\beta - 3)) / 6$ $c_3 = -(\beta(\alpha - 3)) / 6$ $c_4 = \alpha\beta / 6$ $v = h\sqrt{\alpha^2 + \beta^2}(c_1 v_1 + c_2 v_2 + c_3 v_3 + c_4 v_4)$		
$\phi_1 \leq 0 \wedge \phi_2 \leq 0$ $\wedge \phi_3 > 0 \wedge \phi_4 > 0$	 $\alpha = \phi_1 / (\phi_1 - \phi_2)$ $\beta = \phi_2 / (\phi_2 - \phi_3)$ $c_1 = -(3\alpha^2 + 2\alpha(\beta - 4) + \beta(\beta - 4)) / 24$ $c_2 = (\beta(8 - 3\beta) - 2\alpha(\beta - 2) - \alpha^2) / 24$ $c_3 = (3\alpha^2 + 2\alpha\beta + \beta^2) / 24$ $c_4 = (\alpha^2 + 2\alpha\beta + 3\beta^2) / 24$ $v = h^2(c_1 v_1 + c_2 v_2 + c_3 v_3 + c_4 v_4)$	$\alpha = \phi_1 / (\phi_1 - \phi_2)$ $\beta = \phi_2 / (\phi_2 - \phi_3)$ $c_1 = (3 - \alpha - 2\alpha - \beta) / 6$ $c_2 = (3 - \alpha - 2\beta) / 6$ $c_3 = (2\alpha + \beta) / 6$ $c_4 = (\alpha + 2\beta) / 6$ $v = h\sqrt{\alpha^2 + \beta^2}(c_1 v_1 + c_2 v_2 + c_3 v_3 + c_4 v_4)$		
$\phi_1 \leq 0 \wedge \phi_2 > 0$ $\wedge \phi_3 > 0 \wedge \phi_4 \leq 0$		This configuration only occurs on low grid resolutions when the boundary is not properly resolved. It is hence not implemented.		

Table 2.1.: Integration over a part of a cell



The elasticity has to be evaluated only over active nodes. As one can see from Tab. 2.1, cells that have no inner nodes as vertices have a zero integral (outer cells).

Note that these quadrature schemes are not exact. Interpolation within a quad yields a quadratic formula (the basis functions are quadratic functions), hence the boundary is a parabola. In the numerical schemes from Table 2.1 we make the simplifying assumption of a linear boundary. This leads to a relative error of 3.8% or an absolute error of 1.7% on average. This error is small enough to don't visually influence the simulation results and are therefore neglected in the following.

#### 2.5.2.4. Matrix assembly

The parts of the Weak Form can now be written in matrix form similar to the mesh discretization (compare Eq. (2.5.5) to Eq. (2.5.9)). The assembly of the mass matrix, stiffness matrix and Neumann forces is almost the same as in the case of a mesh. The difference is that the per-element matrices are now of size  $8 \times 8$  because there are now four vertices involved instead of three. Furthermore, the integrals have to be evaluated now with the equations presented in Sec. 2.5.2.3.

$$\int_{\Omega^e} m \ddot{u} \cdot v \, dx = (\underline{\mathbf{v}}^e)^T \underbrace{\int_{\Omega^e} \Phi^e(x)^T m \Phi^e(x) \, dx}_{=: M^e \in \mathbb{R}^{8 \times 8}} \underline{\mathbf{u}}^e \quad (2.5.14)$$

$$\int_{\Omega^e} \mu \dots + \lambda \dots \, dx = (\underline{\mathbf{v}}^e)^T \underbrace{\int_{\Omega^e} B^e(x)^T C B^e(x) \, dx}_{=: K^e \in \mathbb{R}^{8 \times 8}} \underline{\mathbf{u}}^e \quad (2.5.15)$$

$$\int_{\Gamma_N^e} f_S \cdot v \, ds = (\underline{\mathbf{v}}^e)^T \underbrace{\int_{\Gamma_N^e} \Phi^e(x)^T \, dx}_{=: \mathbf{f}^e \in \mathbb{R}^8} \mathbf{f}_S. \quad (2.5.16)$$

Now that the object boundary does not necessarily coincide with the grid boundaries, hard Dirichlet boundaries (see Sec. 2.4.2) can't be used any longer. Instead, the Dirichlet boundaries have to be enforced weakly with Nietsche's method (see Sec. 2.4.3).

$$\begin{aligned}
& \int_{\Gamma_D^e} \sum_{j=1}^d (P(u)_j \cdot \mathbf{n}) v_j \, ds = \\
& \int_{\Gamma_D^e} \sum_{j=1}^d \left( (\mu(\nabla u + (\nabla u)^T) + \lambda \operatorname{tr}(\nabla u) \mathbf{1})_j \cdot \mathbf{n} \right) v_j \, ds = \\
& = \int_{\Gamma_D^e} \sum_{j=1}^d \left( \mu(\nabla u_j \cdot \mathbf{n} + \frac{\partial u}{\partial x_j} \cdot \mathbf{n}) + \lambda \mathbf{n}_j \sum_{i=1}^d \frac{\partial u_i}{\partial x_i} \right) v_j \, ds = \\
& = (\underline{\mathbf{v}}^e)^T \int_{\Gamma_D^e} \underbrace{\Phi^e(x)^T \begin{pmatrix} 2\mu \mathbf{n}_1 + \lambda \mathbf{n}_1 & \lambda \mathbf{n}_1 & \mu \mathbf{n}_2 \\ \lambda \mathbf{n}_2 & 2\mu \mathbf{n}_2 + \lambda \mathbf{n}_2 & \mu \mathbf{n}_1 \end{pmatrix}}_{=: D^e} B^e(x) \, ds \underline{\mathbf{u}}^e. \quad (2.5.17)
\end{aligned}$$

The matrix  $D^e$  can be seen as a variation of the stiffness matrix  $C$  that includes the surface normal. Therefore, it has to be recomputed for every element. Recall that the normal is simply given by  $\nabla \phi$ .

By symmetry, the same derivation from above is used for the other Nietsche terms  $\int_{\Gamma_D^e} \sum_{j=1}^d (P(v)_j \cdot \mathbf{n}) u_j \, ds$  and  $\int_{\Gamma_D^e} \sum_{j=1}^d (P(v)_j \cdot \mathbf{n}) u_j^0 \, ds$ . Note that the latter one contains no dependency on  $u$  and is therefore part of the force vector.

The regularizer term  $-\eta \int_{\Gamma_D^e} (u - u^0) \cdot v \, ds$  is discretized as

$$(\underline{\mathbf{v}}^e)^T \int_{\Gamma_D^e} \Phi^e(x)^T - \eta \Phi^e(x) \, ds \underline{\mathbf{u}}^e = (\underline{\mathbf{v}}^e)^T \int_{\Gamma_D^e} \Phi^e(x)^T \eta \Phi^e(x) \, ds \underline{\mathbf{u}}^{0e}. \quad (2.5.18)$$

Collecting all these 8-by-8 element matrices and 8D force vectors into large matrices and vectors gives again rise to a linear system with  $2N$  unknowns

$$K \underline{\mathbf{u}} = \underline{\mathbf{f}}_B - \underline{\mathbf{f}}_N \quad (2.5.19)$$

for the static case and

$$M \underline{\ddot{\mathbf{u}}} + K \underline{\mathbf{u}} = \underline{\mathbf{f}}_B - \underline{\mathbf{f}}_N \quad (2.5.20)$$

for the dynamic case. This system is again symmetric and positive definite. Note that the number of unknowns is  $2N$ , which corresponds to the number of active nodes (inside or boundary). On grid elements that are completely outside of the object, the matrices and vectors are zero. This means, the system so far only provides displacement vectors  $u$  for active nodes. This introduces problems with the levelset advection described in the next section.

### 2.5.2.5. Levelset advection

The last step is to advect the levelset with the computed displacements  $u$ . The simulation, however, only provides the displacements on the active nodes, not for the whole domain. This is not enough for the advection. There are two approaches to handle that issue:

## 1. Explicit diffusion:

In a post-processing step, diffuse the displacements into the outer nodes using the simple diffusion process  $\Delta u = 0$  with Dirichlet boundaries at the known vertices and Neumann boundaries at the grid boundary. This could be approximated by a simple flood fill algorithm but we found that that this leads to clearly visible errors.

Advantage: Fast, Soft Body simulation stays unmodified.

Disadvantage: A separate diffusion system has to be solved.

## 2. Implicit diffusion:

Augment the matrix  $K$  with the diffusion kernel

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

that arose from the discretization of  $\Delta u = 0$ .

Again, use Neumann boundaries on the grid boundary. The values for the Dirichlet boundaries are not known in advance because they are the result of the main simulation. This diffusion kernel is added to the nodes that are not part of a grid element that contains the object, i.e. whose matrix entries would be zero. The matrix  $K$  therefore has size  $2wh \times 2wh$  with  $w$  and  $h$  the width and height of the grid (number of nodes), instead of being restricted to active nodes.

Advantage: Solves directly for the displacements on the whole grid.

Disadvantage: The matrix  $K$  is no longer symmetric and positive definite and much larger. This makes it way harder to solve.

It turns out that the first approach, explicit diffusion, is by far superior to the second approach, implicit diffusion. Solving the two smaller symmetric systems (elasticity and diffusion) after each other is way faster and more stable than solving the big, unsymmetric joint system from the implicit diffusion. Hence, we will now always use the explicit diffusion.

Now that we have a dense displacement map  $u$ , the next step is to solve the advection equation

$$\frac{\partial \phi}{\partial t} + u \cdot \nabla \phi = 0. \quad (2.5.21)$$

We tried three different approaches:

## 1. Semi-Lagrange Advection:

This is the standard approach that is used for fluid advection. The new values of the levelset are given by interpolating the current levelset values from the positions back in time

$$\phi^{\text{new}}(x) = \phi^{\text{old}}(x - \Delta t u(x)), \quad (2.5.22)$$

where  $\Delta t$  denotes the time step. Bilinear interpolation is used to access the displacement at positions between grid cells.

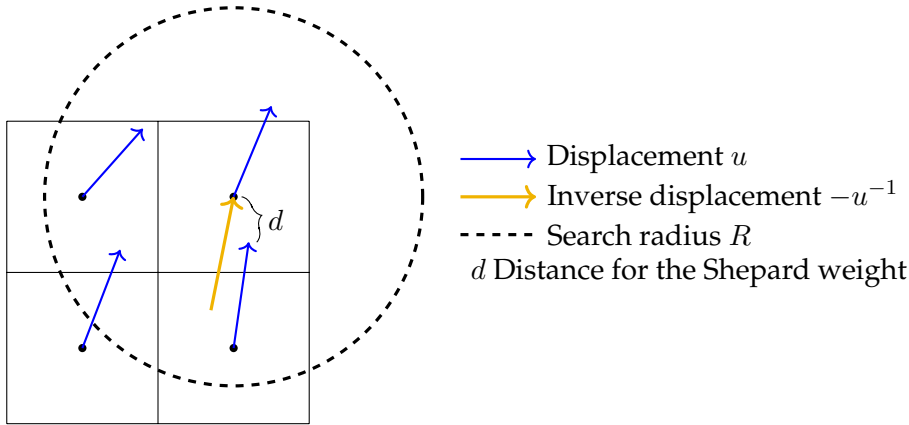


Figure 2.6.: Inversion of a displacement field with Shepard interpolation

2. Semi-Lagrange Advection on the inverse field:

The problem with the first approach is that it takes the displacements that point towards the target as if they would point from the source to the current position. This subtle difference is visualized in Fig. 2.6, it is the difference between the blue arrow (normal displacement) and the yellow arrow (inverse displacement).

For fluid simulations, this error is negligible, since the velocities are very smooth, of smaller magnitude and the velocity field is divergence free. Here in the elasticity, the displacements can become quite large so that these errors are noticeable.

To improve the result, we first calculate the inverse displacements  $u^{-1}$  using a simple Shepard method from [22] (this paper also contains some more advanced methods):

$$-u^{-1}(x) = \frac{\sum_i w(|d_i - u(x + d_i)|)u(x + d_i)}{\sum_i w(|d_i - u(x + d_i)|)}, \quad (2.5.23)$$

where

$$w(d) = \begin{cases} (\frac{1}{d} - \frac{1}{R})^2 & , d \leq R \\ 0 & , \text{otherwise} \end{cases} \quad (2.5.24)$$

is the weighting term and  $R$  the search radius (has to be greater than the maximal displacement). The sum runs over all cells  $i$  within radius  $R$  around  $x$ .  $d_i$  is the search direction / translation from  $x$  to the cell  $i$ . This means, when we check the contribution of the displacement  $u(x + d_i)$  we weight it against the distance between  $x$  and the target position of the displacement at cell  $i$ ,  $x + u(x + d_i)$ . A visualization of this method can be seen in Fig. 2.6.

Finally, given the inverse displacement  $u^{-1}$  at each node, the levelset is advected by

$$\phi^{\text{new}}(x) = \phi^{\text{old}}(x + \Delta t u^{-1}(x)). \quad (2.5.25)$$

### 3. Direct Forward Advection:

The displacement inversion is slow to compute and difficult to invert (that is needed for solving the inverse problems in Sec. 5.5.1). Instead of pulling interpolated values into the current cell like in the Semi-Lagrange approach, here, we take the value of the current cell and blend it into the new levelset at the displaced position using a truncated Gaussian blurring kernel.

The implementation of the algorithm is given in the Appendix in Alg. A.2, with the kernel size given by  $\tau$ . A smaller kernel size leads to less violation of the signed distance property of the output levelset due to the blurring, but it can happen that some cells will not be filled. We use a kernel size of  $\tau = 1.5$  and fill the cells without data (that only occur at the domain boundary) with a simple flood fill.

Fig. 2.7 shows a comparison of the three approaches. The error of approach 1 (only Semi-Lagrange Advection) is clearly visible, the lower end of the disk is not deformed properly. Between the second approach (Shepard-Inversion and Semi-Lagrange Advection) and the third approach (Direct Forward advection), there is almost no visual difference to be seen. Hence, we'll use the Direct Forward advection for it's computational speed and simple adjoint code (see Sec. 5.5.1).

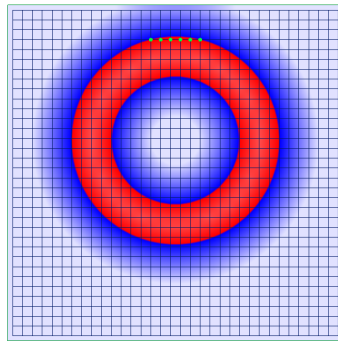
A visualization of the whole procedure of diffusion and advection can be seen in Fig. 2.8. Here, Shepard-Inversion is still used in combination with the Semi-Lagrange advection to visualize the inverse displacement field.

Optionally, the advection of the levelset can be followed by a reinitialization of the levelset to reobtain a signed distance function [60, 59].

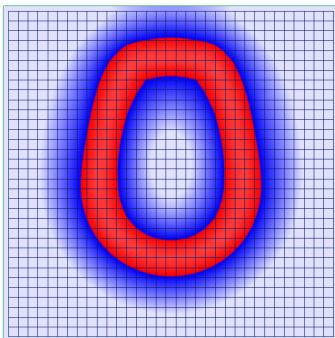
#### 2.5.2.6. Stability

The matrix  $K$  becomes numerically ill-conditioned when the levelset function gets very close to zero at a node, i.e. the boundary runs just past a node. Because of the integration over parts of the cells, the magnitude of entries in  $K$  may differ extremely between neighboring vertices in these cases, resulting in floating point precision errors. A simple way to overcome this problem is to set the value of the levelset at a node to zero if the absolute value lies beneath a certain threshold, e.g.  $1e - 5$ . The integration can handle boundaries exactly through vertices without problems.

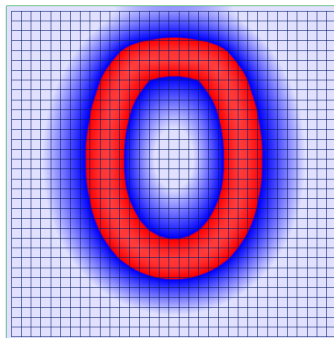
Further, the Nietsche Dirichlet Boundaries can also give rise to stability issues if they are attached to vertices that are almost outside of the object. During the advection of the levelset in the dynamic setting, it can happen that these vertices are now outside of the object due to numerical imprecisions. Hence the Dirichlet Boundaries are not active any longer and the matrix  $K$  becomes degenerated. To fix this, one has to check and modify the position of the Dirichlet Boundaries so that they are always on the current boundary. In an extreme case, one could also revert to hard Dirichlet Boundaries by fixing the displacements of some internal vertices to fixed values.



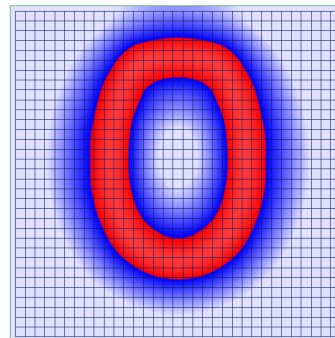
(a) Reference configuration with Dirichlet boundary conditions



(b) only Semi-Lagrange Advection (Approach 1)



(c) Inverted displacements + Semi-Lagrange (Approach 2)



(d) Direct Forward advection (Approach 3)

Figure 2.7.: Comparison of different advection methods

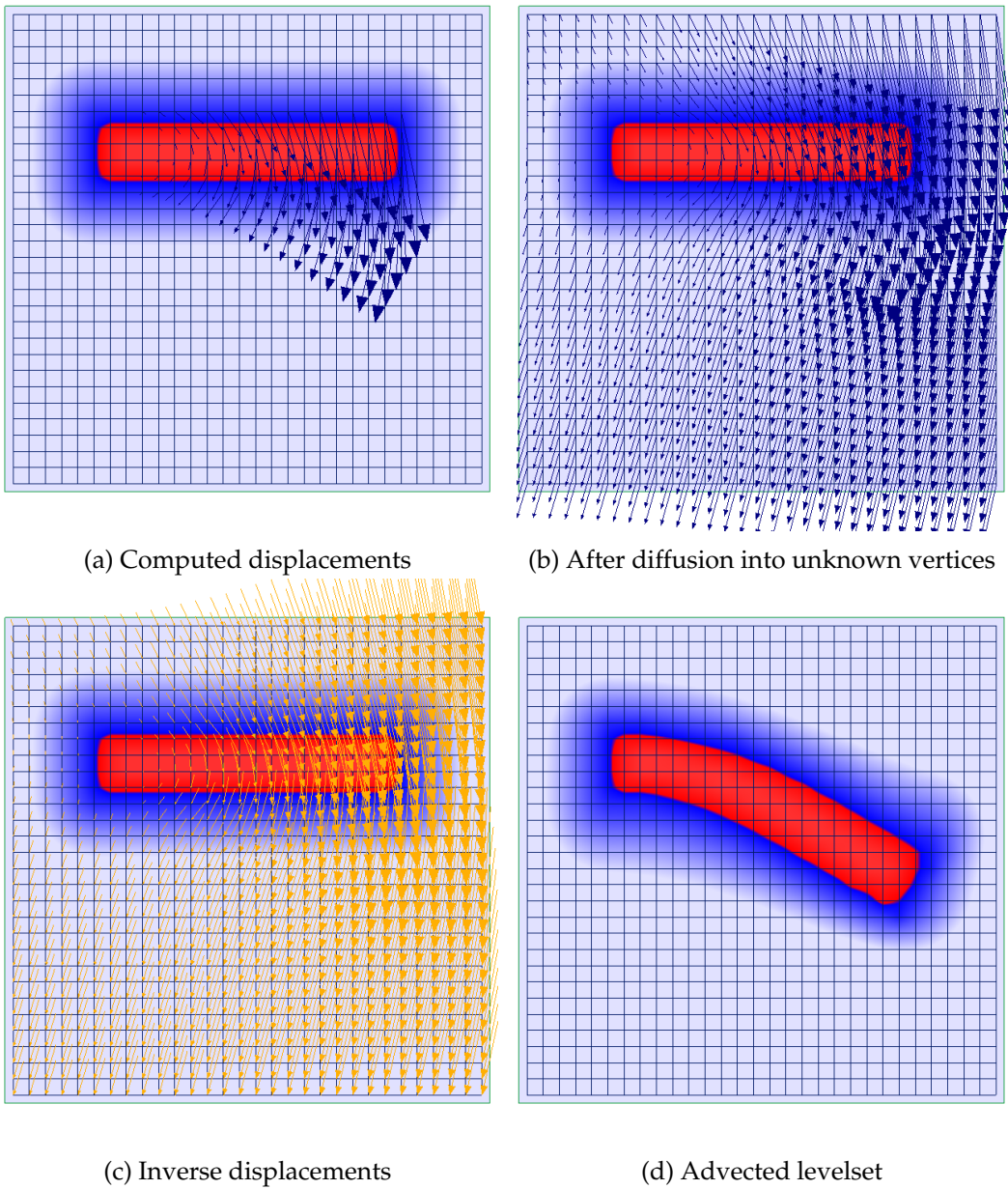


Figure 2.8.: Full procedure of the levelset advection

## 2.6. Time Integration - Dynamic Elasticity

So far, we considered only the static case  $Ku = f$ . The dynamic soft body elasticity  $M\ddot{u} + D\dot{u} + Ku = f$  (with  $D$  being the damping matrix) contains time derivatives and therefore needs a special time integrator. We use the standard *Rayleigh damping*

$$D = \alpha_1 M + \alpha_2 K, \quad (2.6.1)$$

where  $\alpha_1, \alpha_2 \geq 0$  are non-negative constants defining the strength of the damping.

There exist several time integration schemes in the literature. In the following list, we present a selection of the most commonly used methods. Let the time step size be denoted by  $\Delta t$ . The current time step is  $n$  and is computed based on the previous time step  $(n-1)$ . The core of each scheme is again the solution of a linear problem  $Ax = b$ , how  $A$  and  $b$  are computed differs from scheme to scheme:

- Newmark 1 [36]

$$\begin{aligned} & \left( \frac{1}{\theta \Delta t} M + D + \theta \Delta t K \right) u^{(n)} \\ &= \left( \frac{1}{\theta \Delta t} M + D + (1 - \theta) \Delta t K \right) u^{(n-1)} + \frac{1}{\theta} M \dot{u}^{(n-1)} + \Delta t f \end{aligned} \quad (2.6.2)$$

with  $\frac{1}{2} \leq \theta < 1$  and

$$\dot{u}^{(n)} = \frac{1}{\theta \Delta t} (u^{(n)} - u^{(n-1)}) - \frac{1 - \theta}{\theta} \dot{u}^{(n-1)}. \quad (2.6.3)$$

If  $f$  is time-dependent, it is given as

$$f = \theta f^{(n)} + (1 - \theta) f^{(n-1)}. \quad (2.6.4)$$

- Newmark 2 [25]

$$\begin{aligned} & \left( \frac{4}{\Delta t^2} M + \frac{2}{\Delta t} D + K \right) u^{(n)} \\ &= f + M \left( \frac{4}{\Delta t^2} (u^{(n-1)} + \dot{u}^{(n-1)}) + \ddot{u}^{(n-1)} \right) + D \left( \frac{2}{\Delta t} u^{(n-1)} + \dot{u}^{(n-1)} \right) \end{aligned} \quad (2.6.5)$$

with  $\dot{u}^{(n)} = \frac{2}{\Delta t} (u^{(n)} - u^{(n-1)}) - \dot{u}^{(n-1)}$   
and  $\ddot{u}^{(n)} = \frac{4}{\Delta t^2} (u^{(n)} - u^{(n-1)} - \Delta t \dot{u}^{(n-1)}) - \ddot{u}^{(n-1)}$ .

- Explicit Central Differences [32]

$$\begin{aligned} & \left( \frac{1}{\Delta t^2} M + \frac{1}{2\Delta t} D \right) u^{(n)} \\ &= \left( \frac{2}{\Delta t^2} M - K \right) u^{(n-1)} + \left( \frac{-1}{\Delta t^2} M + \frac{1}{2\Delta t} D \right) u^{(n-2)} + f. \end{aligned} \quad (2.6.6)$$



- Implicit Linear Acceleration [32]

$$\begin{aligned}
 & \left( M + \frac{\Delta t}{2}D + \frac{\Delta t^2}{6}K \right) \ddot{u}^{(n)} \\
 &= \left( -\frac{\Delta t}{2}D - \frac{\Delta t^2}{3}K \right) \ddot{u}^{(n-1)} - Ku^{(n-1)} - D\dot{u}^{(n-1)} - \Delta tK\dot{u}^{(n-1)} + f \\
 \dot{u}^{(n)} &= \dot{u}^{(n-1)} + \frac{\Delta t}{2}(\ddot{u}^{(n)} + \ddot{u}^{(n-1)}) \\
 u^{(n)} &= u^{(n-1)} + \Delta t\dot{u}^{(n-1)} + \frac{\Delta t^2}{6}(\ddot{u}^{(n)} + 2\ddot{u}^{(n-1)}). \tag{2.6.7}
 \end{aligned}$$

- Newmark 3 [32]

$$\begin{aligned}
 & \left( \frac{6}{\Delta t^2}M + \frac{3}{\Delta t}D + A \right) \Delta x \\
 &= f + \left( 3M + \frac{\Delta t}{2}D \right) \ddot{u}^{(n-1)} + \left( \frac{6}{\Delta t}M + 3D \right) \dot{u}^{(n-1)} \\
 u^{(n)} &= u^{(n-1)} + \Delta x \\
 \dot{u}^{(n)} &= -2\dot{u}^{(n-1)} - \frac{\Delta t}{2}\ddot{u}^{(n-1)} + \frac{3}{\Delta t}\Delta x \\
 \ddot{u}^{(n)} &= -M^{-1}(D\dot{u}^{(n)} + Ku^{(n)} - f). \tag{2.6.8}
 \end{aligned}$$

- HHT- $\alpha$  Method [32]

$$\begin{aligned}
 & (M + \Delta t(1 - \alpha)\gamma D + \Delta t^2(1 - \alpha)\beta K) \ddot{u}^{(n)} \\
 &= f - \left( \Delta t(1 - \alpha)(1 - \gamma)D + \Delta t^2(1 - \alpha)\left(\frac{1}{2} - \beta\right)K \right) \ddot{u}^{(n-1)} \\
 & \quad - (D + \Delta t(1 - \alpha)K) \dot{u}^{(n-1)} \\
 & \quad - Ku^{(n-1)} \\
 u^{(n)} &= u^{(n-1)} + \Delta t\dot{u}^{(n-1)} + \Delta t^2 \left( \left(\frac{1}{2} - \beta\right)\ddot{u}^{(n-1)} + \beta\ddot{u}^{(n)} \right) \\
 \dot{u}^{(n)} &= \dot{u}^{(n-1)} + \Delta t \left( (1 - \gamma)\ddot{u}^{(n-1)} + \gamma\ddot{u}^{(n)} \right) \tag{2.6.9}
 \end{aligned}$$

with  $0 \leq \alpha \leq \frac{1}{3}$ ,  $\beta = (1 + \alpha)^2/4$ ,  $\gamma = \frac{1}{2} + \alpha$ .

All these methods, except for the Explicit Central Differences, convergence for a sufficiently small timestep. The Explicit Central Differences produces growing oscillations if the damping is too small. Experiments show that Newmark 1 (Eq. (2.6.2)) is the most robust integration method that can handle small and large time steps.

Note that the time integration only acts on the displacements of the free vertices. This means for the grid simulation, that the time integration acts on the free vertices with respect to the initial configuration. The advection part is independently performed in every time step on the new displacements. The resulting signed distance function does not influence the region over which the integrals of the elasticity simulation are evaluated in the next time step. The integration region stays the same for all time steps.

## 2.7. Corotational Formulation

Recall that in Sec. 2.4 we used the linear Green Strain instead of the full nonlinear St. Venant-Kirchoff Strain. This introduces artificial forces in situations with large rotations that blow up the elements, see Fig. 2.9.

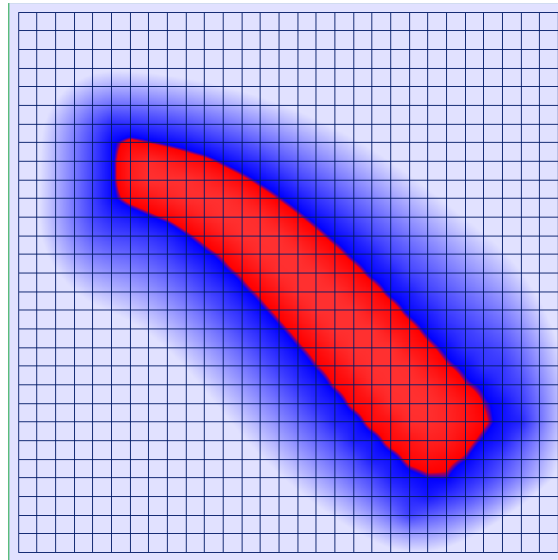


Figure 2.9.: Strong rotation introduces a blow-up effect

In order to correct these artificial forces, a corotation formulation [25, 72, 40] is used. The basic idea is to first subtract the rotation from the displacements, then compute the strain and stress, and finally add the rotation again. In the matrix formulation, the per-element term  $K^e \underline{\mathbf{u}}^e$  is replaced by

$$T^e K^e ((R^e)^T (\underline{\mathbf{x}}^e + \underline{\mathbf{u}}^e) - \underline{\mathbf{x}}^e), \quad (2.7.1)$$

where the matrix  $R^e$  contains the purely rotational part of the displacement of the element  $e$ . This equation contains both  $\underline{\mathbf{x}}^e$  and  $\underline{\mathbf{u}}^e$ . While  $\underline{\mathbf{u}}^e$  is unknown, the part with  $\underline{\mathbf{x}}^e$  is

known and has to be added to the force term. This leads to the following decomposition of Eq. (2.7.1):

$$\begin{aligned} f^e &\leftarrow f^e - R^e K^e ((R^e)^T \underline{\mathbf{x}}^e - \underline{\mathbf{x}}^e) \\ K^e &\leftarrow R^e K^e (R^e)^T \underline{\mathbf{u}}^e \end{aligned} \quad (2.7.2)$$

with  $K^e$  and  $f^e$  being the per-element stiffness matrix and force vector.

The rotational part of the displacement  $\hat{R}^e$  can be computed in the following way:

First, the Jacobian of the displacement  $F := \nabla \varphi = \nabla u + \mathbf{1}$  (see Eq. (2.2.3)) per element  $F^e$  is computed.

- 2D Mesh Discretization:

Let  $\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3, \mathbf{u}_1, \mathbf{u}_2, \mathbf{u}_3 \in \mathbb{R}^2$  be the reference position and the displacement of the three vertices. Let  $\mathbf{d}_i = \mathbf{x}_i + \mathbf{u}_i$  be the deformed position of vertex  $i$ . Then:

$$F^e = \begin{pmatrix} \mathbf{d}_{1,x} - \mathbf{d}_{3,x} & \mathbf{d}_{2,x} - \mathbf{d}_{3,x} \\ \mathbf{d}_{1,y} - \mathbf{d}_{3,y} & \mathbf{d}_{2,y} - \mathbf{d}_{3,y} \end{pmatrix} \begin{pmatrix} \mathbf{x}_{1,x} - \mathbf{x}_{3,x} & \mathbf{x}_{2,x} - \mathbf{x}_{3,x} \\ \mathbf{x}_{1,y} - \mathbf{x}_{3,y} & \mathbf{x}_{2,y} - \mathbf{x}_{3,y} \end{pmatrix}^{-1}. \quad (2.7.3)$$

- 2D Grid Discretization:

Let  $i, j$  be the position of the current cell with the vertices  $(i, j)$  to  $(i + 1, j + 1)$ . Then the value of  $F$  at the four vertices that are needed for the integration are given by the outer products

$$\begin{aligned} F_{i,j} &= \mathbf{u}_{i,j} \begin{pmatrix} -1 & -1 \end{pmatrix} \\ F_{i+1,j} &= \mathbf{u}_{i+1,j} \begin{pmatrix} 1 & -1 \end{pmatrix} \\ F_{i,j+1} &= \mathbf{u}_{i,j+1} \begin{pmatrix} -1 & 1 \end{pmatrix} \\ F_{i+1,j+1} &= \mathbf{u}_{i+1,j+1} \begin{pmatrix} 1 & 1 \end{pmatrix}. \end{aligned} \quad (2.7.4)$$

The final value of  $F$  is then computed as

$$F^e = \frac{1}{2} (F_{i,j} + F_{i+1,j} + F_{i,j+1} + F_{i+1,j+1} + \mathbf{1}^2). \quad (2.7.5)$$

Note that no integration over the partially filled cells is required, since the displacement is fully defined on all four vertices.

Then the rotational part of  $F$  is extracted using the polar decomposition  $F = RS$  where any matrix  $F$  can be decomposed into a rotation (orthonormal) matrix  $R$  and a symmetric positive definite matrix  $S$ . Algorithms for the general case were first presented in [41], repeated and improved e.g. in [42, 40, 71].

For 2x2 matrices, an explicit expression is available [71]:

$$R = F^e + \text{sign}(\det(F)) \begin{pmatrix} F_{2,2}^e & -F_{2,1}^e \\ -F_{1,2}^e & F_{1,1}^e \end{pmatrix}. \quad (2.7.6)$$

This rotation matrix is applied to every vertex of the cell in Eq. (2.7.1). This is realized by repeating the matrix  $R$  several times in the element matrix  $\hat{R}^e$

$$R^e := \begin{pmatrix} R & & \\ & \ddots & \\ & & R \end{pmatrix}, \quad (2.7.7)$$

where  $R$  is repeated three times for the 2D mesh or four times for the 2D grid.

The algorithm as pseudocode is presented in the Appendix in Alg. A.3.

Important:

In the grid simulation, the rotation correction has to be applied to  $K_e$  and  $F_e$  **before** the Nietsche-Dirichlet boundaries (Eq. (2.5.17)) are added.

The corotational formulation only repairs the stresses when there are already some deformations. Otherwise, the Jacobian  $F$  is zero and  $R$  should be set to the identity matrix. Therefore, it can directly be plugged in the dynamic simulation. Surprisingly, it can also be applied in the static simulation by repeatedly solving the static simulation while using the displacements of the previous solution for the corotational formulation to repair the forces. Convergence is typically achieved within a few iterations, see Fig. 2.10.

## 2.8. Collision

Collision is a topic in the elasticity simulation that was covered by many research articles in the last decade. The problem is usually split into two separate tasks. First, the collision detection: when do two objects collide, what is the collision normal. See e.g. Georgii *et al.* [33] for a GPU detection algorithm and Teschner *et al.* [79] for a study on hierarchical collision checks. Second, the collision resolution and the computation of contact forces: the idea is to formulate the contact as an inequality condition  $C(x) \leq 0$  that has to be fulfilled for a valid state (a simple case is the distance to the ground). Advanced models also include Coulomb Friction and other effects like drag in water. These constraints can be included into the elasticity simulation using Lagrange multipliers. For details on this method see [43, 31]. For full physical accuracy using a Linear Complementary Problem (LCP) see [76, 2] for rigid bodies and [4, 54] for soft bodies. A solver for the LCP is presented in [26].

Since an implementation of the LCP is beyond the scope of this these, we will only use a simple spring system. This is a good trade-off between physical accuracy and algorithmical complexity. The basic idea comes from rigid body collisions and mass-spring systems [73], and is used extensively in cloth simulation (see e.g. [13]) and also volumetric elasticity simulation (see e.g. [20]).

Fig. 2.11 visualizes the idea of a spring based collision handling. Wherever the object enters the ground, a virtual spring is attached to it that generates a repulsive force. The latter pushes the object back out of the ground. Because we use lumped mass for both the

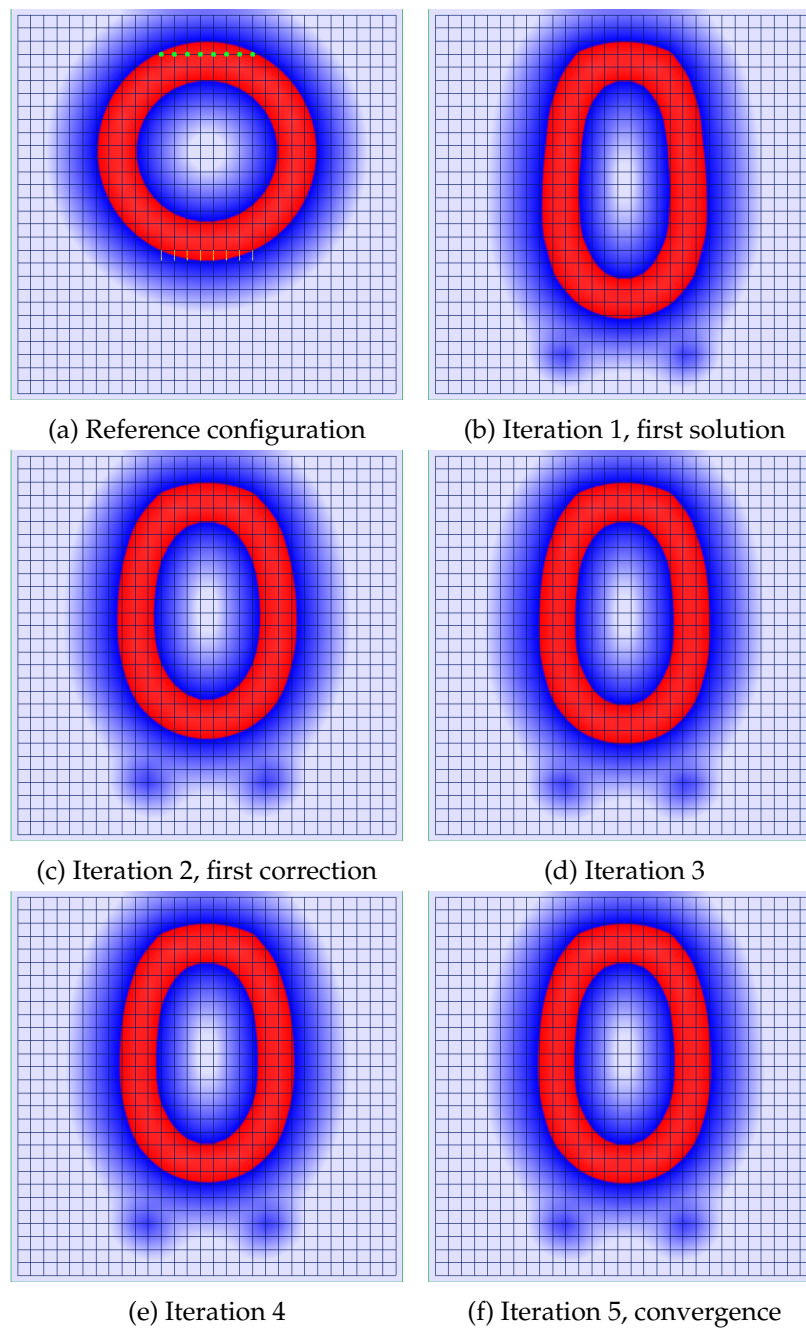


Figure 2.10.: Iteratively solving the rotation corrected static solution

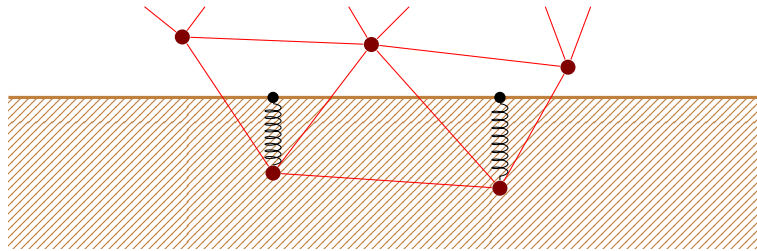


Figure 2.11.: Collision handling with virtual springs on a simple mesh

mesh and grid simulation, these springs have to be attached only on the nodes, not continuously over the whole intersecting area. Further we assume that the ground has infinite mass and that for each position  $\mathbf{x}$  we can compute the signed distance to the ground as  $x = \text{dist}(\mathbf{x})$ .

The force of a spring is described by Hooke's Law:  $\mathbf{f} = -kx\mathbf{n}$  with the stiffness factor  $k$ , the distance  $x$  from the rest position (negative for compression, positive for extension) and  $\mathbf{n}$  is the unit outward normal vector. In our case, the spring must not exert an attractive force towards the ground when the object is outside the ground. The spring should only repulse the object on penetrations. Therefore, we use the following clamped force, visualized in Fig. 2.12 (red plot):

$$\mathbf{f}_c = -k \min(0, x)\mathbf{n}. \quad (2.8.1)$$

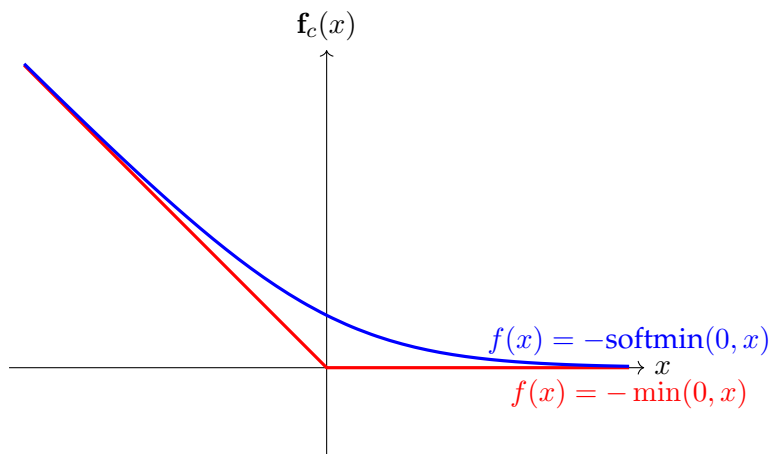


Figure 2.12.: Collision forces on the nodes

For each penetrating node, this force is added to the Neumann boundary conditions and integrated in the next time step. This method, however, is an explicit method and requires very small time steps for a stable simulation. Recall from Sec. 2.6 that if the forces

in the Newmark integration (see Eq. (2.6.2)) are time dependent, they are computed as  $b = \theta b^{(n)} + (1 - \theta)b^{(n-1)}$ . So for the simple explicit formulation, we set  $b^{(n)} = b^{(n-1)} = f_c^{(n-1)}$  for the computation of the next time step. An implicit method is achieved when  $f_c^{(n)}$  is used directly in the Newmark integration. For that, however, we need the positions at the next time step, which are yet unknown. Further,  $f_c^{(n)}$  can't be added to the left side (integrated in the matrix  $A$  of Newmark), because the above model is non-linear. Hence, we have to revert to an approximation of  $f_c^{(n)}$ :

$$f_c^{(n)} \approx f_c^{(n-1)} + \Delta t \frac{\partial f_c^{(n-1)}}{\partial t}. \quad (2.8.2)$$

The above equation (2.8.1) for  $f_c$  is non-differentiable at  $x = 0$ . Therefore, we use a non-physical smoothing by replacing the hard minimum with a soft minimum (blue graph in Fig. 2.12):

$$\begin{aligned} \mathbf{f}_c &= -k \text{softmin}(0, x) \mathbf{n} \\ \text{with } \text{softmin}_\alpha(a, b) &:= -\ln(e^{-x\alpha} + e^{-y\alpha}) / \alpha. \end{aligned} \quad (2.8.3)$$

The parameter  $\alpha$  represents the "softness". As it approaches positive infinity, this softmin becomes the regular (hard) minimum function. This formulation was proposed by Cook [18] and a numerical accurate implementation for  $\ln(1 + e^x)$  that arises in our special case  $\text{softmin}_\alpha(0, x)$  is given by Mächler [50]. As a side effect, since the softmin is positive for the whole domain, it already introduces small repulsive forces when the object is close to entering. This reduces the amount of penetration into the ground. We can compute the time derivative now as

$$\begin{aligned} \frac{\partial f_c^{(n-1)}}{\partial t} &= \frac{\partial -k \text{softmin}_\alpha(0, \text{dist}(\mathbf{x}^{(n-1)}))}{\partial t} \\ &= \frac{1}{1 + e^{\alpha \text{dist}(\mathbf{x}^{(n-1)})}} \text{dist}_t(\dot{\mathbf{x}}^{(n-1)}). \end{aligned} \quad (2.8.4)$$

By plugging Eq. (2.8.4) and Eq. (2.8.2) into Eq. (2.6.4) we obtain the resulting force  $f = \partial f_c^{(n-1)} + \theta \Delta t \frac{\partial f_c^{(n-1)}}{\partial t}$  as the Neumann force. Together with Eq. (2.5.8), that gives rise to the right hand side  $b$  of the Newmark time integration.

The collision handling on the grid simulation is very similar to the one on the mesh, see Fig. 2.13. The essential difference lies in how collisions are detected and the Neumann boundary conditions are applied. First, instead of checking every node of the grid whether they penetrate the ground, these tests are performed on the analytic intersection of the surface, represented by the signed distance function values on the grid nodes, with the grid cell edges. For these at most four points on the edges per cell, the spring forces are computed as above. Each force is then added to the two adjacent vertices weighted by the distance to the vertex (linear interpolation).

As an example, in Fig. 2.14 (mesh simulation) and Fig. 2.15 (grid simulation), a torus with a small Young's Modulus collides and bounces off a sloped floor.

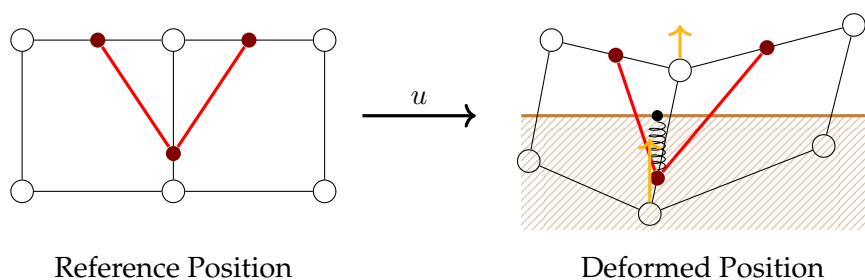


Figure 2.13.: Collision detection and handling for the grid simulation

## 2.9. Conclusion

We presented two discretization schemes for the elasticity simulation: on a tetrahedra mesh and on a regular grid with an implicit object representation using a SDF. Previously, the regular grid, in which the object boundary does not coincide with the cell boundaries, was mostly attempted for Navier-Stokes fluid simulations. We have shown that it can also be used for soft body elasticity simulations.

An extensive comparison of the mesh and grid approach is done in the 3D implementation, see Sec. 4.



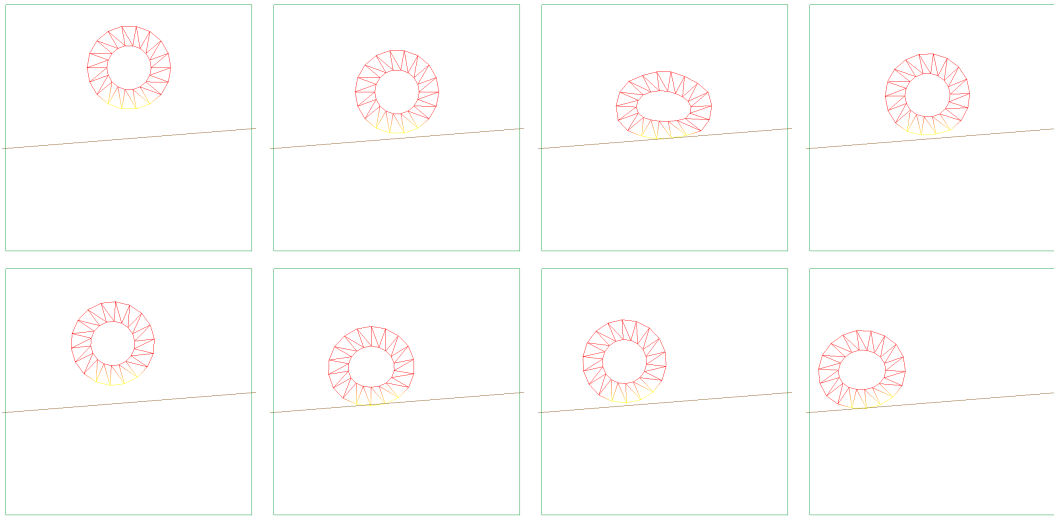


Figure 2.14.: Torus colliding with the ground: mesh simulation

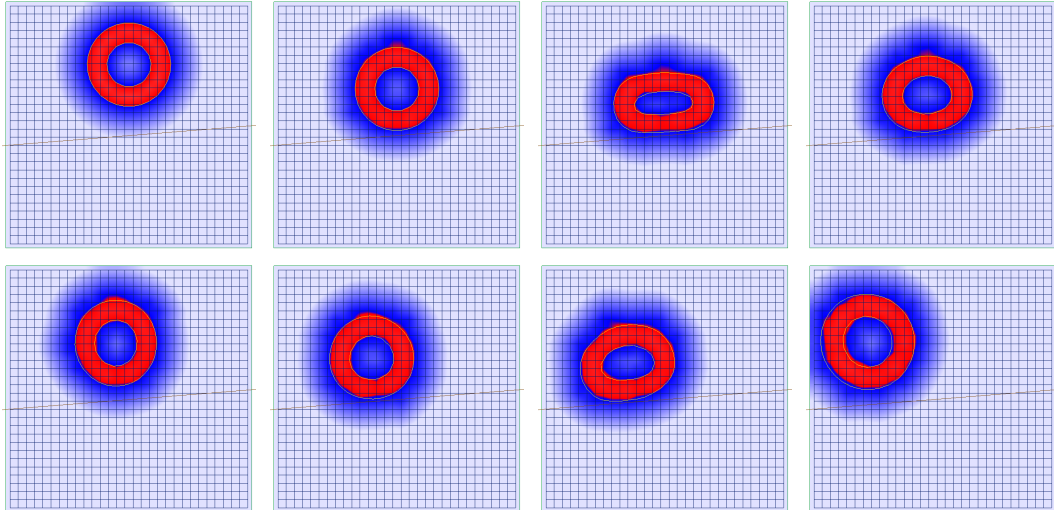


Figure 2.15.: Torus colliding with the ground: grid simulation



### 3. cuMat - Linear Algebra Library for CUDA

Before describing how to efficiently implement the soft body simulation in 3D, we introduce *cuMat* [84], a linear algebra library for CUDA [58], newly developed by the author of this thesis. *cuMat* has been made available as open-source under the MIT license<sup>1</sup>. The goal of *cuMat* is to provide a port of *Eigen* [37], a popular library for linear algebra computation on the CPU in C++, to the massive parallel architecture of CUDA. *cuMat* is later used in the 3D implementation of the soft body simulation in Sec. 4.

The usage of *Eigen* is very simple. For example, if one wants to sum up two vectors *a* and *b* into a third vector *c*, one simply writes

---

```
1 Eigen::VectorXf a = ..., b = ...; //some initializations
2 Eigen::VectorXf c = a + b; //CPU
```

---

This so called "AXPY" operation is already provided on the GPU in the library *cuBLAS* by NVIDIA [56], enabling the performance improvement of the GPU. The above code implemented with *cuBLAS* reads as follows (ignoring error checks):

---

```
1 int n = ...; //size of the vectors
2 float* a = ..., b = ...; //some initializations
3 float* c = ...; //output memory
4 cublasHandle_t handle;
5 cublasCreate(&handle);
6 float alpha = 1; //optional scaling factor of b; axpy: c += alpha * b
7 cudaMemcpy(c, a, sizeof(float)*n, cudaMemcpyDeviceToDevice); //copy a into c, GPU
8 cublasSaxpy(handle, n, &alpha, b, 1, c, 1); //add b to c, GPU
9 cublasDestroy(&handle);
```

---

This code is quite verbose and the usage of *cuBLAS* becomes even more complicated for more complex operations. Hence, *cuMat* strives to provide an API similar to *Eigen*, but everything is executed on the GPU:

---

```
1 cuMat::VectorXf a = ..., b = ...; //some initialization
2 cuMat::VectorXf c = a + b; //GPU
```

---

One of the main reasons why *Eigen* is so fast, is the Kernel Merging enabled by Expression Templates. Kernel Merging means that expressions like  $\mathbf{d} = \mathbf{a} + \mathbf{b} + \mathbf{c}$  are not evaluated as individual sums into temporary vectors ( $\mathbf{v}' = \mathbf{a} + \mathbf{b}$ ,  $\mathbf{d} = \mathbf{v}' + \mathbf{c}$ ), but instead are evaluated in a single loop ( $\forall i : \mathbf{d}_i = \mathbf{a}_i + \mathbf{b}_i + \mathbf{c}_i$ ). This drastically reduced the number

---

<sup>1</sup><https://gitlab.com/shaman42/cuMat>

of memory accesses and thereby improves the performance. Since memory bandwidth is most of the time the limiting factor on the GPU, *cuMat* also implements Kernel Merging, see Sec. 3.4.

The development of *cuMat* has started before this thesis, but it was greatly extended by the author for this thesis, e.g. by sparse operations, iterative solvers and inplace compound assignments.

## 3.1. Related Work

There already exist many linear algebra libraries for the GPU, most noticeable *cuBlas*, *cuSolver* and *cuSparse* (developed by NVIDIA [56]), as well as *MAGMA* (a university collaboration [81]). All these libraries provide their functionality through verbose BLAS-like functions.

In the works of Wiemann *et al.* [85] and Tillet *et al.* [80, 68] it was shown that Kernel Merging with Expression Templates is possible on the GPU. A simple implementation of these methods was done in *VexCL* [24], but limited to basic operations on vectors. They didn't address the problems that arise with more complicated matrix-matrix operations (see Sec. 3.4). Further implementations can be found in *ViennaCL* [67] for OpenCL.

*Eigen* claims that it can also be executed in CUDA<sup>2</sup>, but only for operations on small matrices on a per-thread basis. No parallelization over the whole GPU is performed.

*Tensorflow* [1], the de-facto standard library for machine learning applications, provides GPU implementations for matrix operations with a simple-to-use Python frontend. Only recently, *Tensorflow* started to implement Kernel Merging (XLA<sup>3</sup>), but this feature is still in development and not enabled by default.

## 3.2. A (very short) review of CUDA

In this section we introduce the concepts behind the parallel execution in CUDA in a simplified way. For a complete introduction to CUDA, we refer to the books of Cook [19] and Kirk [46].

In CUDA, a task (called a kernel) is parallelized over a three-dimensional grid. This grid is subdivided into blocks of uniform size, called a work group. Each block is again subdivided into threads. Each thread then processes one item in the task. On the hardware, the work groups are executed on Streaming Multiprocessors (SM). Each SM computes one work group at a time and processes 32 threads at the same time (called a warp) in a SIMD fashion. The number of threads per work group is limited by the available registers per SM. This implies the following constraints when developing parallel code:

- Synchronization between threads can only be performed within a work group

---

<sup>2</sup><https://eigen.tuxfamily.org/dox/TopicCUDA.html>

<sup>3</sup><https://www.tensorflow.org/performance/xla/>

- The number of threads should be a multiple of 32, otherwise some threads in the SM are unused
- The number of threads should be high enough to hide memory latency, but small enough to fit all used variables into registers
- Branching is expensive since both branches have to be executed serially, unless the branching condition is the same for all threads within a warp.

In *cuMat*, these constraints are hidden from the user, the library computes the optimal grid and work group size automatically. They will be relevant again for the matrix assembly in Sec. 4.4.1.

### 3.3. Batched Evaluation and the Matrix class

In *cuMat*, every operation supports batched evaluation, i.e. every matrix has a third dimension "batches" over which every operation is parallelized. This allows the parallel execution of the same expression over different data, stored in different batches. For example, the batched evaluation support can be seen directly in the actual type of the matrix class:

---

```
1 template<typename T, int Rows, int Columns, int Batches, int Flags> Matrix;
```

---

The individual parameters have the following meaning:

- `T`: the scalar type of the matrix, e.g. `float` or `cdouble` (complex double)
- `Rows`, `Columns`, `Batches`: size of the matrix on compile time. If not known, the symbol `Dynamic` is used. Providing as much information as possible during compile time allows *cuMat* to perform some optimizations. For example, loops are unrolled, explicit formulas for matrix determinants and inverses are used for tiny (batched) matrices and similar optimizations. Further, some operations like a component-wise inner and outer product are only available for compile-time vectors (either `Rows` or `Columns` is one).
- `Flags`: additional flags, currently contains only the storage order (column major or row major)

The batched evaluation is combined with a broadcasting mechanism: For an operation  $a \circ b$ , if  $a$  has a batch size of one and  $b$  a batch size greater one or dynamic, then the entries of  $a$  are broadcasted over the batches when applied to  $b$ .

One application of this method is a batched solve. Assume that we have a non-batched matrix  $A$  and a batched vector  $b$ . Then *cuMat* allows us to use a batched conjugate gradient solver that solves for all batches in  $b$  at once:

---

```
1 ConjugateGradient<MatrixXf> cg(A);
2 BVectorXf x = cg.solve(b); //prefix 'B': predefined batched types
```

---

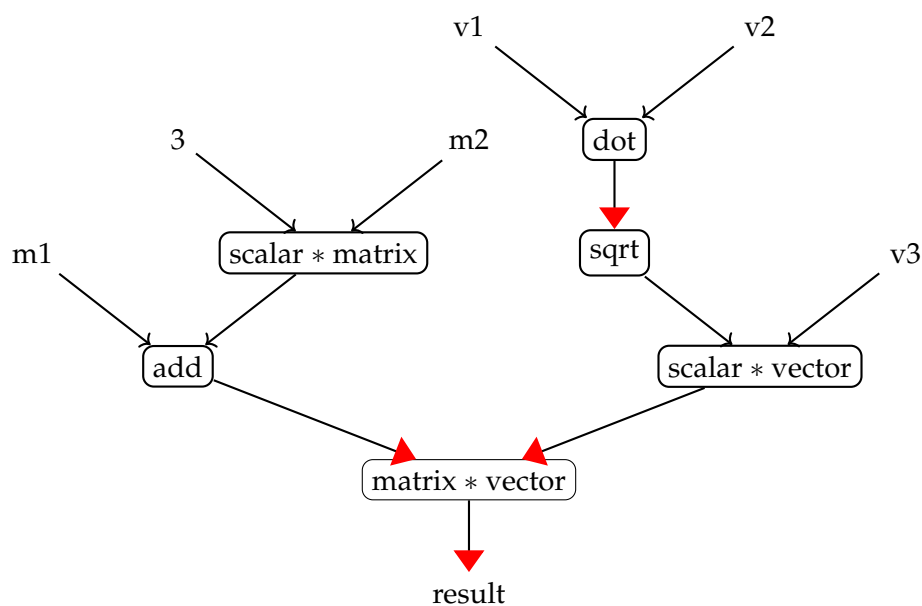


Figure 3.1.: Expression tree of `result = (m1+3*m2) * (v1.dot(v2).cwiseSqrt()*v3)`, red arrow tips indicate evaluations

### 3.4. Expression Templates and Kernel Merging

In a general-purpose linear algebra library, various operations might be combined. For demonstration purpose, assume we want to compute the following:

---

```

1 cuMat::VectorXf v1, v2, v3;
2 cuMat::MatrixXf m1, m2;
3 cuMat::MatrixXf result = (m1+3*m2) * (v1.dot(v2).cwiseSqrt()*v3);

```

---

The concept of Expression Templates is that an operation (e.g.  $a + b$ ) does not immediately compute the result, but returns an object that represents this operation (e.g. `BinaryOp<VectorXf, VectorXf, Add>`). These objects are then chained to a full evaluation tree as in Fig. 3.1, and are only evaluated when assigned to a matrix.

The reality, however, is not that simple. Each operation has different access requirements on the inputs and different access properties of the output. The three possible access modes of the inputs and outputs are:

- None: no assumption, only for the output,
- Component-wise: component-wise reading is possible,
- Direct: a raw pointer to the data is needed, only for the input, implies Component-wise access.

If the access mode of an output does not fulfill the requirements for the input to the next stage, the expressions are evaluated and the results saved to temporary memory. For example, all component-wise operations (add, subtract, scalar multiplication, component-wise square root, ...) have "Component-wise" as input and output, allowing them to be chained together. A reduction (dot-product in the example), on the other hand, can take Component-wise expression as inputs, but returns the result with the access mode "None", hence forcing an evaluation. In Fig. 3.1, arrows with a thick red tip indicate evaluations.

The reason why evaluation only happens on the inputs, instead of directly on the outputs, is best explained in the following code snippet:

---

```

1 cuMat::MatrixXfR mat1, mat2; //row-major
2 cuMat::MatrixXfC mat3; //column-major
3 mat2 = mat1.transpose(); //explicit transpose
4 mat3 = mat1.transpose(); //no-op, contents are simply reinterpreted

```

---

This example shows that the evaluation requires knowledge of the target layout to choose, which operation should be performed.

For further details we refer to the documentation of *cuMat*<sup>4</sup>.

### 3.5. Benchmarks

For a performance evaluation, we compared *cuMat* to *cuBlas*, *Eigen* and *numpy* in four different test cases described below. All test cases have in common that the CPU libraries (*Eigen*, *numpy*) are often faster for small input sizes due to the overhead of GPU calls, but are clearly outperformed by the GPU libraries (*cuMat*, *cuBlas*) for larger problem sizes. The benchmarks were executed on a desktop computer running Windows 10 Home, with an Intel i7-6700 CPU (3.4GHz, 4 physical cores, 8 logical cores), 16.0GB RAM and a NVIDIA GeForce GTX 1060 6GB GPU. The reported times are averages of ten individual runs. The exact results can be found in the Appendix in Table B.1 to B.4.

The first two test cases are a linear combination of vectors  $\sum_i^k \alpha_i v_i, v_i \in \mathbb{R}^n$ . This is a test on how kernel merging improves the performance. In Fig. 3.2a, the number of combination is fixed to two and the size of the vector varies. Here, *cuBlas* outperforms *cuMat*, but since the source code of *cuBlas* is not public, we don't know the reasons for the good performance of *cuBlas*. Further, the CPU libraries *Eigen* and *numpy* show a linear increase in computation time with the size of the input, while the GPU libraries almost stay constant. This highlights the parallel computation power of the GPU.

The other example for the linear combination is shown in Fig. 3.2b. Here, the size of the vector is constant and the number of combinations varies. *cuMat* now outperforms *cuBlas* for four or more combinations. This clearly shows the advantage of the Kernel Merging approach with a single evaluation kernel over multiple basic operations as in *cuBlas*.

---

<sup>4</sup><https://shaman42.gitlab.io/cuMat/>

Next we evaluate the performance of our custom sparse matrix (CSR-format) - vector multiplication routine in Fig. 3.2c. Our implementation achieves the same performance as the optimized routine provided by NVIDIA's *cuSparse* library, sometimes even slightly better.

Last, Fig. 3.2d shows how the libraries perform in a practical algorithm, in this example a Conjugate Gradient solver. This solver requires sparse matrix - vector products, reductions and other component-wise operations, thus testing every aspect of the library. The example problem is the 2D diffusion equation with random Dirichlet and Neumann boundaries. The solver is run until convergence (tolerance of  $10^{-6}$ ) and therefore the number of iterations also increases. For a grid size of  $1000 \times 1000$ , the implementation in *cuMat* is about 10x as fast as the implementation in *Eigen*.

## 3.6. Conclusion

We presented a library, *cuMat* for linear algebra computations on the GPU. This library offers competitive performance to other GPU libraries and outperforms any tested CPU library, given a sufficient large input.

Furthermore, the simple and intuitive API reduces the development time and avoids errors in the verbose Blas-like calls of existing libraries.



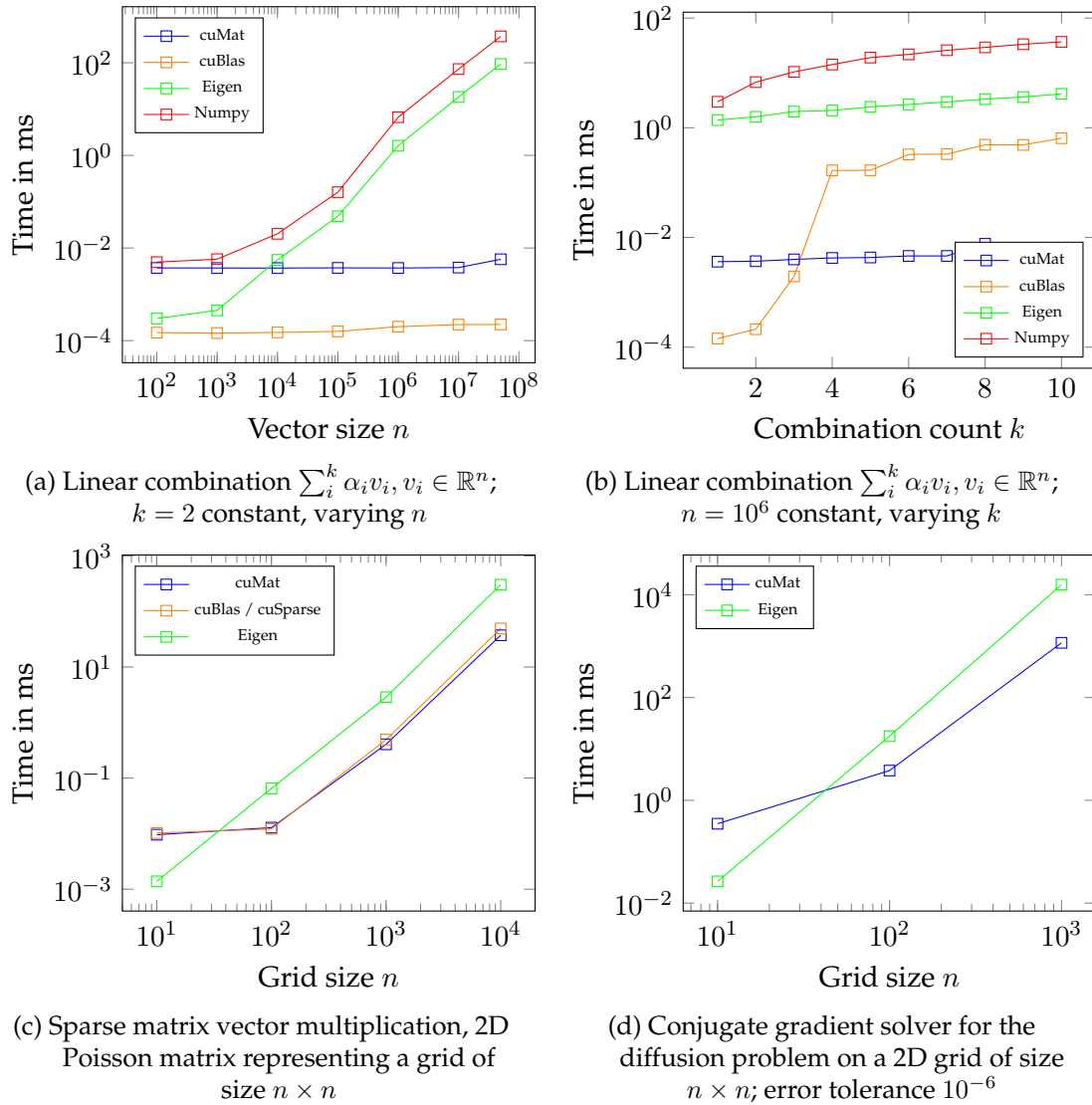


Figure 3.2.: Benchmark of *cuMat* and other libraries



## 4. 3D Implementation with CUDA

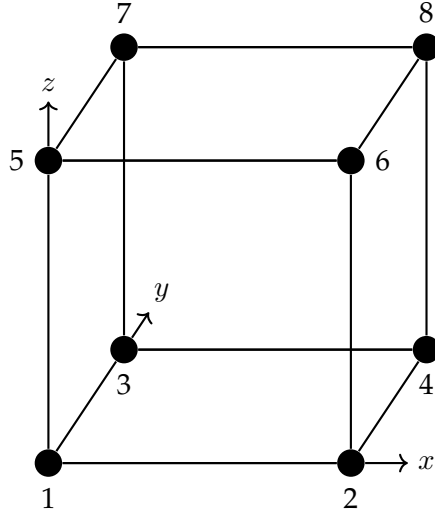
In Sec. 2.5.2, we described the discretization of the elasticity problem on the regular grid, mainly targeted for 2D. Now we present how to efficiently implement the elasticity simulation on the regular grid in 3D with the help of *cuMat* (see Sec. 3) on the GPU. Since we use the novel partially filled hexahedral cells, a new parallel algorithm for computing the stiffness matrix is presented.

### 4.1. Related Work

Methods on how to accelerate the elasticity simulation on the GPU have been proposed since the early days of programmable graphics hardware. For example, Tejada and Ertl [78] have already shown in the year 2005 how to simulate mass-spring based elasticity using pixel shaders on the GPU. With the introduction of CUDA 2006, GPU solvers became simpler to implement, e.g. shown by Liu *et al.* [48] for tetrahedra meshes without corotation. Dick [25] showed 2011 how to implement FEM elasticity on hexahedral elements (without partial integrals) with corotation and proposed a matrix-free multigrid solver. Recent advances in the GPU implementations have made FEM based elasticity ready for real-time applications [30, 21, 86], mostly by improving the iterative solver for the large linear system. Further, Lamecki *et al.* [47] showed how to support higher-order FEM elements on the GPU.

### 4.2. Basis Functions and Partial Integrals

In 3D on a regular grid, the FEM elements take the form of cubes, referred to as *cells* here. Each cell has eight incident vertices / nodes. The side length of each cell is denoted by  $h$ . Then the labelling of the eight nodes and the respective basis functions are given as follows:



$$\begin{aligned}
 N_1(\mathbf{x}) &:= \left(1 - \frac{x}{h}\right) \left(1 - \frac{y}{h}\right) \left(1 - \frac{z}{h}\right) \\
 N_2(\mathbf{x}) &:= \frac{x}{h} \left(1 - \frac{y}{h}\right) \left(1 - \frac{z}{h}\right) \\
 &\vdots \\
 N_8(\mathbf{x}) &:= \frac{x}{h} \frac{y}{h} \frac{z}{h}
 \end{aligned}$$

Furthermore, we want to evaluate integrals over parts of the cell. Let  $\phi_1, \dots, \phi_8$  be the values of the SDF at the eight corners of the cell. Let  $\Omega^e \subseteq [0, 1]^3$  be the part of the cell that is inside the volume defined by the SDF, and  $\Gamma^e = \partial\Omega^e$  the boundary surface. We assume for the evaluation of any function over  $\Omega^e$  that it can be represented as a linear interpolation of its value at the eight corners

$$f(x) = \sum_{i=1}^8 f(v_i) N_i(x) \quad (4.2.1)$$

with  $v_1, \dots, v_8$  being the coordinates of the eight corners. This allows us to write the integrals in the following form:

$$\int_{\Omega^e} f(x) \, dx = \int_{\Omega^e} \sum_{i=1}^8 f(v_i) N_i(x) \, dx = \sum_{i=1}^8 f(v_i) \underbrace{\int_{\Omega^e} N_i(x) \, dx}_{=:w_v(e,i)} \quad (4.2.2)$$

$$\int_{\Gamma^e} f(s) \, ds = \int_{\Gamma^e} \sum_{i=1}^8 f(v_i) N_i(s) \, ds = \sum_{i=1}^8 f(v_i) \underbrace{\int_{\Gamma^e} N_i(s) \, ds}_{=:w_b(e,i)} \quad (4.2.3)$$

The integrals  $w_v$  and  $w_b$  can be solved analytically. For the details we refer to Appendix C.

### 4.3. Datastructures and Sparsity Pattern

In this section, we will collect the data that is needed to perform all steps in the elasticity simulation: Stiffness matrix assembly (Sec. 2.5.2.4, Eq. (2.5.15), Sec. 2.7), external forces (Sec. 2.8) and advection (Sec. 2.5.2.5).

### 4.3.1. World Grid

The reference SDF  $\phi$  is given on a regular 3D grid in space. From this grid we first derive the active cells (cells with at least an inner vertex) and the active nodes (nodes that are a vertex of an active cell). Then we define a mapping  $m : \mathbb{Z}^3 \rightarrow \mathbb{Z}$  that maps a position in the 3D grid into a linear index for the matrices and vectors. Active nodes are numbered in increasing order starting by 1 (x-coordinate is fastest) and inactive nodes in decreasing order starting by -1. This allows us to use the same mapping grid for both the elasticity simulation as well as the diffusion and levelset advection (Sec. 4.4.5).

### 4.3.2. Input Data per Node and Element

For the evaluation of the per cell integrals that arise in the stiffness matrix and forces, the following data is needed per element  $e$ :

- Values of the reference SDF at the eight incident nodes  $\phi(e, i)$  (8x float),
- Precomputed integration weights for the partial cell integrals over the volume and boundary  $w_v(e, i), w_b(e, i)$  (2x 8x float),
- Precomputed surface normal for the Dirichlet boundaries  $n(e)$  (3x float),
- Mapping of the eight incident nodes to the global linear index  $s(e, i)$ . Because the mapping is created in such a way that the index increases fastest in x-direction, only 4 integer values have to be stored (for vertex 1, 3, 5 and 7).

### 4.3.3. Blocked CSR for the Stiffness Matrix

Recapitulate that the global stiffness matrix  $K$  is assembled by the element stiffness matrices  $K_e = B_e^T C B_e$ . This matrix has a very special form. It is a very sparse matrix of small 3x3 blocks. Each of these 3x3 blocks encodes the relationship between the 3 degrees of freedom (3D-simulation) of a pair of nodes. Therefore, it is advantageous to save the stiffness matrix in a Block Compressed Sparse Row format (BSR).

In the Compressed Sparse Row (CSR) format, a matrix  $A \in \mathbb{R}^{m \times n}$  is stored using three tables: *val*, the non-zero entries of  $A$  in row-major format, *colInd*, the column indices of the non-zero entries, and *rowPtr*, an array of size  $m + 1$  where *rowPtr*[i] points to the first entry in row i [57]. As an example, the matrix

$$A := \begin{pmatrix} 1 & 4 & 0 & 0 & 0 \\ 0 & 2 & 3 & 0 & 0 \\ 5 & 0 & 0 & 7 & 8 \\ 0 & 0 & 9 & 0 & 6 \end{pmatrix}$$

is stored in the CSR format with zero-based indexing as

$$\begin{aligned} \text{val} &= [1, 4, 2, 3, 5, 7, 8, 9, 6] \\ \text{colInd} &= [0, 1, 1, 2, 0, 3, 5, 2, 4] \\ \text{rowPtr} &= [0, 2, 4, 7, 9]. \end{aligned}$$

In Block CSR, each entry in *val* is not a single scalar element, but a dense matrix of fixed size. In our case, 3x3 blocks.

This storage format drastically reduces the amount of memory needed to store the outer and inner indices, since they have to be stored only per 3x3 block, not per single scalar value. Similarly, the force vectors are also blocked vectors of 3x1 vectors.

Furthermore, *cuMat* allows the use of custom datatypes as scalar elements for matrices and elements. We utilize this feature by defining a custom structure for 3x3-matrices:

---

```
1 typedef float4 real3; //built-in vector type in CUDA of four 4-byte floats.
2 struct real3x3 {real3 r1,r2,r3;}; //a 3x3 matrix in row-major order
3 typedef cuMat::Matrix<real3, Dynamic, 1, 1> Vector3X;
4 typedef cuMat::SparseMatrix<real3x3, 1, CSR> SparseMatrix;
```

---

Note that we use `float4` as our vector type, although we only need a three-dimensional vector. This is because CUDA provides vector load and store operations that can access a `float4` in memory with one single instruction. This is not possible for a `float3`.

From the perspective of *cuMat* our blocked stiffness matrix is a simple CSR matrix with a custom scalar type. The advantages of this method are the following:

- By providing appropriate overloads for the scalar operations, these blocked matrices can directly be used in all operations of *cuMat*, including the Conjugate Gradient solver.
- Simple access to the per-node data. In the assembly (Sec. 4.4.1 and Sec. 4.4.2), always either the whole 3x3 block (matrix) or the 3x1 block (force vector) is needed. For example, `float fy = forceNonBlocked[3*e+1]` is simplified to `float fy = forceBlocked[e].y`.
- Efficient memory access. We use the built-in type `float4` as the underlying type, for which CUDA provides memory access instructions that can read the whole `float4` in one command instead of accessing the individual elements in three consecutive commands.

Last, we precompute the sparsity pattern (inner and outer indices) of the stiffness matrix. This sparsity pattern only depends on the reference configuration and does not change over the time steps.

## 4.4. Matrix Assembly and Solving

In this section we present the changes at the equations from 2D to 3D and the steps required for an efficient parallel implementation on the GPU.

### 4.4.1. Stiffness Matrix

The final per-element stiffness matrix takes contributions from three parts: the basic stiffness matrix from the elasticity simulation (Eq. (2.5.15)), the corotation correction (Eq. (2.7.2)) and the Nietsche-Dirichlet boundaries (Eq. (2.5.17)) in that order.

#### 4.4.1.1. Elasticity

First, recall that the element stiffness matrix is given by (compare with Eq. (2.5.15))

$$K^e := \int_{\Omega^e} B^e(x)^T C B^e(x) dx \in \mathbb{R}^{24 \times 24}, \quad (4.4.1)$$

where the 3D-versions of  $B^e(x)$  and  $C$  is given by (compare with Eq. (2.5.4))

$$B^e(x) := \left( \begin{array}{ccc|ccc} \frac{\partial N_1(x)}{\partial x_1} & & & \dots & \frac{\partial N_8(x)}{\partial x_1} & & \\ & \frac{\partial N_1(x)}{\partial x_2} & & & & \frac{\partial N_8(x)}{\partial x_2} & \\ & & \frac{\partial N_1(x)}{\partial x_3} & & & & \frac{\partial N_8(x)}{\partial x_3} \\ \frac{\partial N_1(x)}{\partial x_2} & \frac{\partial N_1(x)}{\partial x_1} & & & \frac{\partial N_8(x)}{\partial x_2} & \frac{\partial N_8(x)}{\partial x_1} & \\ \frac{\partial N_1(x)}{\partial x_3} & & \frac{\partial N_1(x)}{\partial x_1} & & \frac{\partial N_8(x)}{\partial x_3} & & \frac{\partial N_8(x)}{\partial x_1} \\ & \frac{\partial N_1(x)}{\partial x_3} & \frac{\partial N_1(x)}{\partial x_2} & & \frac{\partial N_8(x)}{\partial x_3} & \frac{\partial N_8(x)}{\partial x_2} & \frac{\partial N_8(x)}{\partial x_1} \end{array} \right) \in \mathbb{R}^{6 \times 24} \quad (4.4.2)$$

and (compare Eq. (2.5.5))

$$C := \begin{pmatrix} 2\mu + \lambda & \lambda & \lambda & & & & & \\ \lambda & 2\mu + \lambda & \lambda & & & & & \\ \lambda & \lambda & 2\mu + \lambda & & & & & \\ & & & \mu & & & & \\ & & & & \mu & & & \\ & & & & & \mu & & \end{pmatrix} \in \mathbb{R}^{6 \times 6}. \quad (4.4.3)$$

Observe that the matrix  $K^e$  can be decomposed into 3x3 blocks as

$$K^e = \begin{pmatrix} K_{1,1} & \cdots & K_{1,8} \\ \vdots & \ddots & \vdots \\ K_{8,1} & \cdots & K_{8,8} \end{pmatrix}. \quad (4.4.4)$$

Each block  $K_{i,j}$  describes the relationship between the two nodes  $i$  and  $j$  of the current cell. They are all constructed in the same way as

$$K_{i,j}(x) = \begin{bmatrix} (2\mu + \lambda) \frac{\partial N_i(x)}{\partial x_1} \frac{\partial N_j(x)}{\partial x_1} + \mu \left( \frac{\partial N_i(x)}{\partial x_2} \frac{\partial N_j(x)}{\partial x_2} + \frac{\partial N_i(x)}{\partial x_3} \frac{\partial N_j(x)}{\partial x_3} \right) \\ \mu \frac{\partial N_i(x)}{\partial x_1} \frac{\partial N_j(x)}{\partial x_2} + \lambda \frac{\partial N_i(x)}{\partial x_2} \frac{\partial N_j(x)}{\partial x_1} \\ \mu \frac{\partial N_i(x)}{\partial x_1} \frac{\partial N_j(x)}{\partial x_3} + \lambda \frac{\partial N_i(x)}{\partial x_3} \frac{\partial N_j(x)}{\partial x_1} \\ \mu \frac{\partial N_i(x)}{\partial x_2} \frac{\partial N_j(x)}{\partial x_1} + \lambda \frac{\partial N_i(x)}{\partial x_1} \frac{\partial N_j(x)}{\partial x_2} \\ (2\mu + \lambda) \frac{\partial N_i(x)}{\partial x_2} \frac{\partial N_j(x)}{\partial x_2} + \mu \left( \frac{\partial N_i(x)}{\partial x_1} \frac{\partial N_j(x)}{\partial x_1} + \frac{\partial N_i(x)}{\partial x_3} \frac{\partial N_j(x)}{\partial x_3} \right) \\ \mu \frac{\partial N_i(x)}{\partial x_2} \frac{\partial N_j(x)}{\partial x_3} + \lambda \frac{\partial N_i(x)}{\partial x_3} \frac{\partial N_j(x)}{\partial x_2} \\ \mu \frac{\partial N_i(x)}{\partial x_3} \frac{\partial N_j(x)}{\partial x_1} + \lambda \frac{\partial N_i(x)}{\partial x_1} \frac{\partial N_j(x)}{\partial x_3} \\ \mu \frac{\partial N_i(x)}{\partial x_3} \frac{\partial N_j(x)}{\partial x_2} + \lambda \frac{\partial N_i(x)}{\partial x_2} \frac{\partial N_j(x)}{\partial x_3} \\ (2\mu + \lambda) \frac{\partial N_i(x)}{\partial x_3} \frac{\partial N_j(x)}{\partial x_3} + \mu \left( \frac{\partial N_i(x)}{\partial x_1} \frac{\partial N_j(x)}{\partial x_1} + \frac{\partial N_i(x)}{\partial x_2} \frac{\partial N_j(x)}{\partial x_2} \right) \end{bmatrix}, \quad (4.4.5)$$

$$\in \mathbb{R}^{3 \times 3}.$$

(For layouting purposes, this matrix is displayed with the three columns underneath each other)

For the integration over the cells, the  $K_{i,j}$ 's are evaluated at the eight corner points  $c$  and then summed together, weighted by  $w_v(e, c)$  as computed in Sec. 4.2.

This splitting of the big matrix  $K^e$  into the small blocks allows the efficient parallelization on the GPU. It is also applicable for the Nietsche Dirichlet boundaries and the corotation formulation.

#### 4.4.1.2. Corotation

Recall from the description of the corotation in Sec. 2.7 that the element stiffness matrix  $K^e$  and the force vector  $f^e$  is changed to

$$f^e \leftarrow f^e - R^e K^e ((R^e)^T \underline{\mathbf{x}}^e - \underline{\mathbf{x}}^e) \quad (4.4.6)$$

$$K^e \leftarrow R^e K^e (R^e)^T \underline{\mathbf{u}}^e. \quad (4.4.7)$$

in order to compensate for the rotation.

To compute the element rotation  $R^e$ , first the average deformation gradient is computed as shown in Dick [25]:

$$F^e = \mathbf{1}_3 + \frac{1}{4h} \sum_{i=1}^8 \mathbf{u}_{s(e,i)} \begin{pmatrix} (-1)^i \\ (-1)^{\lceil i/2 \rceil} \\ (-1)1^{\lceil i/4 \rceil} \end{pmatrix}^T. \quad (4.4.8)$$

The rotational component  $R^e$  is then given by the polar decomposition  $F^e = R^e S^e$ . It can be computed using the following iterative scheme:

$$R^{(0)} = F^e, \\ R^{(i)} = \frac{1}{2} \left( R^{(i-1)} + (R^{(i-1)})^{-T} \right) \quad \text{for } i > 0. \quad (4.4.9)$$



Dick [25] reports that 5 iterations are usually enough for a good estimate of the rotation.

The application of this rotation matrix to the element stiffness matrix and the force vector can again be formulated using the 3x3 blocking

$$\begin{aligned} f_i &\leftarrow f_i - \sum_{j=1}^8 R^e K_{i,j} ((R^e)^T \mathbf{v}_j - v_j), \\ K_{i,j} &\leftarrow R^e K_{i,j} (R^e)^T, \end{aligned} \quad (4.4.10)$$

where  $v_j$  is the position of the  $j^{\text{th}}$  corner of the cell as before.

#### 4.4.1.3. Nietsche Dirichlet Boundaries

The Nietsche Dirichlet Boundaries in the weak form are given by

$$- \int_{\Gamma_N^r} P(u) \cdot \mathbf{n} \cdot v \, ds - \int_{\Gamma_N^r} P(v) \cdot \mathbf{n} \cdot (u - u_0) \, ds - \eta \int_{\Gamma_N^r} (u - u_0) \cdot v \, ds, \quad (4.4.11)$$

see Eq. (2.4.5). These terms are added to the left side of the weak form of the PDE.

For simplicity we only support boundary conditions with  $u_0 = 0$ . Furthermore, we assume that if a cell has Dirichlet boundaries, the whole surface boundary inside this cell are Dirichlet boundaries. Plugging in the definition of  $P(u)$  and simplifying the above equation leads to the following terms that are added to the weak form:

$$\begin{aligned} & - \underbrace{\int_{\Gamma_D^e} \sum_{j=1}^3 \left( \mu \left( \nabla u_j \cdot \mathbf{n} + \frac{\partial u}{\partial x_j} \cdot \mathbf{n} \right) + \lambda n_j \sum_{i=1}^3 \frac{\partial u_i}{\partial x_i} \right)}_{\text{(I)}} v_j \, ds \\ & - \underbrace{\int_{\Gamma_D^e} \sum_{j=1}^3 \left( \mu \left( \nabla v_j \cdot \mathbf{n} + \frac{\partial v}{\partial x_j} \cdot \mathbf{n} \right) + \lambda n_j \sum_{i=1}^3 \frac{\partial v_i}{\partial x_i} \right)}_{\text{(I)}} u_j \, ds \\ & - \underbrace{\eta \int_{\Gamma_D^e} u \cdot v \, ds}_{\text{(II)}} \end{aligned} \quad (4.4.12)$$

Expressing  $u$  and  $v$  using the basis functions and their derivatives, combined in  $\Phi^e$  and  $B^e$ , we obtain

$$K^e = \int_{\Gamma_D^e} \Phi^e(x)^T D^e B^e(x) \, ds \quad (4.4.13)$$

for (4.4.12.I) with  $D^e$  given by

$$D^e := \begin{pmatrix} 2\mu n_1 + \lambda n_1 & \lambda n_1 & \lambda n_1 \\ \lambda n_2 & 2\mu n_2 + \lambda n_2 & \lambda n_2 \\ \lambda n_3 & \lambda n_3 & 2\mu n_3 + \lambda n_3 \end{pmatrix}. \quad (4.4.14)$$

The equation for (4.4.12.I') is the above equation, just transposed. Again, this integral is evaluated by computing the values at the eight corners and summing them up weighted by  $w_b(e, c)$  similar to how the stiffness matrix was computed in Sec. 4.4.1.1. The value of  $\Phi_i$  evaluated at vertex  $c$  has the special property that  $\Phi_i = \mathbf{1}_3 \mathbb{1}_{i=c}$ . This allows us to derive the following simplification and solution of Eq. (4.4.13), combining both (4.4.12.I) and (4.4.12.I'):

$$\text{KD}_{j,c} := \begin{bmatrix} B_1(\lambda n_1 + 2\mu n_1) + B_2\mu n_2 + B_3\mu n_3 \\ B_2\mu n_1 + B_1\lambda n_2 \\ B_3\mu n_1 + B_1\lambda n_3 \end{bmatrix},$$

$$\begin{bmatrix} B_2\lambda n_1 + B_1\mu n_2 \\ B_1\mu n_1 + B_2(\lambda n_2 + 2\mu n_2) + B_3\mu n_3 \\ B_3\mu n_2 + B_1\lambda n_3 \end{bmatrix},$$

$$\begin{bmatrix} B_3\lambda n_1 + B_1\mu n_3 \\ B_3\lambda n_2 + B_2\mu n_3 \\ B_1\mu n_1 + B_2\mu n_2 + B_3(\lambda n_3 + 2\mu n_3) \end{bmatrix} \in \mathbb{R}^{3 \times 3} \quad (4.4.15)$$

with  $B_i := \frac{\partial N_j(v_c)}{\partial x_i}$

$$K_{i,j} = w_b(e, i)\text{KD}_{j,i} + w_b(e, j)\text{KD}_{i,j}^T. \quad (4.4.16)$$

Part (4.4.12.II) even simplifies to the following expression:

$$K_{i,j} = \mathbf{1}_{i=j} \eta w_b(e, i) \mathbf{1}_3. \quad (4.4.17)$$

For the Nietsche constant  $\eta$ , a rather large value is required for a stable simulation. We used  $\eta = 10^8$  in our experiments.

#### 4.4.1.4. Final Algorithm

The computation of the full matrix  $K^e$  in one CUDA thread is too expensive, but each single  $K_{i,j}$  can be computed as described in the previous sections without running into hardware limits. Furthermore, we can arrange the computation of these  $8 * 8 = 64$  blocks in a work group of size 64 (which is a multiple of 32, hence all threads of a SM are utilized), and utilize shared memory and synchronization between the threads to avoid duplicate computations. In an algorithmic view, this looks as in Alg. 4.1. For the evaluation of the  $K_{i,j}$ 's and the Dirichlet boundaries, the values of the derivatives of the basis functions  $N_1, \dots, N_8$  with respect to  $x, y, z$  are needed at the eight corner points. Since these values only depend on the grid size  $h$ , they are precomputed and stored in a constant array `__constant__ float derivatives[8][8][3]`.

#### 4.4.2. Mass and Force Vector

The computation of the mass matrix and force vector (gravity) differ only slightly from the 2D version in Eq. (2.5.14) and Eq. (2.5.16), therefore we don't repeat the computation

**Algorithm 4.1** Sketched kernel code for the computation of the block  $K_{i,j}$ 


---

```

1 //Launched with a work size of 64
2 __global__ void GridComputeStiffnessMatrixKernel(...) {
3   int elementIdx = blockIdx.x; //element index e
4   int i = threadIdx.x % 8; // threadIdx.x goes from 0 to 63
5   int j = threadIdx.x / 8;
6   float wv[8] = ...; // Fetch the volume integration weights  $w_v(e,i)$  from memory
7   float mapping[8] = ...; // Fetch the mapping indices  $s(e,i)$  from memory
8
9   //each thread computes  $K_{i,j}$ , integrated over the volume
10  real3x3 KePart = {0}
11  for (int c=0; c<8; ++c) {
12    KePart += wv[c] * real3x3(...); //  $K_{i,j}$  evaluated at  $v_c$ 
13  }
14
15  //Corotation
16  __shared__ __device__ real3x3 rot; //shared memory over all threads
17  __shared__ __device__ real3 subforces[8][8];
18  if (j==0) {
19    real3 displacement = lastDisplacements[mapping[i]]; // fetch  $u_{s(e,i)}$ 
20    real3x3 FPart = real3x3::OuterProduct(displacement,
21      make_real3(i&1?-1, i&2?-1, i&4?-1)/h);
22    real3x3 F = ...; // warp reduce with __shfl_down to assemble  $F^e$ , see Eq. (4.4.8)
23    if (i==0) { // only one thread computes the polar decomposition
24      rot = polarDecomposition(F); // see Eq. (4.4.9), written into shared memory
25    }
26  }
27  __syncthreads(); // now every thread can read 'rot'
28  real3 pos = h * make_real3(j&1?1:0, j&2?1:0, j&4?1:0); //  $v_j$ 
29  subforces[i][j] = rot*KePart*(rot.transpose()*pos-pos); // see Eq. (4.4.6)
30  __syncthreads();
31  KePart = rotation*KePart*rotation.transpose(); // see Eq. (4.4.7)
32  if (j==0) {
33    real3 fi = ...; // reduction of subforces[i][j] over j, final value of  $f_i$ 
34    atomicSub(outputForce + mapping[i], fi); // update force vector
35  }
36
37  //Dirichlet boundaries
38  if (isDirichlet(elementIdx)) {
39    float wb[8] = ...; // Fetch the boundary integration weights  $w_b(e,i)$  from memory
40    float3 n = ...; // Fetch the surface normal from memory
41    KePart -= ...; // Nietsche condition, see Eq. (4.4.16)
42    KePart -= ...; // Nietsche regularizer, see Eq. (4.4.17)
43  }
44
45  //Find the position of  $K_{i,j}$  in the large sparse matrix  $K$ 
46  // Fetch the range of inner indices from the outer index table JA
47  for (int k = JA[mapping[i]]; k < mapping[mapping[i]+1]; ++k) {
48    if (IA[k] == mapping[j]) { // inner index matches
49      atomicAdd(KData + k, KePart); // atomically add  $K_{i,j}$  to  $K$ 
50    }
51  }
52 }

```

---

## 4. 3D Implementation with CUDA

---

here. Note that, since we use lumped mass, the mass matrix is diagonal. Hence, every of the diagonal entries for the 3x3 block per node is the same, so we only store one float per node, instead of e.g. three floats for each coordinate as in the body forces. Since these vector's don't change, they are precomputed into the vectors

---

```
1 cuMat::Matrix<real, cuMat::Dynamic, 1, 1, ColumnMajor> lumpedMass;  
2 cuMat::Matrix<real3, cuMat::Dynamic, 1, 1, ColumnMajor> bodyForces;
```

---

Collision forces are computed exactly as in the 2D case (Sec. 2.8) with the difference that all 12 edges of the cell have to be tested for intersections with the ground. The details are presented when the adjoint version are handled in Alg. 5.5. The collision forces are computed per time step and added to the force vector.

All the three operations described above are implemented on the GPU. Each thread computes the contribution of a single cell, the work group size is chosen to optimize occupancy.

### 4.4.3. Newmark Time Integration

We use the Newmark 1 integration scheme as presented in Eq. (2.6.2), together with Rayleigh damping (Eq. (2.6.1)). For the time integration, a linear system  $Au^{(n)} = b$  has to be solved where  $A$  and  $b$  are given by:

---

```
1 SparseMatrix A; Vector3X b; //preallocated  
2 A = ((1/(theta*delta_t) + alpha_1) * lumpedMass).cast<real3>().asDiagonal()  
3   + real3x3((alpha_2 + theta*delta_t)) * stiffnessMatrix;  
4 b = ((1/(theta*delta_t) + alpha_1) * lumpedMass).cast<real3>().asDiagonal()  
5   + (alpha_2 + (1 - theta)*delta_t) * stiffnessMatrix  
6   .sparseView<CSR>(stiffnessMatrix.getSparsityPattern()) * prevDisplacement  
7   + (1/theta * lumpedMass).cast<real3>().cwiseMul(prevVelocity)  
8   + real3(delta_t) * bodyForces;
```

---

In the above code, the Rayleigh damping is directly plugged in to the Newmark scheme, so that the damping matrix  $D$  is never explicitly computed. Here, the Kernel Merging of *cuMat* is used to its extremes. No intermediate storage is used at all, the only evaluation (i.e. kernel call) is done at the two assignments. Further, one can see the following design decisions of *cuMat* in action:

- No implicit type conversion between matrices of different types, the explicit casting expression `.cast<NewType>()` has to be used. This decision was made to prevent obscure errors introduced by implicit type casts.
- Scalar-Matrix multiplication requires the same data types, hence the castings to `real3` or `real3x3`. This is again a safety measure required by *cuMat*.
- `.asDiagonal()` converts a vector into a virtual diagonal matrix.

- Line 6, `.sparseView(...)`: This is called on an addition-expression of two matrices. Without `.sparseView(...)`, the following matrix-vector multiplication would trigger a dense evaluation since `cuMat` can't infer the sparsity pattern of the addition. With this token, the sparsity pattern of the stiffness matrix (that contains the diagonal, hence the lumped mass entries are included) is enforced and a sparse matrix-vector multiplication is triggered, that is evaluated component-wise.

#### 4.4.4. Conjugate Gradient Solver

Next, a standard Conjugate Gradient Solver with a diagonal preconditioner is used to solve the above linear system. We use

$$u_{\text{init}} = u^{(n)} + \Delta t \dot{u}^{(n)} \quad (4.4.18)$$

as initialization for the solution to the next timestep  $u^{(n+1)}$ . With this choice, the CG typically converges within a few (10-30) iterations.

#### 4.4.5. Levelset Advection

Lastly, for the levelset advection, the same ideas as presented for the 2D case in Sec. 2.5.2.5 are applied.

First, the displacements are diffused into the empty nodes by solving a 3D Poisson diffusion equation for each of the three coordinates of the displacements. This is done with a batched CG solve on a precomputed diffusion matrix. Solving a linear system with a batched right hand side is twice as fast as solving a separate system for each coordinate. As an optimization, we only compute the nodes within a distance of five nodes (in l1-norm) to the surface (the active nodes). The nodes farther away do not influence the surface location during the advection step. This allows us to drastically reduce the number of nodes to diffuse if the object has large empty areas within the grid.

Second, the levelset is advected with the Direct Forward method with the following improvement: Since the object can travel arbitrary long distances over the course of the simulation, it is very ineffective to include everything in a gigantic grid. Instead we first compute a bounding box that encloses the active nodes translated by their current displacement. This bounding box then specifies the grid (plus some extra cells as border) in which the Direct Forward method writes the advected SDF into.

## 4.5. Rendering

We provide three rendering methods to visualize the results of the elasticity simulation. A comparison of the three methods can be found in Fig. 4.1.

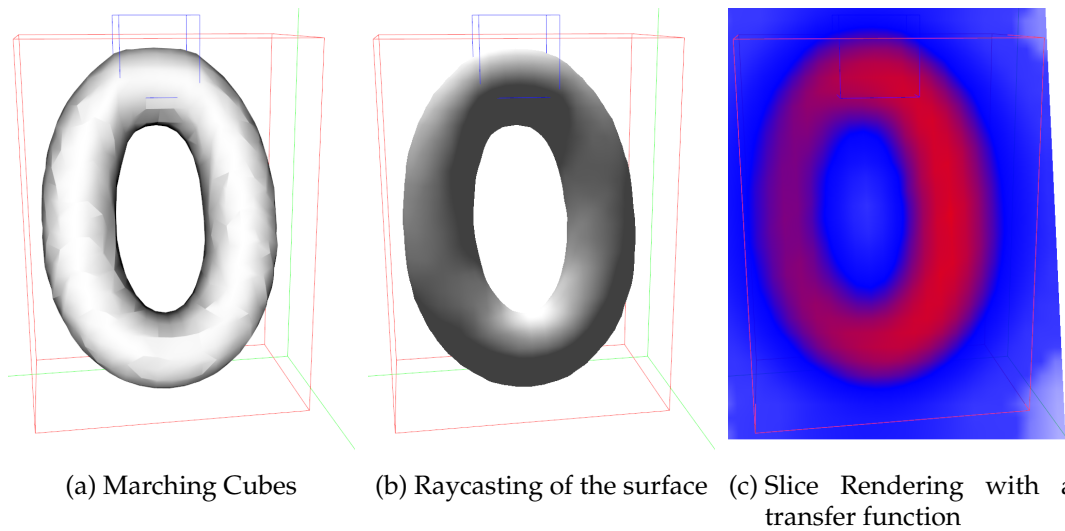


Figure 4.1.: Comparison of the three rendering methods to visualize the results of the elasticity simulation. Blue box: dirichlet boundaries, red box: bounding box from the advection

### 4.5.1. Marching Cubes

The first approach is the rasterization of the surface with the Marching Cubes (MC) method [49, 11], see Fig. 4.1a. Surprisingly, one does not need to compute the advected SDF, the MC can work directly on the displacements of the active nodes: The MC triangulation is precomputed from the reference configuration into a static index and a static vertex buffer. But instead of storing the actual position of each vertex, that always lies on an edge of the cells, the indices of the two incident grid nodes plus the interpolation weight is stored in the vertex buffer. During the rendering, the vertex shader reads the current displacements of the two grid nodes from a Shader Storage Buffer and interpolates them to compute the actual position of the vertex in the current timestep. Because no diffusion and advection is needed, this is the fastest rendering method.

### 4.5.2. Slices

Next, the advected SDF can also be visualized as a slice through the volume. The slice can be orientated towards one of the three axis or always facing the camera (see Fig. 4.1c), and the position of the slice can be changed. The mapping from the SDF value to a color is determined by a transfer function. This visualization method is inferior to the other two rendering method in terms of visual perception, and mainly included for debugging purposes.

### 4.5.3. Volume Raycasting

Last, the surface of the advected SDF can be visualized with a ray caster implemented in the Fragment Shader (see Fig. 4.1c). The surface normal is directly given by the gradient of the SDF and a simple Phong shading is used to determine the color. Since the visualized volume is a SDF, the values are the shortest distance to the surface, hence they are used for an adaptive step size, thus greatly decreasing the required number of steps. This visualization method is the slowest, but gives the best results.

## 4.6. Benchmarks

For a comparison of the new grid approach with existing simulations on tetrahedron meshes, we also implemented the above GPU-Matrix Assembly for tetrahedra and compared their performances. All benchmarks were executed on a desktop computer running Windows 10 Home, with an Intel i7-6700 CPU (3.4GHz), 16.0GB RAM and a NVIDIA GeForce GTX 1060 6GB GPU. During the execution of the benchmarks, we measured a constant GPU load of 96% and a memory controller load of 18%. This indicates that the limiting factor are the actual computations and that the memory access patterns are optimized enough to not restrict the performance.

In the first two benchmarks, a cuboid is placed into the world, centering at  $(0, 1, 0)$  with size  $(1.29, 0.4, 0.8)$  and then discretized into grid cells or tetrahedra with a local resolution of  $1/\text{resolution}$ . The resolution is varied over the different test cases. In the first benchmark (Fig. 4.2a,4.2b), the cuboid is fixed with Dirichlet boundaries and then deformed; in the second benchmark (Fig. 4.2c,4.2d), no Dirichlet boundaries are used, instead the object collides with the ground. In all cases, corotation is enabled and 30 timesteps are performed.

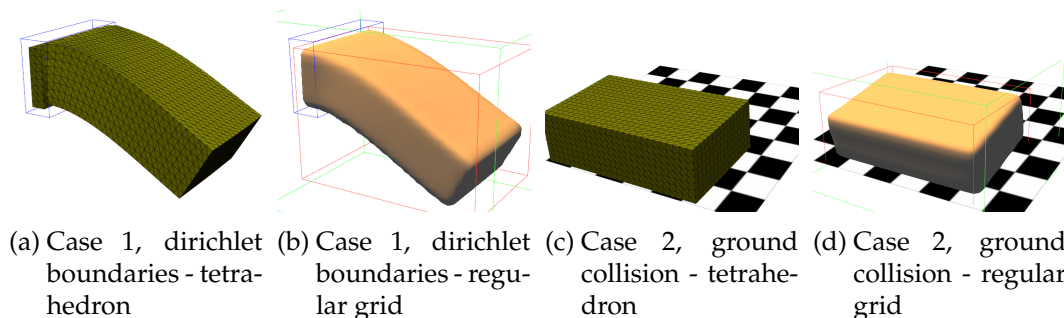


Figure 4.2.: Comparison of the tetrahedron and regular grid simulation on a simple bar

The results of the discretization into tetrahedra and the regular hexahedral grid can be found in Table 4.1. Note how the number of nodes differ from the tetrahedral simulation to the new hexahedral grid simulation. This is due to the partial cells: The grid simulation

#### 4. 3D Implementation with CUDA

Resolution	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Tet: # free nodes	605	1014	1554	2044	2736	3951	5290	6776	8008	9744	12415	15162	18180	20340	23584	28373
Tet: # fixed nodes	25	42	42	112	144	180	220	264	462	468	585	714	720	810	1216	1360
Grid: # free nodes	675	1309	1729	3105	3825	4131	6479	7623	10465	12025	13325	17415	19575	25823	28611	29733
Grid: # empty nodes	5443	6491	8479	9016	11475	13365	14269	16977	18179	22097	24405	25641	29673	29752	35349	39583
Tet: # elements	2496	4500	7128	9828	13524	19968	27216	35340	42840	52272	67392	83148	99792	112056	132300	159744
Grid: # elements	448	960	1296	2464	3072	3328	5400	6400	8976	10368	11520	15288	17248	23040	25600	26624

Table 4.1.: Number of nodes and elements in the different discretizations

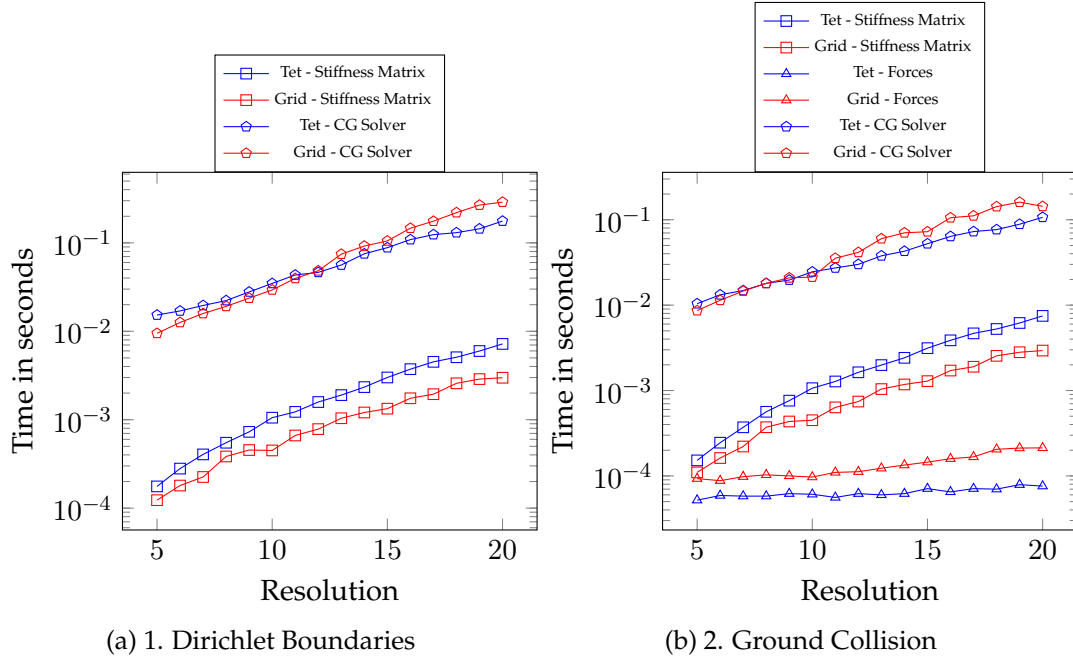


Figure 4.3.: Timings of the individual steps in the cuboid benchmark

resolves the cuboid with partly filled cells and thus also includes nodes that are located outside of the object (nodes of active cells). For the second benchmark without Dirichlet boundaries, the fixed nodes in the tet-simulation become free nodes. For the grid simulation, the number of empty nodes are the nodes that are filled by the diffusion, before the advection step. Furthermore, note that the number of elements in the tet-simulation is five to six times as high as the number of elements in the grid-simulation, respectively. This is because each cube cell is split into six tetrahedra.

Next, Fig. 4.3 shows the timings for the individual steps in the simulation: collision forces (if enabled), matrix assembly and CG solver. For the exact numbers, see Appendix B.5 and B.6. The matrix assembly of the grid simulation is two to three times faster as of the tet simulation. This shows that it is way more advantageous to evaluate fewer but more complicated FEM elements as in the grid simulation, than more but much simpler elements as in the tet simulation. The computation of the collision forces is so fast that the differences in the timings may be influenced by launch overheads in the CUDA calls.



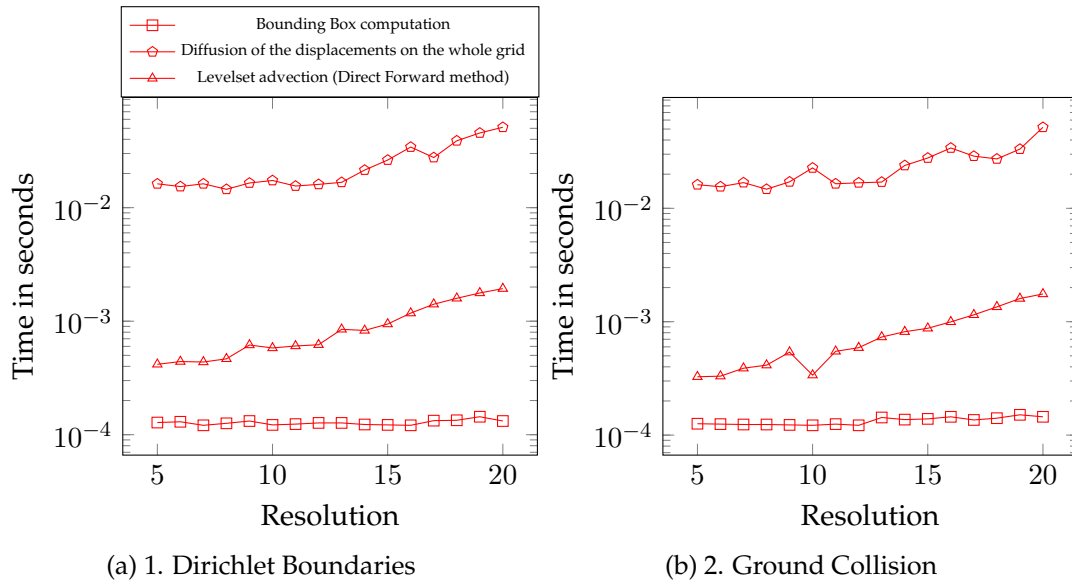


Figure 4.4.: Additional cost introduced by the levelset advection

The performance of the conjugate gradient solver, however, is slower by about 15% on average in the grid simulation than in the tet simulation, even though it requires around 5% fewer iterations until convergence (error below  $10^{-6}$ ). This may be due to the denser stiffness matrix. The grid simulation has about 25 entries per row on average, while the tet simulation has only around 14 entries on average. This linear solve is the dominating factor in the execution of the simulation. The stiffness matrix assembly as described in Sec. 4.4.1 only takes 1 to 5% of the time to solve the linear system.

Furthermore, if the grid simulation should be rendered with the raytracing method, the SDF defining the object has to be advected. The three required steps (diffusion of the displacements over the whole grid, the computation of the new bounding box, the advection using the Direct Forward method) were presented in detail in Sec. 4.4.5. The timings are plotted in Fig. 4.4. One can see that the dominating factor is again the linear solve that is needed in the diffusion step. This solver, however, scales better with a higher resolution since only a boundary around the object of constant size is needed. The degrees of freedom in this linear system scales only quadratically with the voxel size whereas the degrees of freedom in the elasticity solve scales cubically. Therefore, for high resolutions, the full levelset advection only takes around 20% of the total time of the elasticity simulation.



## 5. Inverse Simulation

In the previous part, we described how to perform the elasticity simulation given all initial parameters. But what if not all of those parameters are known? In this part we describe how these unknown parameters can be reconstructed from observations (known outputs) and known parameters (parameters that are not reconstructed like boundary conditions and the global gravity).

### 5.1. Related Work

Solving the inverse problem of a physical simulation arises in many fields. In medical imaging, for example, one is interested in reconstructing the stiffness of organs or tissue from scanned data to detect diseases [23, 87, 70, 3, 34, 66, 65, 10]. Some of these works use probabilistic methods like a Monte-Carlo Markov-Chain simulation in Risholm *et al.* [65] instead of the Adjoint Method presented here. This allows them to avoid the computation of a full gradient and gives a probability distribution over the parameter space, but are computationally very expensive.

In the Computer Graphic community inverse simulations are mostly used to a simple artistic control of animations. The artists wants to specify only a few selected keyframes. The inverse simulation then figures out the external forces and boundary conditions to match the desired keyframes. The corresponding literature is split into works that reconstruct the initial reference position [20, 17], and works that reconstruct external forces like simulated muscles [69] or general forces [6, 7]. The works mentioned above have in common that they take the deformations as input. But in some cases, only observed images are given. This is e.g. handled by Wang *et al.* [83]. Some methods are specialized only on simulations on humans [45], however, they focus more on reconstructing the skeletal motion than on the soft body simulation of the outer tissue. Other methods specialize on thin shell deformation instead of volumetric deformation [9].

Similar methods are also used in controlling fluid animations [82, 51, 75, 53, 61, 62]. We will draw upon these works if they solve problems that also arise in our case.

### 5.2. General Adjoint Method

Before diving into the soft body simulation, we present the adjoint method for a general problem based on the works of Stam [75] and McNamara [51] for controlling fluid animations.

### 5.2.1. Problem Statement

Let  $u \in \mathbb{R}^U$  be the  $U \in \mathbb{N}$  states of the system, the intermediate and output variables. Let  $p \in \mathbb{R}^P$  be the  $P$  control parameters of the system, the input variables. The general problem is then defined by the equation

$$\mathbf{E}(u, p) = \mathbf{0}, \quad \mathbf{E} : \mathbb{R}^U \times \mathbb{R}^P \rightarrow \mathbb{R}^U \quad (5.2.1)$$

that relates the control parameters to the state variables with a problem-specific function  $\mathbf{E}$ .

The goal is to optimize the control parameters with respect to a given cost function

$$J(u, p), \quad J : \mathbb{R}^U \times \mathbb{R}^P \rightarrow \mathbb{R}. \quad (5.2.2)$$

### 5.2.2. Gradient Evaluation

We now want to find the total derivative of the cost with respect to the parameters  $\frac{dJ}{dp}$ . The total derivative can be decomposed as

$$\frac{dJ}{dp} = \left( \frac{\partial J}{\partial u} \right)^T \frac{du}{dp} + \frac{\partial J}{\partial p}. \quad (5.2.3)$$

A remark on the dimensions: If an operation has degrees of freedom in both state and parameters, the states are always indexed by row and the parameters by column. As an example, the above expressions have the following dimensions:

- $\frac{dJ}{dp} \in \mathbb{R}^{1,P}$ : row vector, scalar property derived over the parameters,
- $\frac{\partial J}{\partial u} \in \mathbb{R}^{U,1}$ : column vector, scalar property derived over the states,
- $\frac{du}{dp} \in \mathbb{R}^{U,P}$ : vector property over states derived over the parameters,
- $\frac{\partial J}{\partial p} \in \mathbb{R}^{1,P}$ : row vector, scalar property derived over the parameters.

The above expression contains the total derivative of the states with respect to the parameters  $\frac{du}{dp}$ . We can find an expression for this term by taking the total derivative of the problem  $\mathbf{E}$  with respect to the parameters

$$\frac{d\mathbf{E}}{dp} = \frac{\partial \mathbf{E}}{\partial u} \frac{du}{dp} + \frac{\partial \mathbf{E}}{\partial p} = \mathbf{0}. \quad (5.2.4)$$

In order to solve the equation above, we introduce some notation:

$$A := \frac{\partial \mathbf{E}}{\partial u} \in \mathbb{R}^{U,U} \quad (5.2.5)$$

$$X := \frac{du}{dp} \in \mathbb{R}^{U,P} \quad (5.2.6)$$

$$f_i := -\frac{\partial \mathbf{E}}{\partial p_i} \in \mathbb{R}^{U,1}, i = 1, \dots, P \quad (5.2.7)$$

$$F := (f_1, \dots, f_P) = -\frac{\partial \mathbf{E}}{\partial p} \in \mathbb{R}^{U,P} \quad (5.2.8)$$

$$r^T := \frac{\partial J}{\partial p} \in \mathbb{R}^{1,P} \quad (5.2.9)$$

$$g := \frac{\partial J}{\partial u} \in \mathbb{R}^{U,1}. \quad (5.2.10)$$

Then we can rewrite Eq. (5.2.4) as  $AX = F$ . Hence, these are  $P$  linear systems with  $U$  unknowns each. While this might be feasible for a small number of control parameters, it does not scale well with an increasing number of parameters.

In the end, we are only interested in the result of the product  $\left(\frac{\partial J}{\partial u}\right)^T \frac{du}{dp} = g^T X$ . The Adjoint Method provides a different way to evaluate that term. Instead of solving  $AX = F$  we solve for the adjoint problem

$$A^T y = g \quad (5.2.11)$$

with the solution  $y \in \mathbb{R}^{U,1}$ . This is only one linear system of equations.

This allows us to write the product  $g^T X$  as

$$g^T X = (A^T y)^T X = y^T AX = y^T F = (y^T f_1, \dots, y^T f_P). \quad (5.2.12)$$

Hence, in total we only have to solve one linear system for  $y$  and then for each parameter  $i$  compute the vector-vector multiplication  $y^T f_i$  to obtain the final product  $y^T F$ .

Therefore, the Adjoint Method can be summarized in the following two steps:

- Solve  $A^T y = g$  for  $y$
- Sum over all parameters and priors  $\frac{dJ}{dp} = y^T F + r^T$ .

### 5.2.3. Computing the Adjoint of the Problem

For  $\circ$  being an operation or a variable in the forward pass (contained in  $\mathbf{E}$ ), let  $\hat{\circ}$  denote the associated adjoint operation or variable.

Luckily, the adjoint  $A^T y = g$  can be computed mechanically. Assume that the operation  $\mathbf{E}$  is indeed composed out of a sequence of instructions

$$E_1, E_2, E_3, \dots, E_n$$

that generate the sequence of states  $u_1, u_2, \dots, u_n$  (combined in  $u$  above), then the adjoint of the whole process (called backward pass) can be built out of the adjoints of the individual instructions in reversed order

$$\hat{E}_n, \hat{E}_{n-1}, \dots, \hat{E}_2, \hat{E}_1.$$

### 5.2.3.1. Matrix view

Given a decomposition of the operation  $\mathbf{E}$  into a sequence of instructions as described above, the matrices involved in solving the adjoint problem take the special form of block-triangular matrices:

$$\mathbf{E} = \begin{pmatrix} E_1(u_1, \{p\}) \\ E_2(u_2, \{u_1, p\}) \\ \vdots \\ E_n(u_n, \{u_1, \dots, u_{n-1}, p\}) \end{pmatrix} \quad (5.2.13)$$

$$\hat{u} = y = \begin{pmatrix} \hat{u}_1 \\ \hat{u}_2 \\ \vdots \\ \hat{u}_n \end{pmatrix}, \quad \frac{\partial J}{\partial u} = g = \begin{pmatrix} \frac{\partial J}{\partial u_1} \\ \frac{\partial J}{\partial u_2} \\ \vdots \\ \frac{\partial J}{\partial u_n} \end{pmatrix} \quad (5.2.14)$$

$$A = \frac{\partial \mathbf{E}}{\partial u} = \begin{pmatrix} \frac{\partial E_1}{\partial u_1} & 0 & \dots & 0 \\ \frac{\partial E_2}{\partial u_1} & \frac{\partial E_2}{\partial u_2} & 0 & \vdots \\ \vdots & \vdots & \ddots & 0 \\ \frac{\partial E_n}{\partial u_1} & \frac{\partial E_n}{\partial u_2} & \dots & \frac{\partial E_n}{\partial u_n} \end{pmatrix} \quad (5.2.15)$$

$$A^T = \begin{pmatrix} \left(\frac{\partial E_1}{\partial u_1}\right)^T & \left(\frac{\partial E_2}{\partial u_1}\right)^T & \dots & \left(\frac{\partial E_n}{\partial u_1}\right)^T \\ 0 & \left(\frac{\partial E_2}{\partial u_2}\right)^T & \dots & \left(\frac{\partial E_n}{\partial u_2}\right)^T \\ \vdots & 0 & \ddots & \vdots \\ 0 & \dots & 0 & \left(\frac{\partial E_n}{\partial u_n}\right)^T \end{pmatrix}. \quad (5.2.16)$$

Hence,  $A^T y = g$  can be solved by backward substitution

$$\begin{aligned}
 \left(\frac{\partial E_n}{\partial u_n}\right)^T \hat{u}_n &= \frac{\partial J}{\partial u_n} \\
 \left(\frac{\partial E_{n-1}}{\partial u_{n-1}}\right)^T \hat{u}_{n-1} &= \frac{\partial J}{\partial u_{n-1}} - \left(\frac{\partial E_n}{\partial u_{n-1}}\right)^T \hat{u}_n \\
 &\vdots \\
 \left(\frac{\partial E_j}{\partial u_j}\right)^T \hat{u}_j &= \frac{\partial J}{\partial u_j} - \sum_{i=j+1}^n \left(\frac{\partial E_i}{\partial u_j}\right)^T \hat{u}_i \\
 &\vdots \\
 \left(\frac{\partial E_1}{\partial u_1}\right)^T \hat{u}_1 &= \frac{\partial J}{\partial u_1} - \sum_{i=2}^n \left(\frac{\partial E_i}{\partial u_1}\right)^T \hat{u}_i.
 \end{aligned} \tag{5.2.17}$$

A more algorithmic view is shown in Fig. 5.1:

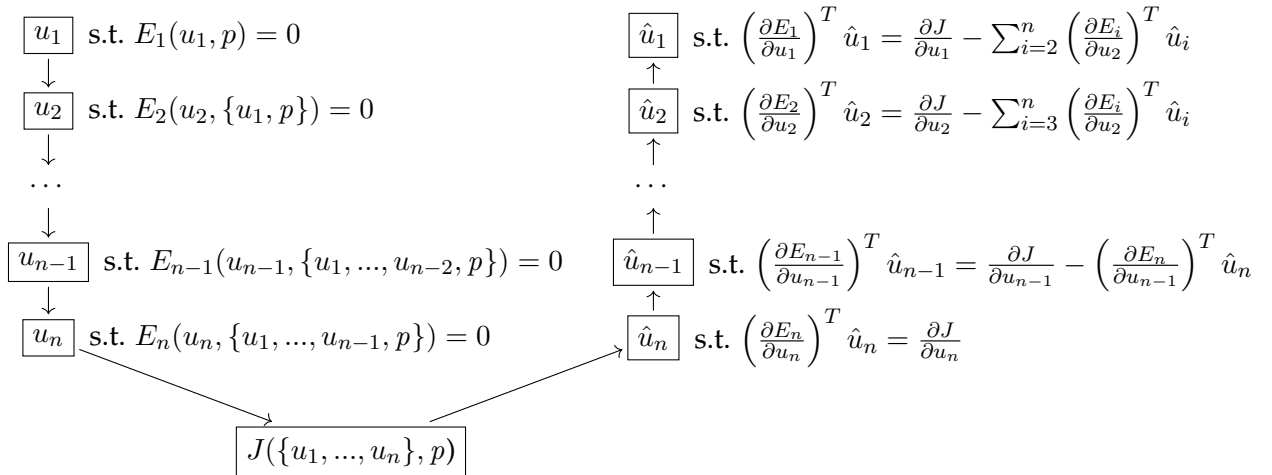


Figure 5.1.: Execution of the forward and adjoint pass

### 5.2.3.2. Instruction view

Instead of using matrix expressions for computing the adjoint of the operation  $E$ , one can also view it as a sequence of primitive instructions  $u \leftarrow f(u, p)$  and compute the adjoint for them. This view on the Adjoint Method has the advantage of being closer to the actual code and allows to mechanically translate also loops and if-statements. We will now present the adjoint code of the most common primitive instructions that allow us to mechanically compute the adjoint code of any forward code, see [75]. A comparison of the Matrix View and the Instruction View together with examples is presented in the next Section 5.2.4.

- Arbitrary function  $E : x \leftarrow f(x, y_1, \dots, y_m)$ , possibly vector valued  
The adjoint code  $\hat{E}$  then takes the form

$$\begin{aligned}
 \hat{y}_m &\leftarrow \hat{y}_m + \left( \frac{\partial f(x, y_1, \dots, y_m)}{\partial y_m} \right)^T \hat{x} \\
 &\vdots \\
 \hat{y}_1 &\leftarrow \hat{y}_1 + \left( \frac{\partial f(x, y_1, \dots, y_m)}{\partial y_1} \right)^T \hat{x} \\
 \hat{x} &\leftarrow \left( \frac{\partial f(x, y_1, \dots, y_m)}{\partial x} \right)^T \hat{x}.
 \end{aligned} \tag{5.2.18}$$

In particular, note that if the function  $f$  does not take  $x$  as input, the adjoint  $\hat{x}$  has to be set to zero in order to avoid that it participates in later adjoint instructions, because previous forward instructions modifying  $x$  were overwritten by the current instruction.

The adjoint variables  $\hat{y}_i, \hat{x}$  are initialized with the negative derivative of the cost function:  $\hat{y}_i = -\frac{\partial J}{\partial y_i}, \hat{x} = -\frac{\partial J}{\partial x}$ .

- Loops:

$$\begin{array}{ll}
 \text{for } i = 1, \dots, n \text{ do} & \Rightarrow \hat{E} : \text{for } i = n, \dots, 1 \text{ do} \\
 E : E_i & \hat{E}_i \\
 \text{end for} & \text{end for}
 \end{array}$$

The loops are processed in reverse order.

- If-Statements:

$$\begin{array}{ll}
 \text{if } C \text{ then} & \text{if } C \text{ then} \\
 E_1 & \hat{E}_1 \\
 E : \text{else} & \Rightarrow \hat{E} : \text{else} \\
 E_2 & \hat{E}_2 \\
 \text{end if} & \text{end if}
 \end{array}$$

No change to the order of execution, the value of the condition  $C$  has to be stored from the forward pass or reevaluated.

### 5.2.4. Examples

In the previous section we presented two views on computing the adjoint problem. Here, we will show by some examples how they are used and in which points they show differences.

- Simple function

$$\begin{aligned}
 u_2 = f(u_1, \{p_1, p_2\}) &:= u_1^3 + \sin(p_1^2)p_2 \\
 &\Downarrow \\
 E_2(u_2, \{u_1, p_1, p_2\}) &:= u_2 - f(u_1, \{p_1, p_2\}) = 0
 \end{aligned}$$



The external input to the adjoint code is the gradient with respect to the output  $u_2$ ,  $g = \frac{\partial J}{\partial u_2}$ .

– Matrix view:

$$A = \frac{\partial E_2}{\partial u_2} = 1, F = -\frac{\partial E_2}{\partial p_1, p_2} = (2p_1 \cos(p_1^2)p_2, \sin(p_2^2))$$

$$\text{Solve } A^T \hat{u}_2 = g \rightarrow \hat{u}_2 = g.$$

$$\text{Return } \left(\frac{\partial J}{\partial u}\right)^T \frac{du}{dp} = \hat{u}_2^T F = g^T (2p_1 \cos(p_1^2)p_2, \sin(p_2^2)) \in \mathbb{R}^{1,2}.$$

– Instruction view:

$$\text{Initialize } \hat{u}_2 = g.$$

If  $\hat{u}_1, \hat{p}_1, \hat{p}_2$  have not been used before, initialize them with zero.

Back-propagate

$$\hat{u}_1 += 3u_1^2 \hat{u}_2,$$

$$\hat{p}_1 += 2p_1 \cos(p_1^2)p_2 \hat{u}_2,$$

$$\hat{p}_2 += \sin(p_2^2) \hat{u}_2$$

Then the gradient is directly given as:

$$\left(\frac{\partial J}{\partial u}\right)^T \frac{du}{dp} = (\hat{p}_1^T, \hat{p}_2^T) = g^T (2p_1 \cos(p_1^2)p_2, \sin(p_2^2)) \in \mathbb{R}^{1,2}.$$

• Linear system

$$u = f(p) := M^{-1}Bp$$

$$\Downarrow$$

$$E(u, p) = Mu - Bp = 0$$

with  $u \in \mathbb{R}^n, p \in \mathbb{R}^m, M \in \mathbb{R}^{n,n}, B \in \mathbb{R}^{n,m}$ . The external input to the adjoint code is the gradient with respect to the output  $u_2, g = \frac{\partial J}{\partial u_2} \in \mathbb{R}^{n,1}$ .

– Matrix view:

$$A = \frac{\partial E}{\partial u} = M, F = -\frac{\partial E}{\partial p} = B$$

$$\text{Solve } A^T \hat{u} = g \rightarrow \hat{u} = M^{-T}g$$

$$\text{Return } \left(\frac{\partial J}{\partial u}\right)^T \frac{du}{dp} = \hat{u}^T F = (M^{-T}g)^T B \in \mathbb{R}^{1,m}.$$

– Instruction view:

$$\text{Initialize } \hat{u} = g.$$

If  $\hat{p}$  has not been used before, initialize it with zero.

$$\text{Back-propagate } \hat{p} += \frac{\partial f^T}{\partial p} \hat{u} = (M^{-1}B)^T \hat{u} = B^T M^{-T} \hat{u}$$

$$\text{Return } \left(\frac{\partial J}{\partial u}\right)^T \frac{du}{dp} = \hat{p}^T = (B^T M^{-T}g)^T = (M^{-T}g)^T B \in \mathbb{R}^{1,m}.$$

Both approaches lead to the same result, as expected, but their usage of the "adjoint variables" is different. The Matrix view computes the adjoint variables ( $y$ ) and the derivative with respect to the controls ( $F$ ) separately, while the Instruction view directly evaluates the product ( $y^T F$ ) into the adjoint variables of the control parameters. They are equal if

the operation  $E_i$  has the form of a function  $u_i = f_i(u_{i-1})$  that only takes the previous state as input and has no dependency to parameters that are optimized for.

The advantage of the Instruction view is a simplified model because the matrix  $F$  does not have to be computed separately. Moreover, the memory footprint might be lower because  $y$  and  $F$  don't need to be stored explicitly, only their product in the adjoint variables of the controls. The disadvantage is that it is harder to add more control variables later on.

We will switch between the matrix view and instruction view in the adjoint code based on which gives the simpler expression. Generally, we'll use the matrix view for the "big picture" to allow a simple plug-in of different derivatives of control parameters and for expressions that involve the solution of system of linear equations, while we use the instruction view for helper functions. Table 5.1 contains a reference of commonly used expressions and their adjoint version that are used later on.

### 5.2.5. On Testing Adjoint Code

The meaning of the adjoint variables is not always obvious, so we'll present a simple randomized method to validate the adjoint code. Suppose we have a forward function

$$y = f(x) , x \in \mathbb{R}^n , y \in \mathbb{R}^m$$

and have written the adjoint function (that might need the input and output of the forward function again)

$$\hat{x} = \hat{f}(\hat{y}, x, y) = \left( \frac{\partial f}{\partial x} \right)^T \hat{y} , \hat{y} \in \mathbb{R}^m , \hat{x} \in \mathbb{R}^n .$$

Then for sufficiently many trials, generate a random input  $x_{\text{rnd}}$  and a random adjoint output  $\hat{y}_{\text{rnd}}$  and compare the adjoint code with a numerical derivative using the following equality:

$\forall i \in \{1, \dots, n\} :$

$$\mathbf{e}_i^T \hat{f}(\hat{y}_{\text{rnd}}) = \mathbf{e}_i^T \left( \frac{\partial f}{\partial x} \right)^T \hat{y}_{\text{rnd}} \stackrel{!}{=} \left( \frac{\partial f}{\partial x} \mathbf{e}_i \right)^T \hat{y}_{\text{rnd}} = \left( \lim_{\epsilon \rightarrow 0} \frac{f(x_{\text{rnd}} + \epsilon \mathbf{e}_i) - f(x_{\text{rnd}})}{\epsilon} \right)^T \hat{y}_{\text{rnd}} .$$

Table 5.1.: Reference of basic operations and their adjoint [63, 75].

All assignments in the adjoint are to be treated additively, e.g.  $\hat{A} += \hat{X}$  in the first entry

	Constraints	Forward Code	Adjoint Code
a)	$A, B, X \in \mathbb{R}^{n \times m}$	$X = A + B$	$\hat{A}, \hat{B} = \hat{X}$
b)	$A, X \in \mathbb{R}^{n \times m}, \alpha \in \mathbb{R}$	$X = \alpha A$	$\hat{A} = \alpha \hat{X}$ $\hat{\alpha} = \sum_{i,j} A_{ij} \hat{X}_{ij} = \text{vec}(A) \bullet \text{vec}(\hat{X})$
c)	$x, y \in \mathbb{R}^n, \alpha \in \mathbb{R}$	$\alpha = x^T y = x \bullet y$	$\hat{x} = \hat{\alpha} y$ $\hat{y} = \hat{\alpha} x$
d)	$A \in \mathbb{R}^{n \times k}, B \in \mathbb{R}^{k \times m}, X \in \mathbb{R}^{n \times m}$	$X = AB$	$\hat{A} = \hat{X} B^T$ $\hat{B} = A^T \hat{X}$
e)	$x, b \in \mathbb{R}^n, A \in \mathbb{R}^{n \times n}$	Solve $Ax = b$ for $x$	Input: $\hat{x}$ Solve $A^T \hat{b} = \hat{x}$ for $\hat{b}$ $\hat{A} = \hat{b} x^T$
f)	$A \in \mathbb{R}^{n \times m}, B \in \mathbb{R}^{m \times m}, X \in \mathbb{R}^{n \times n}$	$X = ABA^T$	$\hat{A} = \hat{X}^T AB^T + \hat{X} AB$ $\hat{B} = A^T \hat{X} A$
g)	$A, B \in \mathbb{R}^{n \times n}, x, y \in \mathbb{R}^n$	$y = AB(A^T x - x)$	$\hat{A} = x \hat{y}^T AB + \hat{y} x^T AB^T - \hat{y} x^T B$ $\hat{B} = A^T \hat{y} (A^T x - x)^T$ $\hat{x} = \hat{y} AB (A^T - \mathbf{1})$
h)	$A, B, X \in \mathbb{R}^{n \times n}$	$X = AB^{-1}$	$\hat{A} = \hat{X} B^{-T}$ $\hat{B} = -B^{-T} A^T \hat{X} B^{-T}$
i)	$A, X \in \mathbb{R}^{n \times n}$	$X = A^{-T}$	$\hat{A} = -X \hat{X}^T X$
j)	$n \in \mathbb{N}$ operations $E_i$	for $i = 1, \dots, n$ do $E_i$ end for	for $i = n, \dots, 1$ do $\hat{E}_i$ end for
k)	operations $E_1, E_2$	if $C$ $E_1$ else $E_2$ end if	if $C$ $\hat{E}_1$ else $\hat{E}_2$ end if

### 5.3. Adjoint Method for Soft Body Simulations

In this section we present the adjoint version of the elasticity simulation on the grid simulation. First, as a recapitulation, the full forward simulation is given in Alg. 5.1.

---

**Algorithm 5.1** Full forward simulation for the regular grid discretization

---

- Input:** Reference configuration  $\phi^{(0)}$ , elasticity parameters  $(\lambda, \mu, m, \dots)$
- 1: Precompute the lumped mass matrix  $M$  ▷ See Sec. 4.4.2 and Eq. (2.5.14)
  - 2: Precompute the body forces  $f_{\text{body}}$  ▷ See Sec. 4.4.2 and Eq. (2.5.16)
  - 3: Initialize the states  $u^{(0)}, \dot{u}^{(0)}$
  - 4: **for**  $t=1, \dots, T$  **do**
  - 5:   Compute collision forces  $f_{\text{collision}}^{(t)}$  ▷ See Sec. 2.8
  - 6:   Compute stiffness matrix  $K^{(t)}$  and corotation forces  $f_{\text{rot}}^{(t)}$  ▷ See Sec. 4.4.1
  - 7:   Final force vector  $f^{(t)} = f_{\text{body}} + f_{\text{collision}}^{(t)} + f_{\text{rot}}^{(t)}$
  - 8:   Compute Newmark matrices  $A^{(t)}, b^{(t)}$  ▷ See Sec. 4.4.3
  - 9:   Solve for the next displacements  $A^{(t)}u^{(t)} = b^{(t)}$  ▷ See Sec. 4.4.4
  - 10:   Compute the new velocity  $\dot{u}^{(t)}$  ▷ See Eq. (2.6.3)
  - 11:   Diffuse the displacements over the whole grid  $u_{\text{grid}}^{(t)}$  ▷ See Sec. 4.4.5
  - 12:   Compute the advected levelset  $\phi^{(t)}$  ▷ See Sec. 4.4.5
  - 13: **end for**
- 

In the next sections we'll present the possible choices for a cost function, followed by the adjoint versions of the individual steps of Alg. 5.1.

### 5.4. Cost Function

In inverse problems, the cost function  $J(u) \rightarrow \mathbb{R}$  measures the difference between the current solution and the target solution.

Furthermore, the cost function can also be used to penalize violations in the physical model as a prior. These weak physical constraints are used e.g. in Pan and Manocha [62]. In our case, the physical constraints are enforced strongly by the forward and adjoint simulation. Hence the cost function consists solely of the difference of the current solution to the target solution, plus optional priors on the model parameters.

For the grid simulation, there are several different ways to define a cost function, as presented in the following sections.

### 5.4.1. Differences on the Displacements

The simplest cost function penalizes only the differences in the displacements and velocities on the active cells:

$$\begin{aligned}
 J(u) &:= \sum_{t=1}^T \frac{1}{2} w_t \left\| u^{(t)} - u_{\text{ref}^{(t)}} \right\|^2 + \frac{1}{2} v_t \left\| \dot{u}^{(t)} - \dot{u}_{\text{ref}^{(t)}} \right\|^2 \\
 g_{u^{(t)}} &= \frac{\partial J}{\partial u^{(t)}} := w_t \left( u^{(t)} - u_{\text{ref}}^{(t)} \right) \\
 g_{\dot{u}^{(t)}} &= \frac{\partial J}{\partial \dot{u}^{(t)}} := w_t \left( \dot{u}^{(t)} - \dot{u}_{\text{ref}}^{(t)} \right).
 \end{aligned} \tag{5.4.1}$$

This cost function is often used in the inverse elasticity problems on tetrahedral meshes where the target configuration is defined by the positions of the mesh vertices. It is used in slightly modified versions e.g. in Pan and Manocha [62] or Coros *et al.* [20].

For the grid simulation, we observe the following:

Advantage: No advection and diffusion steps are needed, only the displacement from the elasticity simulation is needed.

Disadvantage: This cost function requires that the displacements are directly observed, which is e.g. not the case if the target is based on 3D reconstructions.

### 5.4.2. Differences on the Levelset

Next, we can define the cost function as the squared differences on the whole levelset:

$$\begin{aligned}
 J(\phi) &:= \sum_{t=1}^T \frac{1}{2} w_t \left\| W_t \odot \left( \phi^{(t)} - \phi_{\text{ref}}^{(t)} \right) \right\|^2 \\
 g_{\phi^{(t)}} &= \frac{\partial J}{\partial \phi^{(t)}} := w_t W_t \odot \left( \phi^{(t)} - \phi_{\text{ref}}^{(t)} \right).
 \end{aligned}$$

It penalizes differences between the current level set  $\phi^{(t)}$  and the reference level set  $\phi_{\text{ref}}^{(t)}$  at every timestep. The scalar weighting factors per frame  $w_t$  are as in the cost function for the mesh. In addition, we introduce the weighting matrix  $W$  that additionally weights the contribution of the individual nodes in the grid to the cost. For simple scenarios, it can be set to one everywhere, but it might be used to ignore areas with missing data as well.

This choice of the cost function, however, is in general not optimal. It penalizes differences in the values of the signed distance function, while a better approach should penalize differences in the volume represented by the SDF. As an example, suppose we have a region of the SDF with values of around 10 in the current simulation while the reference SDF has values of around 15. This would introduce a high cost, while the shape of the object represented by the SDF is completely unaffected by that change. A value of 10 is outside and far away from the border.

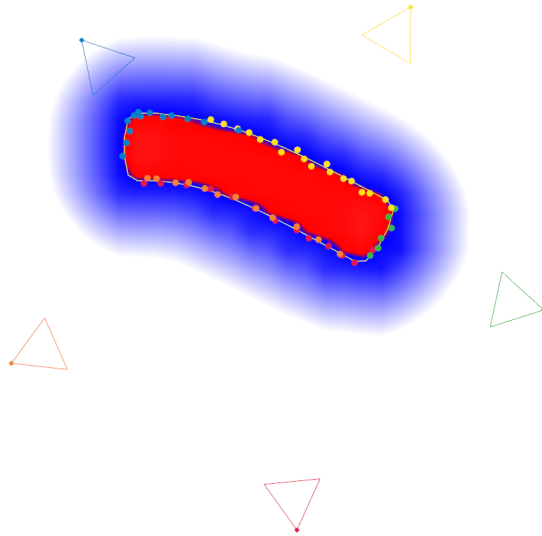


Figure 5.2.: Simulated observations (small colored dots) of a deformed object

A better approach was proposed by McNamara *et al.* [51]. They use a smooth differentiable function  $\gamma$  defined as

$$\gamma(x) := \frac{2}{\pi} \tan^{-1}(x), \quad (5.4.2)$$

that is applied to the SDFs before the comparison:

$$J(\phi) := \sum_{t=1}^T \frac{1}{2} w_t \left\| W_t \odot \left( \gamma(\phi^{(t)}) - \gamma(\phi_{\text{ref}}^{(t)}) \right) \right\|^2. \quad (5.4.3)$$

The function  $\gamma$  smoothly truncates positive values to +1 and negative values to -1 and thus penalizes only differences near to the zero level.

For the grid simulation, we observe the following:

Advantage: Direct comparison with the observed level set.

Disadvantage: Adjoint code of the advection and the optional SDF reinitialization is numerically very unstable, see Sec. 5.5.1.

### 5.4.3. Distance to Point Clouds

The third alternative for the cost function uses sparse point clouds of the object surface as observation. When RGB-D cameras like the Kinect camera are used to observe the reference object, they don't provide directly a dense level set of the whole scene, but rather sparse points on the boundary. These sparse points can be directly used in this new cost function. An example of these sparse observations in 2D can be seen in Fig. 5.2.

Let  $N^{(t)}$  be the number of observations in timestep  $t$  and let  $\mathbf{x}_{t,i}$  be the positions of these observations in space. Since the camera observes points on the surface of the object, the value of the SDF should be zero at these observations, hence

$$J(\phi) := \sum_{t=1}^T \sum_{i=0}^{N^{(t)}} w_{t,i} \frac{1}{2} \left( \phi^{(t)}(\mathbf{x}_{t,i}) \right)^2. \quad (5.4.4)$$

We found a way to express  $\phi^{(t)}(\mathbf{x}_{t,i})$  analytically with  $\phi^{(0)}$  (the SDF of the reference configuration) and  $u^{(t)}$  (the current displacements) without advecting the whole SDF, see below. This allows us to use observed depth images as reference in a computationally cheap way.

Advantage: Uses directly the camera observations, no global levelset advection needed.

Disadvantage: Quality of the gradient depends on how well the observations cover the surface of the object.

For an efficient computation of  $\phi^{(t)}(\mathbf{x}_{t,i})$ , we require two assumptions. First we assume that the displacements  $u^{(t)}$  from the elasticity simulation do not destroy the signed distance property of  $\phi$ , then we can represent  $\phi^{(t)}$  as

$$\phi^{(t)}(\mathbf{x}) \approx \phi^{(0)}\left(\mathbf{x} + u^{(t)}(\mathbf{x})^{-1}\right). \quad (5.4.5)$$

This approximation is best (up to equality) near the boundary of the object but grows worse with increasing distance. The full advection solves this issue with a reinitialization step.

The difficulty now arises from the fact that we need the inverse displacement and can't just use the negative displacement, see Sec. 2.5.2.5 for a detailed analysis of this problem.

Assume that we know the point  $\mathbf{x}'$  with  $\mathbf{x} = \mathbf{x}' + u^{(t)}(\mathbf{x}')$  (and hence  $\mathbf{x}' = \mathbf{x} + u^{(t)}(\mathbf{x})^{-1}$ ). This point is located in some cell  $i, j, k$ . Let  $x'_{i,j}$  to  $x'_{i+1,j+1,k+1}$  be the reference location of the eight corners of the cell (located on a regular grid) and let  $x_{i,j,k}$  to  $x_{i+1,j+1,k+1}$  be the displaced location of these eight corners. Then the location of point  $\mathbf{x}'$  is computed by a trilinear interpolation of the eight reference corner locations with the interpolation weights  $\alpha, \beta, \gamma$ .

The second assumption is that these interpolation weights within the cell don't change during the advection. In other words,  $\mathbf{x}'$  is interpolated between  $x'_{i,j,k}$  to  $x'_{i+1,j+1,k+1}$  with the same weights as  $\mathbf{x}$  is interpolated between  $x_{i,j}$  to  $x_{i+1,j+1,k+1}$  (see Fig. 5.3). Hence we can use the interpolation weights from the interpolation of the position also to interpolate the SDF values. This allows us to formulate the following algorithm in order to compute  $\phi^{(t)}(\mathbf{x})$  (Alg. 5.2):

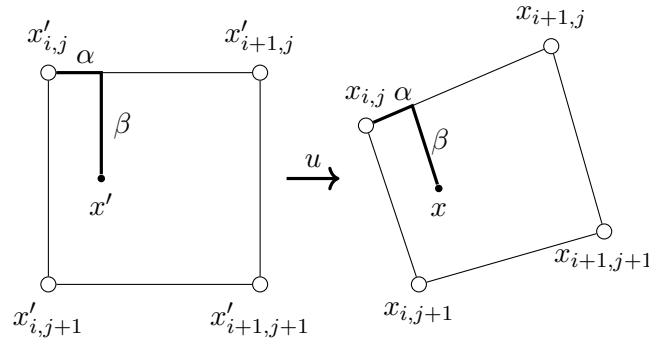


Figure 5.3.: Interpolation on the reference and deformed configuration

---

**Algorithm 5.2** Compute  $\phi^{(t)}(\mathbf{x})$  based on  $\phi^{(0)}$  and  $u^{(t)}$

---

**Input:** The observed point  $\mathbf{x}$

- 1: **for each** cell  $i, j, k$  **do**
  - 2:   Solve  $\mathbf{x} = \text{interpolate}(x_{i,j}, \dots, x_{i+1,j+1,k+1}; \alpha, \beta, \gamma)$  for  $\alpha, \beta, \gamma$
  - 3:   **if**  $(\alpha, \beta, \gamma) \notin (0, 1)^3$  **then continue**  $\triangleright \mathbf{x}$  is not in the current cell
  - 4:   **return**  $\phi^{(t)}(\mathbf{x}) = \text{interpolate}(\phi_{i,j}^{(0)}, \dots, \phi_{i+1,j+1,k+1}^{(0)}; \alpha, \beta, \gamma)$
  - 5: **end for**
- 

For an efficient implementation of Alg. 5.2 with observations from camera images, it is better to flip the loops: Loop over the cells in parallel, project the deformed cell into camera space and check only the pixels that are covered by this projection.

The key step in Alg. 5.2 is solving for the interpolation weights of the trilinear interpolation. Recall that the standard trilinear interpolation is given as

$$\begin{aligned} x &= (1 - \alpha)(1 - \beta)(1 - \gamma)x_1 + \alpha(1 - \beta)(1 - \gamma)x_2 + \dots + \alpha\beta\gamma x_8 \\ &= z_1 + \alpha z_2 + \beta z_3 + \gamma z_4 + \alpha\beta z_5 + \alpha\gamma z_6 + \beta\gamma z_7 + \alpha\beta\gamma z_8. \end{aligned} \tag{5.4.6}$$

If  $x_1$  to  $x_8$  (and hence the derived  $z_1$  to  $z_8$ ) are 3D vectors, as it is the case for the corner positions, this is a system of three equations of first order polynomials in three variables  $\alpha, \beta, \gamma$ .

This system of equations can have multiple solutions but at most one solution that lies within the cell ( $\alpha, \beta, \gamma \in (0, 1)$ ). In the 2D case, an explicit solution can be found. This is, however, in general not possible in 3D. Luckily, a simple Newton iteration converges within just a few iterations to the solution inside the cell with the start values  $\alpha = \beta = \gamma = 0.5$ , if such a solution exists. If not, the Newton iteration might not find a solution outside of the cell, but it will produce a value that lies outside the cell.

The adjoint of Alg. 5.2 is simpler than the forward problem: From the forward problem, the index of the cell that contains  $\mathbf{x}$  is known, so there is no need to search that cell again.



Next, the adjoint of the trilinear interpolation with respect to the interpolation weights is trivial to derive and omitted here. The adjoint of the inverse of the interpolation (solving for the weights) is more challenging. We could mechanically compute the adjoint of the Newton iteration, but it is much simpler if we look at that problem as a whole and apply the Matrix View (see Sec. 5.2.3.1). In 3D we obtain:

$$\begin{aligned}
 \mathbf{E}(u = \{\alpha, \beta, \gamma\}, p = \{z_1, \dots, z_8\}) \\
 &:= z_1 + \alpha z_2 + \beta z_3 + \gamma z_4 + \alpha\beta z_5 + \alpha\gamma z_6 + \beta\gamma z_7 + \alpha\beta\gamma z_8 - x = \mathbf{0}, \\
 A = \frac{\partial \mathbf{E}}{\partial u} &= \begin{pmatrix} | & | & | & | & | & | & | & | \\ z_2 + \beta z_5 + \gamma z_6 + \beta\gamma z_8 & z_3 + \alpha z_5 + \gamma z_7 + \alpha\gamma z_8 & z_4 + \alpha z_6 + \beta z_7 + \alpha\beta z_8 & & & & & \\ | & | & | & | & | & | & | & | \end{pmatrix}, \\
 \text{solve } A^T \begin{pmatrix} \alpha' \\ \beta' \\ \gamma' \end{pmatrix} &= \begin{pmatrix} \hat{\alpha} \\ \hat{\beta} \\ \hat{\gamma} \end{pmatrix}, \hat{\alpha}, \hat{\beta}, \hat{\gamma} \text{ comes from the adjoint of the interpolation of } \phi, \\
 F = \frac{\partial \mathbf{E}}{\partial p} &= - \begin{pmatrix} | & | & | & | & | & | & | & | \\ 1 & \alpha & \beta & \gamma & \alpha\beta & \alpha\gamma & \beta\gamma & \alpha\beta\gamma \end{pmatrix} \in \mathbb{R}^{3,8}, \\
 \Rightarrow (\hat{z}_1, \dots, \hat{z}_8) &= (\alpha', \beta', \gamma') F.
 \end{aligned}$$

Observe that the matrix  $A$  above is exactly the same matrix that appears also in the Newton iteration.

Last, the adjoint values  $\hat{z}_1, \dots, \hat{z}_8$  have to be added to the adjoint values of the displacements on the grid  $\hat{u}_{i,j,k}, \dots, \hat{u}_{i+1,j+1,k+1}$ . Therefore, where the previous two cost functions provided gradients of the displacements on the active nodes, this cost function gives gradients on the whole grid. These are then mapped back to the active nodes with the adjoint of the displacement diffusion, see Sec. 5.5.2.

## 5.5. Adjoint of the Individual Steps

For the main adjoint problem, we trace the derivatives back in time. For that, every step of algorithm Alg. 5.1 is replaced by its adjoint version and the order of the steps is inverted. This is outlined in Alg. 5.3. The details of every step are presented in the next sections.

For the computation of the adjoint simulation, the precomputed mass  $M$  and body force  $f_{\text{body}}$ , as well as  $u^{(t)}, \dot{u}^{(t)}, K^{(t)}, A^{(t)}$  are needed for every timestep.

In the adjoint simulation, we could save all adjoint variables and later compute the gradients of the parameters separately. This is the computation of  $y^T F$  in the classic adjoint method in Matrix View. This approach, however, is very memory intensive. Therefore, we update the adjoint/gradient of the parameters on the fly in an Instruction View fashion, whenever they appear in the equations. This allows us to reuse the same memory for the adjoint variables of the states (like displacement and intermediate matrices) in every timestep.

---

**Algorithm 5.3** Full adjoint/backward simulation for the regular grid discretization

---

**Input:** All necessary intermediate results from the forward simulation

- 1: The adjoint variables  $\hat{u}^{(t)}$ ,  $\hat{u}^{(t)}$  and  $\hat{u}_{\text{grid}}^{(t)}$  are initialized with the gradients from the cost functions
  - 2: All other adjoint variables are initialized with zero
  - 3: **for**  $t=T, \dots, 1$  **do**
  - 4:   Adjoint of the advection  $\hat{\phi}^{(t)} \rightarrow \hat{u}_{\text{grid}}^{(t)}$  ▷ See Sec. 5.5.1
  - 5:   Adjoint of the displacement diffusion  $\hat{u}_{\text{grid}}^{(t)} \rightarrow \hat{u}^{(t)}$  ▷ See Sec. 5.5.2
  - 6:   Adjoint of the new velocity computation  $\hat{u}^{(t)} \rightarrow \hat{u}^{(t)}, \hat{u}^{(t-1)}, \hat{u}^{(t)}$  ▷ See Sec. 5.5.3
  - 7:   Adjoint of the displacement solve  $\hat{u}^{(t)} \rightarrow \hat{A}^{(t)}, \hat{b}^{(t)}$  ▷ See Sec. 5.5.3
  - 8:   Adjoint of the Newmark matrices  $\hat{A}^{(t)}, \hat{b}^{(t)} \rightarrow m, \hat{f}^{(t)}, \hat{K}^{(t)}, \hat{u}^{(t-1)}, \hat{u}^{(t)}$  ▷ See Sec. 5.5.3
  - 9:   Adjoint of the stiffness matrix  $\hat{f}^{(t)}, \hat{K}^{(t)} \rightarrow \hat{u}^{(t-1)}, \hat{\lambda}, \hat{\mu}$  ▷ See Sec. 5.5.4
  - 10:   Adjoint of the collision forces  $\hat{f}^{(t)} \rightarrow \hat{u}^{(t-1)}, \hat{u}^{(t-1)}, \hat{p}_{\text{plane}}$  ▷ See Sec. 5.5.5
  - 11:   Adjoint of the body forces  $\hat{f} \rightarrow \hat{f}_{\text{gravity}}$  ▷ See Sec. 5.5.6
  - 12: **end for**
- 

The parameters for which we compute the gradients are the following:

- the Lamé coefficients  $\hat{\lambda}, \hat{\mu}$ , which are then converted to the gradients of the Young's Modulus  $k$  and Poisson Ratio  $\rho$ ,
- the mass of the object  $\hat{m}$ ,
- the global gravity force  $\hat{f}_{\text{gravity}}$ ,
- the ground plane position and angle  $\hat{p}_{\text{plane}}$ ,
- the Rayleigh damping parameters  $\hat{\alpha}_1, \hat{\alpha}_2$ .

### 5.5.1. Adjoint: Levelset Advection

For the cost function that compares the values of the SDF on the whole grid (Sec. 5.4.1), an advection step to compute the implicit representation of the current state is needed.

The adjoint code for the advection can be mechanically implemented using the rules from the Instruction View Method (see Sec. 5.2.3.2). As a proof of concept, we implemented and successfully applied the adjoint code for the Direct Forward Advection (compare to Alg. A.2 for the forward code) in 2D. The adjoint code can be found in Alg. A.4 in the Appendix. We also tried to compute the adjoint of the Semi-Lagrange advection with the Shepard-Inversion, but found that the adjoint code of the Shepard-Inversion is numerically very unstable.

In 3D, only the cost function on the active nodes (see Sec. 5.4.1) or point clouds (see Sec. 5.4.3) are used, hence we didn't implement the adjoint of the advection in 3D.

### 5.5.2. Adjoint: Displacement Diffusion

The displacement diffusion (compare to Sec. 4.4.5 for the forward version) requires a linear system to be solved for every dimension. From 5.1.e) we know that in the adjoint simulation, we solve for the adjoint variables based on the transposed matrix. Since the diffusion matrix is symmetric, the same matrix as in the forward step can be used.

The mapping from  $\hat{u}_{\text{grid}}^{(t)}$  to the vector for the linear solve, and the mapping from the solution to  $\hat{u}^{(t)}$  is a direct translation of the forward code and omitted here.

### 5.5.3. Adjoint: Newmark Time Integration

The newmark time integration consists of three parts: the assembly of the matrix  $A$  and right hand side  $b$ , solving the linear system for  $u^{(t)}$ , computing the new velocity  $\dot{u}^{(t)}$ .

First, the adjoint of the velocity computation Eq. (2.6.3), applying the results from Tab.5.1, is given by

$$\hat{u}^{(t)} += \frac{1}{\theta\Delta t} \hat{u}^{(t)} \quad (5.5.1a)$$

$$\hat{u}^{(t-1)} += \left(1 - \frac{1}{\theta}\right) \hat{u}^{(t)} \quad (5.5.1b)$$

$$\hat{u}^{(t-1)} += \frac{-1}{\theta\Delta t} \hat{u}^{(t)}. \quad (5.5.1c)$$

Next, the adjoint of the linear system of equations  $A^{(t)}u^{(t)} = b^{(t)}$  was already presented in Tab. 5.1.e). Since  $A^{(t)}$  is symmetric, we can use the same conjugate gradient solver as in the forward step:

$$\text{Solve } A^{(t)}x_{\text{tmp}} = \hat{u}^{(t)} \text{ for } x_{\text{tmp}} \quad (5.5.2a)$$

$$\hat{b}^{(t)} += x_{\text{tmp}} \quad (5.5.2b)$$

$$\hat{A}^{(t)} += x_{\text{tmp}} \left(u^{(t)}\right)^T. \quad (5.5.2c)$$

Note that in the last step, the sparsity pattern of  $A^{(t)}$  is preserved in  $\hat{A}^{(t)}$ . Therefore, the outer product  $x_{\text{tmp}} \left(u^{(t)}\right)^T$  is only evaluated at the non-zero entries of  $\hat{A}^{(t)}$ .

Finally, the adjoint of the matrix computation Eq. (2.6.2), again applying the results from Tab.5.1, is given by

$$\hat{f}^{(t)} += \Delta t \hat{b}^{(t)} \quad (5.5.3a)$$

$$\hat{u}^{(t-1)} += \frac{1}{\theta} M \hat{b}^{(t)} \quad (5.5.3b)$$

$$\hat{u}^{(t-1)} += \left( \left( \frac{1}{\theta \Delta t} + \alpha_1 \right) M + (\alpha_2 - (1 - \theta) \Delta t) K^{(t)} \right) \hat{b}^{(t)} \quad (5.5.3c)$$

$$\hat{m} += \frac{1}{\theta} \hat{b}^{(t)} \left( \hat{u}^{(t-1)} \right)^T + \left( \frac{1}{\theta \Delta t} + \alpha_1 \right) \hat{b}^{(t)} \left( u^{(t-1)} \right)^T + \left( \frac{1}{\theta \Delta t} + \alpha_1 \right) \text{tr} \left( \hat{A}^{(t)} \right) \quad (5.5.3d)$$

$$\hat{K}^{(t)} += (\alpha_2 - (1 - \theta) \Delta t) \hat{b}^{(t)} \left( u^{(t-1)} \right)^T + (\alpha_2 + \theta \Delta t) \hat{A}^{(t)} \quad (5.5.3e)$$

$$\hat{\alpha}_1 += \text{vec}(M) \bullet \left( \hat{A}^{(t)} + \hat{b}^{(t)} \left( u^{(t-1)} \right)^T \right) \quad (5.5.3f)$$

$$\hat{\alpha}_2 += K^{(t)} \bullet \left( \hat{A}^{(t)} + \hat{b}^{(t)} \left( u^{(t-1)} \right)^T \right). \quad (5.5.3g)$$

The transposition of the matrices are omitted here since  $M, K, D$  are all symmetric. For the mass matrix we assumed a uniform mass, hence  $M = m M_0$  with  $M_0$  being the mass distribution, see Eq. (2.5.14).

#### 5.5.4. Adjoint: Stiffness Matrix

For the stiffness matrix  $K^{(t)}$ , we have to compute the adjoint of Alg. 4.1, while collecting the adjoint variables of  $\hat{u}^{(t-1)}$  and the Lamé coefficients  $\hat{\lambda}$  and  $\hat{\mu}$ .

For the corotational correction, the polar decomposition  $F = RS$  has to be computed. An analytic derivative of the polar decomposition was proposed by Feppon [28, Proposition 2.23, p. 74]. It requires the eigenvalue decomposition of  $S$ , which can be computed with the algorithm given by Smith [74]. It is, however, computational cheaper to compute the adjoint of the iterative procedure in Eq. (4.4.9) in our case. Applying the results from 5.1.i) and 5.1.j) gives rise to the following adjoint algorithm:

$$\begin{aligned} \hat{F} &= \hat{R} \\ \hat{F} &= \frac{1}{2} \left( \hat{F} - \left( R^{(t)} \right)^{-T} \hat{F}^T \left( R^{(t)} \right)^{-T} \right) \quad \text{for } t = T, \dots, 1. \end{aligned} \quad (5.5.4)$$

The Lamé coefficients  $\lambda$  and  $\mu$  are used linearly in the blocked element stiffness matrices  $K_{i,j}$  (Eq. (4.4.5)) and in the Nietsche Dirichlet blocked matrices  $KD_{j,c}$  (Eq. (4.4.16)). The per-element derivatives with respect to  $\lambda$  and  $\mu$  are easy to derive and denoted by  $\partial \lambda K_{i,j}$ ,  $\partial \mu K_{i,j}$ ,  $\partial \lambda KD_{j,c}$  and  $\partial \mu KD_{j,c}$ .

The final adjoint version of Alg. 4.1 is sketched in Alg. 5.4.

**Algorithm 5.4** Sketched kernel code for the computation of the block  $K_{i,j}$ 


---

```

1 //Launched with a work size of 64
2 __global__ void GridComputeStiffnessMatrixKernelAdjoint(...) {
3   int elementIdx = blockIdx.x; //element index e
4   int i = threadIdx.x % 8; // threadIdx.x goes from 0 to 63
5   int j = threadIdx.x / 8;
6   float wv[8] = ...; // Fetch the volume integration weights  $w_v(e,i)$  from memory
7   float mapping[8] = ...; // Fetch the mapping indices  $s(e,i)$  from memory
8
9   real adjLambdaPart, adjMuPart = 0;
10
11  //ADJOINT: Find the position of  $K_{i,j}$  in the large sparse matrix  $K$ 
12  real adjKePart;
13  for (int k = JA[mapping[i]]; k < mapping[mapping[i]+1]; ++k) {
14    if (IA[k] == mapping[j]) { // inner index matches
15      adjKePart = adjKData[k];
16    }
17  }
18
19  //ADJOINT: Dirichlet boundaries
20  if (isDirichlet(elementIdx)) {
21    adjLambdaPart += wb[i]*vec(adjKePart).dot(vec( $\partial\lambda\text{KD}_{j,i}$ ))
22                  + wb[j]*vec(adjKePart).dot(vec( $\partial\lambda\text{KD}_{i,j}^T$ ));
23    adjMuPart += wb[i]*vec(adjKePart).dot(vec( $\partial\mu\text{KD}_{j,i}$ ))
24              + wb[j]*vec(adjKePart).dot(vec( $\partial\mu\text{KD}_{i,j}^T$ ));
25  }
26
27  //ADJOINT: Corotation
28  real3 adjFi = adjForce[mapping[i]];
29  //KePart before the correction, FPart, F and rot are recomputed
30  real3x3 adjRot = 0;
31  //Adjoint of  $K^e \leftarrow R^e K^e (R^e)^T \underline{u}^e$  (Eq. (4.4.7)), see Tab. 5.1.f)
32  adjRot += adjKePartT*rot*KePartT+adjKePart*rot*KePart;
33  adjKePart = rotT*adjKePart*rot;
34  //Adjoint of  $f^e \leftarrow f^e - R^e K^e ((R^e)^T \underline{x}^e - \underline{x}^e)$  (Eq. (4.4.6)), see Tab. 5.1.g)
35  adjRot -= pos*adjFiT*rot*KePart + adjFi*posT*rot*KePartT - adjFi*posT*KePart;
36  adjKePart -= rotT*adjFi*(rotT*pos-pos)T;
37  //reduce adjRot into thread 1 over all 64 threads with __shfl_down and shared memory
38  adjRot = ...;
39  //Adjoint of the polar decomposition
40  __shared__ real3x3 adjFPart = 0;
41  if (j==0 && i==0) {
42    real3x3 adjF = polarDecompositionAdjoint(adjRot); //See Eq. (5.5.4)
43    adjFPart = 0.25 * adjF;
44  }
45  if (j==0) {
46    //adjoint of FPart, see Eq. (4.4.8)
47    real3 adjDispPart = adjFPart*(make_real3(i&1?1:-1, i&2?1:-1, i&4?1:-1)/h);
48    atomicAdd(adjLastDisplacements + mapping[i], adjDispPart);
49  }
50
51  //ADJOINT: computation of  $K_{i,j}$ 
52  for (int c=0; c<8; ++c) {
53    adjLambdaPart += wv[c]*vec(adjKePart).dot(vec( $\partial\lambda K_{i,j}$ ));
54    adjMuPart += wv[c]*vec(adjKePart).dot(vec( $\partial\mu K_{i,j}$ ));
55  }
56  atomicAdd(adjLambda, adjLambdaPart); //Reduction into global memory with atomics
57  atomicAdd(adjMu, adjMuPart);

```

---

The adjoint variables  $\hat{\lambda}$  and  $\hat{\mu}$  are converted into the adjoint variables of the Young's Modulus  $\hat{k}$  and the Poisson's Ratio  $\hat{\rho}$  via

$$\hat{k} += \frac{1}{2(1+\rho)}\hat{\mu} + \frac{\rho}{(1-2\rho)(1+\rho)}\hat{\lambda} \quad (5.5.5a)$$

$$\hat{\rho} += \frac{-k}{2(1+\rho)^2}\hat{\mu} + \left( \frac{-k\rho}{(1-2\rho)(1+\rho)^2} + \frac{k}{(1-2\rho)(1+\rho)} + \frac{2k\rho}{(1-2\rho)^2(1+\rho)} \right) \hat{\lambda}. \quad (5.5.5b)$$

### 5.5.5. Adjoint: Collision Forces

Collision Forces are generated by virtual springs as described in Sec. 2.8. The input to the forward simulation are the current displacement  $u^{(t)}$ , velocity  $\dot{u}^{(t)}$  and the ground plane in Hessian normal form (direction  $n_{\text{plane}}$ , distance  $d_{\text{plane}}$ , combined in  $p_{\text{plane}}$ ). The outputs are the collision forces  $f_{\text{collision}}^{(t)}$  that are added to the body forces.

Before the adjoint code of the collision resolution is presented, several helper functions have to be defined. Recall that the softmin-function is defined as  $r = \text{softmin}_\alpha(x, 0) := -\ln(1 + e^{-x\alpha})/\alpha$  (see Eq. (2.8.3)) and the time derivative as  $\dot{r} = \frac{\partial}{\partial t} \text{softmin}_\alpha(\dot{x}, 0) = \frac{1}{1+e^{x\alpha}}$  (see Eq. (2.8.4)). The adjoint versions are given by

$$\hat{x} = \text{adjSoftmin}_\alpha(\hat{r}, x) = \frac{\hat{r}}{1 + e^{x\alpha}} \quad (5.5.6a)$$

$$\hat{\dot{x}} = \text{adjSoftminDt}_\alpha(\hat{r}, \dot{x}) = \frac{-\hat{r}\alpha e^{\dot{x}\alpha}}{(1 + e^{\dot{x}\alpha})^2}. \quad (5.5.6b)$$

Further, we have to define the distance of a point to the ground. Given the ground plane in Hessian normal form with unit normal vector  $n_{\text{plane}}$  and distance from the origin  $d_{\text{plane}}$ , the distance of a point to the plane is given by

$$d = \text{dist}(x) := n_{\text{plane}} \bullet x + d_{\text{plane}} \quad (5.5.7)$$

and the respective time derivative  $\frac{\partial}{\partial t} \text{dist}(\dot{x})$  is given by

$$\dot{d} = \text{distDt}(\dot{x}) = n_{\text{plane}} \bullet \dot{x}. \quad (5.5.8)$$

Further, the direction of the repulsive forces are simply  $n := n_{\text{plane}}$ . The adjoint of the ground distance is then given by

$$\text{adjDist}(\hat{d}, \hat{n}; x) : \hat{n}_{\text{plane}} += \hat{n} + \hat{d}x; \hat{d}_{\text{plane}} -= \hat{d}; \hat{x} += \hat{d}d_{\text{plane}} \quad (5.5.9)$$

and the respective adjoint of the time derivative is given by

$$\text{adjDistDt}(\hat{d}; \dot{x}) : \hat{n}_{\text{plane}} += \hat{d}\dot{x}; \hat{\dot{x}} += \hat{d}d_{\text{plane}}. \quad (5.5.10)$$

The final adjoint algorithm is sketched in Alg. 5.5, it is threaded over each active cell in parallel. The softmin constant  $\alpha$  and the ground stiffness  $k$  are treated as meta parameters.

Because  $d_{\text{plane}}$  is assumed to be normalized, the orientation of the ground plane has actually only two degrees of freedom. Therefore, we convert the direction vector into spherical coordinates  $\phi, \theta$  and optimize over them.

Algorithm 5.5 Sketched kernel code for the computation of the block  $K_{i,j}$ 


---

```

1  __global__ void GridApplyCollisionForcesKernelAdjoint(...) {
2  int idx = ...; //current index of the processes cell
3  //fetch the reference position, displacement, velocity, SDF, adjoint forces at corners
4  real3 refPosX[8], dispX[8], velX[8], sdfX[8], adjForceX[8] = ...;
5  real3 posX[8]; for (int i=0; i<8; ++i) posX[i]=refPosX[i]+dispX[i];
6  //initialize adjoint variables
7  real3 adjDispX[8], adjVelX[8] = {};
8  real3 adjPlaneNormal = 0; real adjPlaneDistance = 0;
9  //the constant table EDGES contains the corner indices of each of the 12 edges
10 static const int2 EDGES[12] = ...;
11 for (int e=0; e<12; ++e) { //now loop over all edges of the cell
12     //find intersection of the surface with the current edge (if it exists)
13     real i = sdfX[EDGES[e].x] / (sdfX[EDGES[e].x]-sdfX[edges[e].y]);
14     if (i<0 || i>1) continue;
15     //get interpolated collision point on the edge
16     real3 pos = posX[EDGES[e].x]*(1-i) + posX[EDGES[e].y]*i;
17     real3 vel = velX[EDGES[e].x]*(1-i) + velX[EDGES[e].y]*i;
18     //forward simulation: collision force
19     real d = dist(pos); //see Eq. (5.5.7)
20     real dDt = distDt(vel); //see Eq. (5.5.8)
21     real smin = softmin(d); //see Eq. (2.8.3)
22     real sminDt = softminDt(d); //see Eq. (2.8.4)
23     real fCurrent = -k*smin;
24     real fDt = -k*sminDt*dDt;
25     real fNext = fCurrent + timestep*fDt; //see Eq. (2.8.2)
26     real f = theta*fNext + (1-theta)*fCurrent; //final force strength, see Eq. (2.6.4)
27     real3 fVec = normal * f;
28     //in the forward code, this force is blended into the output forces at the corners
29     //forceX[EDGES[e].x]+=(1-i)*fVec; forceX[EDGES[e].y]+=i*fVec;
30     //ADJOINT: blend forces
31     real3 adjFVec = (1-i)*adjForceX[EDGES[e].x]+i*adjForceX[EDGES[e].y];
32     //ADJOINT: collision force (line by line)
33     real3 adjNormal, adjVel, adjPos = 0; real adjF, adjFNext, adjFDt, ... = 0;
34     adjNormal += f*adjFVec; adjF += dot(adjFVec, normal);
35     adjFNext += theta*adjF; adjFCurrent += (1-theta)*adjF;
36     adjFCurrent += fNext; adjFDt += timestep*adjFNext;
37     adjSminDt += -k*dDt*adjFDt; adjDDt+=-k*sminDt*adjFDt;
38     adjSmin += -k*adjFCurrent;
39     adjD += adjSoftminDt(adjSminDt);
40     adjD += adjSoftmin(adjSmin);
41     adjVel, adjPlaneDistance += adjDistDt(adjDDt);
42     adjPos, adjPlaneDistance, adjPlaneNormal += adjDist(adjD);
43     //ADJOINT: interpolated collision point
44     adjVelX[EDGES[e].x]+=(1-i)*adjVel; adjVelX[EDGES[e].y]+=i*adjVel;
45     adjDispX[EDGES[e].x]+=(1-i)*adjPos; adjDispX[EDGES[e].y]+=i*adjPos;
46 }
47 //add-reduce the adjoint displacements and velocities into the global vector,
48 //as well as the adjoint ground plane adjPlaneDistance, adjPlaneNormal.
49 //...
50 }

```

---

### 5.5.6. Adjoint: Body Forces

In the forward problem, two values are precomputed: the body forces  $f_{\text{body}}$  based on the global gravity  $f_{\text{gravity}} \in \mathbb{R}^3$ , and the lumped mass matrix  $M = mM_0$ . The mass was already handled in the adjoint of the Newmark matrices, see Sec. 5.5.3.

The adjoint of  $f_{\text{gravity}}$  is computed as

$$\hat{f}_{\text{gravity}} = \hat{f}^{(t)} \bullet f_0, \quad (5.5.11)$$

where  $\hat{f}^{(t)}$  is the adjoint of the current force as computed in Eq. (5.5.3a).  $f_0$  is the force density and computed exactly as  $f_{\text{body}}$  by replacing  $f_{\text{gravity}}$  with  $(1, 1, 1)$ .

## 5.6. Optimization Algorithm

Now, all steps in the adjoint simulation are described. Taken together, we have a function  $g : \mathbb{R}^P \rightarrow \mathbb{R}$  that performs the elasticity simulation given the current parameters  $p \in \mathbb{R}^P$  and computes the value of the cost. Furthermore, we have a way to evaluate the gradient given by the adjoint variables  $\hat{p} = \nabla g(p)$ . This function can now be plugged into optimization routines.

As a first option, we use a simple gradient descent: Starting from an initial guess  $p_0$ , the next step is obtained by  $p_{t+1} = p_t - \gamma_t \nabla g(p_t)$ . The step size  $\gamma_t$  is thereby computed with the Barzilai-Borwein method [29].

As a second option, we use the L-BFGS method [16] as implemented in *Eigen* by Qiu [64]. The L-BFGS method as a Quasi-Newton method results in an adaptive step size for each of the optimized parameters. This is a big advantage over the simple gradient descent because the different parameters differ strongly in scale and importance for the cost function. This leads to a faster convergence of the L-BFGS over the gradient descent.

One might argue that a Gauss-Newton method could be used as an optimizer since all terms in the cost functions (see Sec. 5.4) are sums of squared functions  $\sum_i^N f_i(p)^2$ . Gauss-Newton, however, is not applicable in our case: First, in the case of sparse point clouds as cost function,  $N = T \cdot N_t$  is quite large with  $T$  being the number of timesteps and  $N_t$  the average number of observed points per timestep. Second, Gauss-Newton requires the gradient of each function in the sum  $\nabla f_i(p)$ . This is not possible in the adjoint framework: The adjoint simulation immediately reduces the gradient of the squared differences of observed points to the adjoint of the displacements. These are then traced back through time resulting in the final adjoint values of the parameters. The recovery of the gradients per  $f_i$  is not possible.

## 5.7. Memory Consumption

Until now, the memory required to evaluate a single optimization step is quite high, especially with many timesteps in the simulation. Let  $N_v$  be the number of active nodes and let



$N_g$  be the number of nodes in the whole grid. For an estimate of the memory requirements, we collect the variables that are stored in every timestep:

- Displacement and velocity  $u^{(t)}, \dot{u}^{(t)}$ , require  $2 * N_v * \text{sizeof}(\text{real3}) = 32N_v$  bytes.
- Adjoint of the displacement and velocity  $\hat{u}^{(t)}, \hat{\dot{u}}^{(t)}$ , requires  $32N_v$  bytes.
- Displacement on the whole grid  $u_{\text{grid}}^{(t)}$ , require  $2 * N_g * \text{sizeof}(\text{real3}) = 16N_g$  bytes.
- Adjoint of the displacement on the whole grid  $u_{\text{grid}}^{(t)}$ , requires  $16N_g$  bytes.
- Stiffness matrix  $K^{(t)}$  and force  $f^{(t)}$ . From Sec. 4.6 we know that the matrix has around 25 entries per row, hence  $25N_v$  entries in total. Therefore,  $K^{(t)}$  and  $f^{(t)}$  require  $N_v * (\text{sizeof}(\text{real3}) + 25 * \text{sizeof}(\text{real3} \times 3)) = 1216N_v$  bytes.
- Newmark matrix  $A^{(t)}$  and right-hand-side  $b^{(t)}$  require also  $1216N_v$  bytes.

In the dragon benchmark (see Sec. 5.9), we have  $N_v = 13,575$  and  $N_g = 84,672$ . If we simulate for 300 timesteps, this would require approximately 11 GB. This is way to much to fit into memory.

From the list above, we see that the most expensive entries are the matrices. Furthermore, from the timings in Sec. 4.6 we know that the computation of the stiffness matrix and collision forces only take a very small amount of time compared to solving the linear system. Therefore, we only store the solutions of the elasticity and diffusion solve ( $u^{(t)}, \dot{u}^{(t)}, u_{\text{grid}}^{(t)}$ ) and recompute the stiffness matrix, collision force and Newmark matrix in the adjoint step. Moreover, we don't need to evaluate the cost function for every timestep before the adjoint step. The adjoint step requires only the adjoint variables of the current timestep and the previous timestep. Therefore, we interleave the evaluation of the cost function and its gradients with the computation of the adjoint. Thus, we don't need to store the adjoint versions of the displacements.

With these optimizations, the memory requirement drops to around 540 MB or less than 5% of the memory requirement before the optimization. An increase in the time to evaluate a single adjoint step could not be measured. This is because the evaluation time is dominated by the linear system solvers so that the additional evaluation of the stiffness matrix falls within the uncertainty of the time measurements.

## 5.8. Analysis of the Gradients

To test the adjoint simulation and get insights into how the individual parameters behave and contribute to the cost function, we analyze the gradient of the cost function with respect to individual parameters.

For all parameters except the ground position for the collision, the test case in Fig. 5.4 is used. For the ground position, the test case in Fig. 5.5 is used instead. As a cost function,

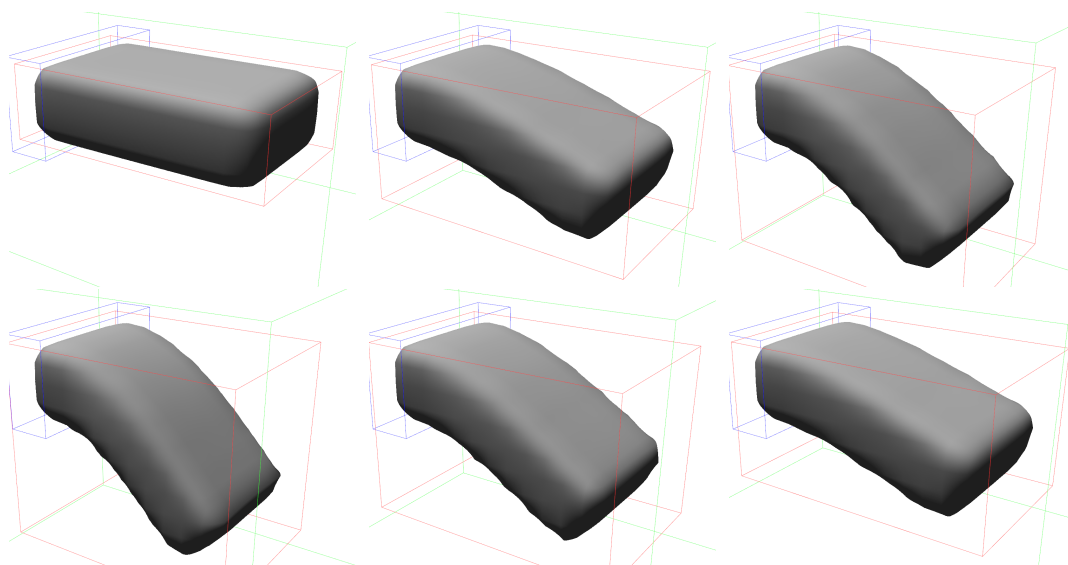


Figure 5.4.: First test case for all examples except collision, 40 timesteps

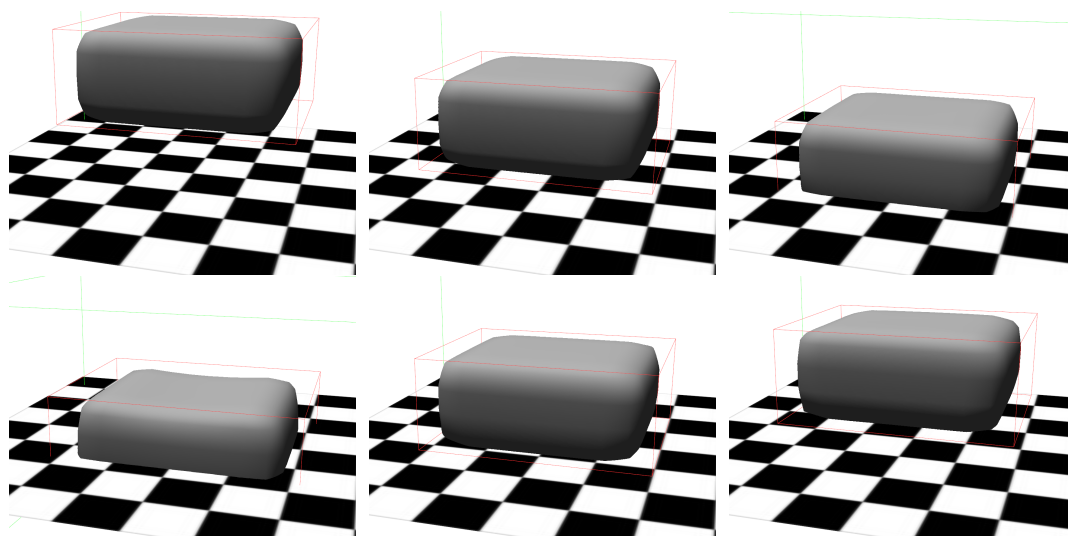


Figure 5.5.: Second test case for the collision gradients, 30 timesteps

the squared differences on the displacements (Sec. 5.4.1) are used. The new sparse point cloud cost function is evaluated separately, see below. Fig. 5.6 plots the cost function value and the gradients when one parameter is changed and the others are fixed. Only for the ground plane orientation, the two angles  $\theta$  and  $\phi$  are displayed together because they are intertwined. The arrows indicate the direction of the optimization, i.e. the negative gradient, scaled in length by a constant factor for each plot for better visual perception. It is not surprising that each of these plots is convex and hence easy to optimize for.

It becomes more interesting if multiple parameters are optimized jointly. For example, in Fig. 5.7a, the cost function value and the gradients are plotted for both the Young's Modulus and Poisson's Ratio jointly. One can see that there exists a flat valley with several local minima. This is a challenge for the optimization since, depending on the initialization, the optimization will get stuck in such a local minimum. The same problem occurs, for example, when jointly optimizing the two Rayleigh damping factors, the damping on mass and on stiffness, in Fig. 5.7b. Again, a long valley can be seen in which the optimization gets stuck.

Furthermore, the two plots described above (Fig. 5.7a and Fig. 5.7b) also show the steps taken during the optimization. As initial values,  $k = 2500$ ,  $\rho = 0.35$ ,  $\alpha_1 = 0.15$ ,  $\alpha_2 = 0.015$  with an initial cost of 464 were chosen. The ground truth values are  $k = 2000$ ,  $\rho = 0.3$ ,  $\alpha_1 = 0.1$ ,  $\alpha_2 = 0.01$  (the setting that are used in the first test case 5.4). After 8 gradient descent steps, the final values are  $k = 1997$ ,  $\rho = 0.287$ ,  $\alpha_1 = 0.113$ ,  $\alpha_2 = 0.0115$  with a cost of 0.2132. The algorithm converged to a local minimum of quite low cost and relatively close to the ground truth values. Fig. 5.7c plots the value of the cost function over the course of the optimization. The first step is very small because of a conservative initial step size. After that first step, the Barzilai-Borwein method provides adaptive step sizes that lead to a fast convergence.

As a last test, we'll evaluate the performance of the sparse point cost function (see Sec. 5.4.3) under increasing noise. Fig. 5.8a shows the setting that is used: nine simulated cameras with each a resolution of 100x100 pixels. First, Fig. 5.8b shows the gradients with respect to the Young's Modulus and Poisson's Ratio with no additional noise, i.e. a perfect observation. Besides a different scaling of the cost function values, the result is indistinguishable from Fig. 5.7a where the cost function on the active displacements was used. When we add a little bit of Gaussian noise with a variance of 0.1 world units (=0.5 voxels) in Fig. 5.8c the value of the cost functions gets more uniform (depicted by a more uniform color), but the gradients don't change. Even with as much as a Gaussian noise with variance of 5 world units (= 25 voxels), the gradient can still be computed, while the absolute value of the cost function is almost uniform (Fig. 5.8e). Only the gradient points to a bit stiffer material. This shows that the sparse point cost function is very robust against noise with zero mean. The measurement errors seem to completely cancel each other when being reduced to the gradients of the parameters.

## 5. Inverse Simulation

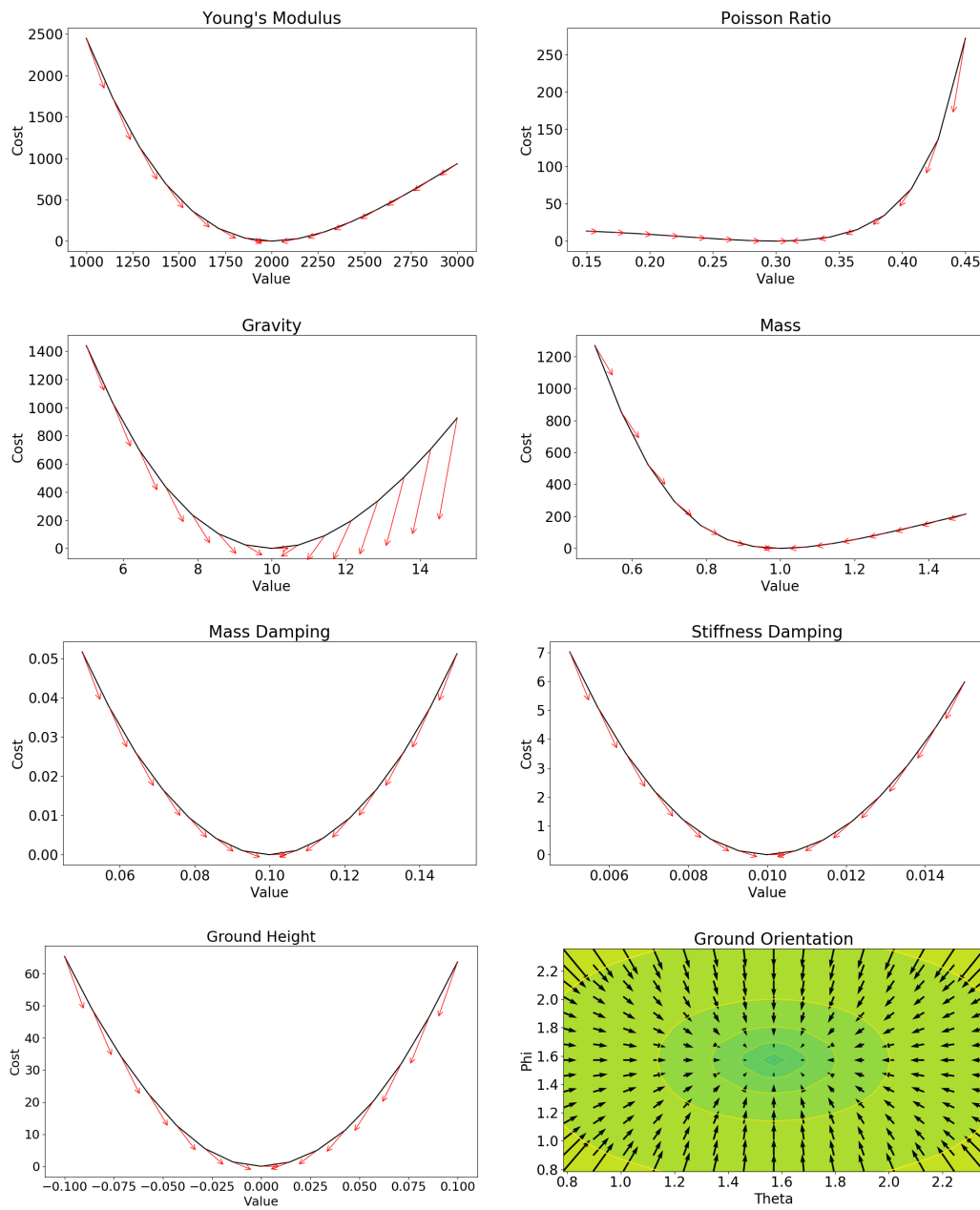


Figure 5.6.: Gradients for the input parameters

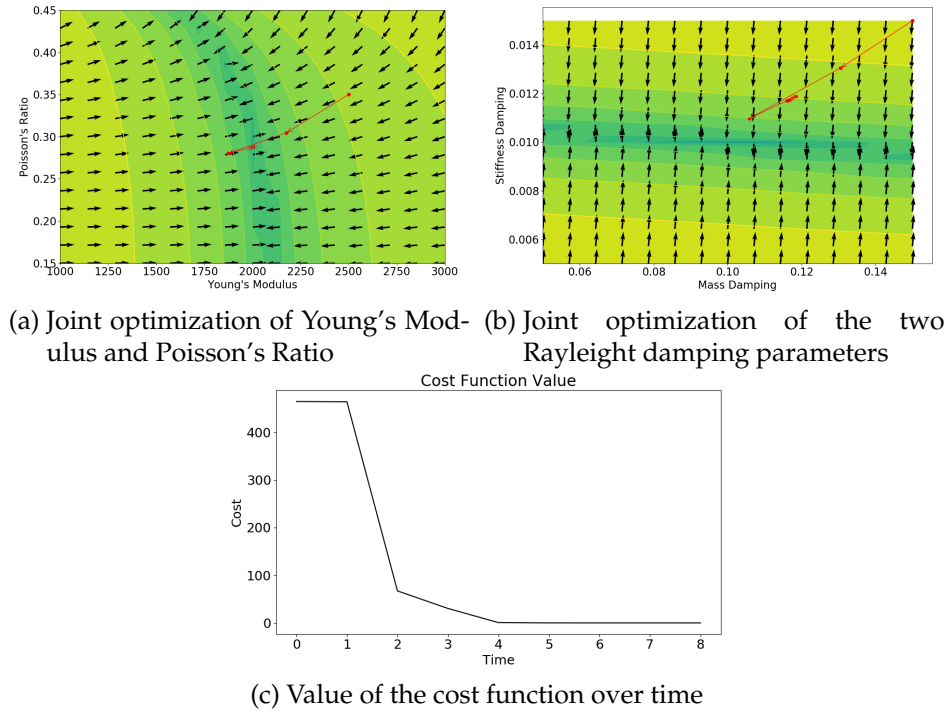


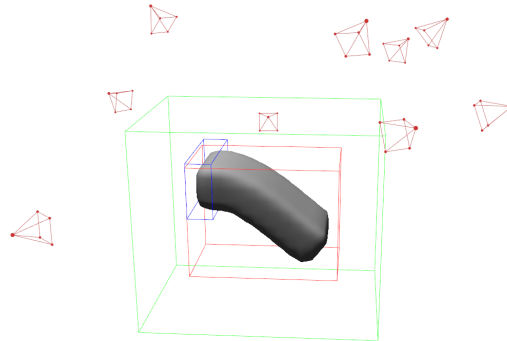
Figure 5.7.: Joint optimization. The black arrows: negated gradient, normalized in length. The red arrows: steps taken by the optimization

## 5.9. Benchmarks and Examples

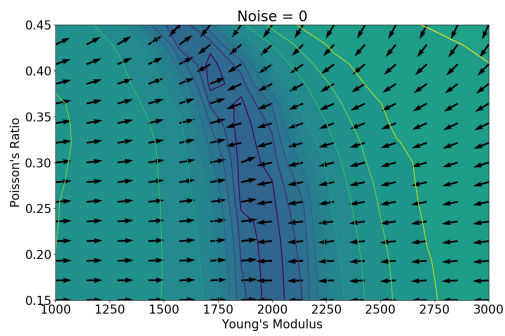
For an analysis of the performance of the presented adjoint framework, we evaluate it at three larger examples. The number of nodes and cells, as well as the timings for the three examples can be found in Tab. 5.2. The timings are all per timestep and per optimization step. Hence, for the time to perform one single optimization step, the timings of the forward step, cost function evaluation and backward step have to be summed and multiplied with the number of timesteps.

The first test case is a torus that is attached in the top (blue box) and is otherwise allowed to swing freely, see Fig. 5.9. In this example, the Young's Modulus and Poisson Ratio are the unknown parameters. As initial values  $k = 2200$ ,  $\rho = 0.32$  were chosen (initial cost is 1.088). The ground truth values are  $k = 2000$ ,  $\rho = 0.30$ . As a cost function, the sparse point cloud cost function with ten cameras was used, the optimization algorithm was L-BFGS. Fig. 5.10 shows how the optimization behaves over time. The cost function decreases constantly except for one step. After 19 steps, the optimization settles in a local minimum with  $k = 1882$  and  $\rho = 0.2568$ , with a final cost of 0.2495.

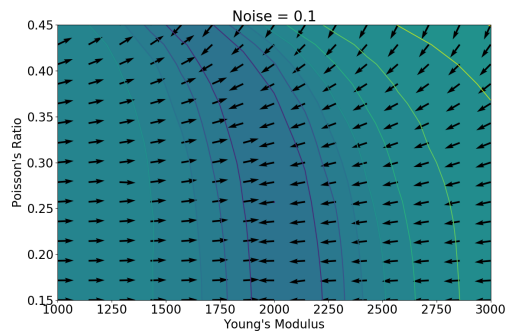
A second test case can be seen in Fig. 5.11. Here, the Stanford Bunny is simulated how it collides with the ground plane, 65 timesteps long. The ground truth values are



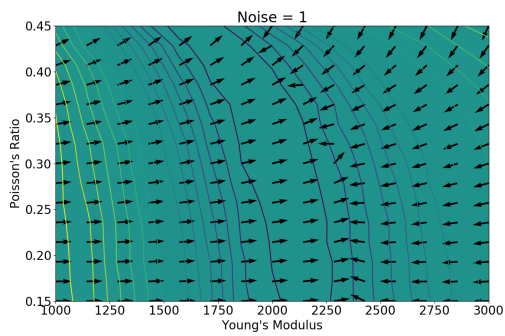
(a) Placement of the simulated cameras



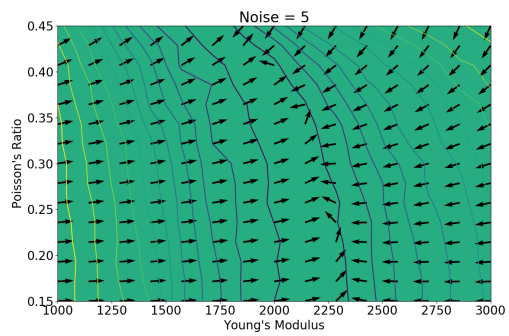
(b) No additional noise



(c) Additional noise of 0.1 world units



(d) Additional noise of one world unit = five voxels



(e) Additional noise of five world units

Figure 5.8.: Behavior of the sparse point cost function under different noise settings

Test Case	Swinging Torus	Bunny	Dragon
# active nodes	9220	3495	13575
# empty nodes	34300	14919	71097
# active cells	7180	2560	9968
# timesteps	30	65	30
forward time per timestep avg, min, max in seconds	0.211, 0.168, 0.343	0.039, 0.027, 0.095	0.450, 0.188, 0.712
cost evaluation per timestep avg, min, max in seconds	0.021, 0.186, 0.026	0.0088, 0.0053, 0.065	0.027, 0.012, 0.044
backward time per timestep avg, min, max in seconds	0.392, 0.211, 0.519	0.077, 0.0039, 0.110	0.900, 0.368, 1.216

Table 5.2.: Timings of the benchmarks

$n_{\text{plane}} = (-0.011, 0.996, -0.091)$  ( $\theta = 1.662, \phi = 1.582$ ) and  $d_{\text{plane}} = 0.0$ . As initial values,  $n_{\text{plane}} = (0.151, 0.957, 0.245)$  ( $\theta = 1.541, \phi = 1.414$ ) and  $d_{\text{plane}} = 0.03$  are chosen. This looks like a small change, but it is quite a difficult problem: The bunny bounces off the ground in a completely different direction, as seen in Fig. 5.11a and Fig. 5.11b. This is also difficult for the sparse point cost function because in late timesteps, the initial simulation is so far away from the observations, that no point-cell correspondence can be found. Therefore, only the first timesteps after the collision give gradients in the beginning. After a couple of iterations when the solution approaches the ground truth, also later timesteps contribute to the cost function. As it can be seen in the plots of the cost function value and the parameters, the optimizer reaches a local minimum after only a few iterations and does not move far afterward. The reconstructed values are  $n_{\text{plane}} = (-0.075, 0.994, -0.075)$  ( $\theta = 1.646, \phi = 1.646$ ) and  $d_{\text{plane}} = -0.042$ . The simulation when run with these parameters is depicted in Fig. 5.11c, which is perceptually quite close to the ground truth. During the optimization, a divergence in the CG solver occurred, leading to a greatly increased cost. This issue is analyzed in detail in Sec. 5.10. Therefore, the optimization was stopped after 14 iterations, the 15<sup>th</sup> timestep with the divergence was discarded. If we would continue, the current parameters would spike to some extreme values, but then slowly recover to a good solution.

As a third example, we test how good the optimization scales with the degrees of freedom in the FE discretization. The dragon in Fig. 5.12 consists of almost 10.000 cells. The head of the dragon is fixed as a Dirichlet boundary while the rest of the body can swing freely. The simulation is performed for 300 timesteps, but only the first 30 are used in the optimization. With 30 timesteps, a single step in the optimization takes around 40 seconds. Because we optimize only for the Young's Modulus in this case, we use the Gradient Descent method as optimizer. The cost function is the sparse point cost with 12 cameras and a Gaussian Noise with variance of 1 voxel. The ground truth value for the Young's Modulus is  $k = 5000$ , the initial value for the optimizer is  $k = 10000$  (cost of 5.55) and the recon-

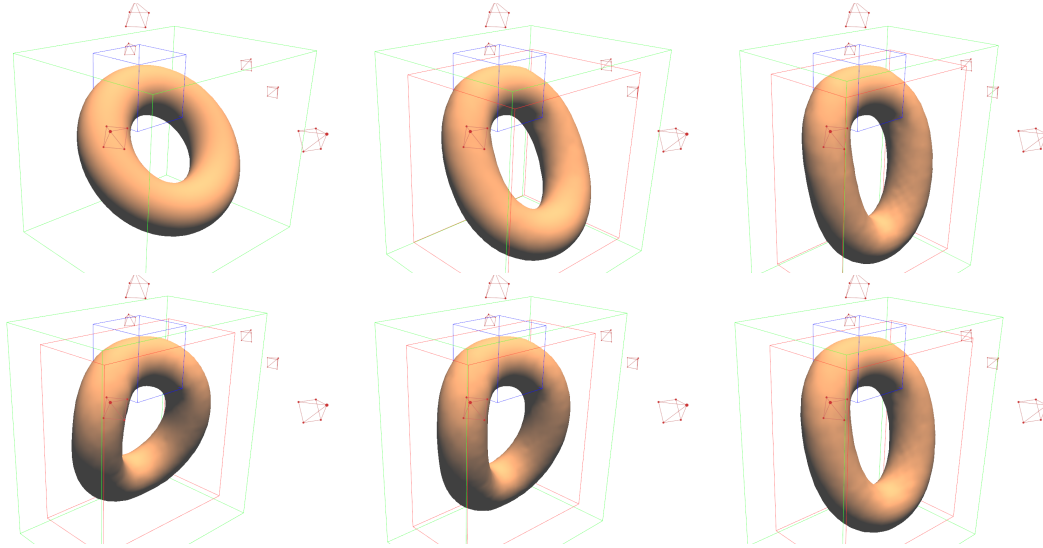


Figure 5.9.: Selected timesteps from the torus benchmark

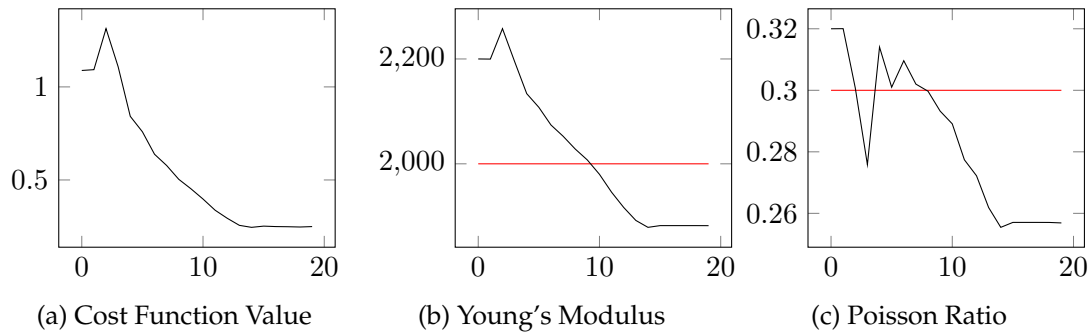


Figure 5.10.: Swinging Torus: Optimization of the unknown parameters. Ground truth values are in red.



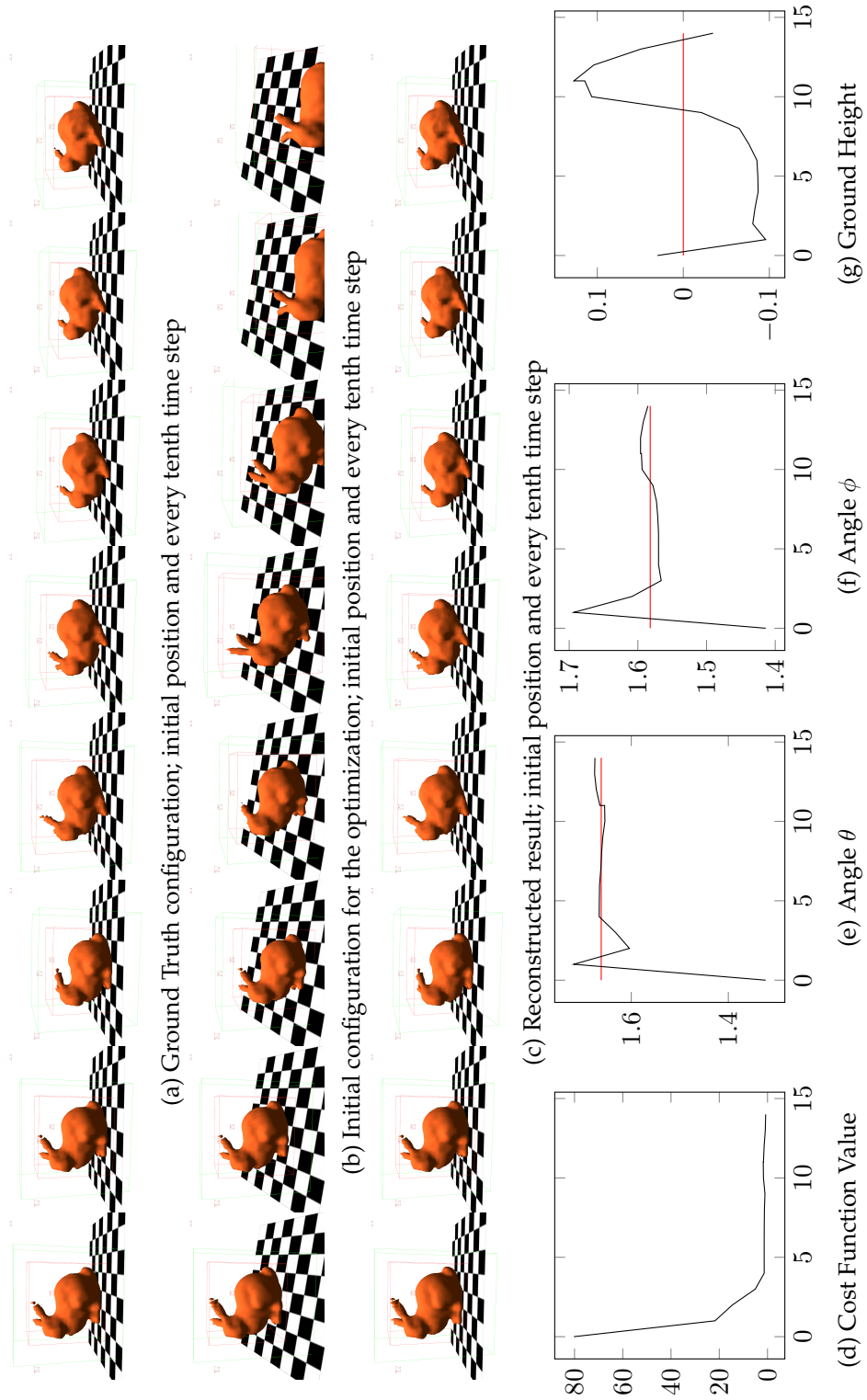


Figure 5.11.: Results of the bunny test case

structed value after 18 steps (12 minutes) is  $k = 4348$  (cost of 3.79). We also tried to run the optimization using the full 300 timesteps but the gradient got too noisy for the Gradient Descent or L-BFGS to converge. This might be due to numerical errors in the Gradient Descent solver that accumulate over time and start to influence the gradient negatively.

In all examples, however, the optimizer could never reconstruct the actual ground truth values. Except for trivial examples with a single parameter, the optimizer will most likely get stuck in local minima. Possible solutions are discussed in the future work section 6.

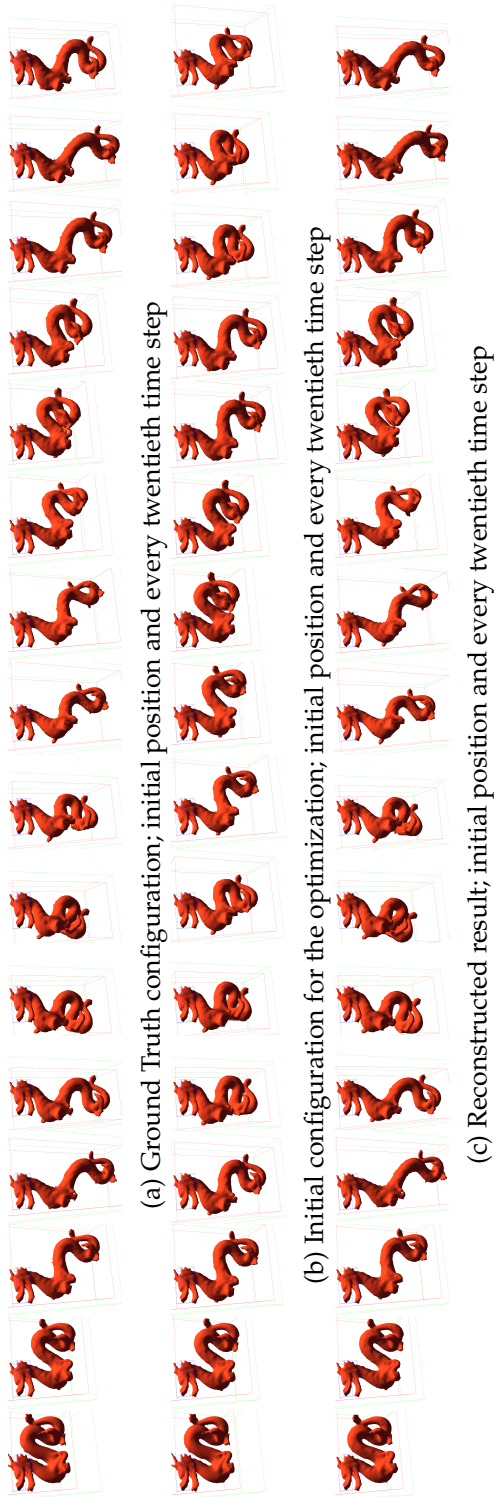
### 5.10. Stability

In rare cases, the conjugate gradient solver in the forward and adjoint step diverges. This problem occurs more often the larger the system becomes. In the dragon benchmark above, this was the case for around every thousandth timestep in the forward simulation, in the bunny benchmark only once for all simulation runs, and never occurred in the torus test case. Our experiments have shown that this seems to be a purely numerical issue. For testing purposes, in the case of divergence in the CG, we copied the Newmark matrices into another math toolkit and could solve the linear system there without any difficulties. Furthermore, this instability is non-deterministic. Running the simulation again with the same settings does not reproduce the same errors. The cause for this numerical issue might be manifold. For example, the CG solver uses atomics at some places for reductions. Since they don't define a fixed ordering of operations, non-deterministic numerical cancellation might be a problem.

If a divergence in the CG occurs in the forward pass during the optimization, the value of the cost function grows strongly. This happened in the bunny test case above, but the optimization would recover after a steps. Because recovery of the optimization can't be guaranteed, we stop the optimization if such a divergent case is detected.

The same numerical instabilities also occur in the adjoint pass. The use of the sparse point cost function increases the number of divergent cases. Recall that in the dragon benchmark only the first 30 timesteps were used in the optimization. With 60 or more timesteps, the adjoint pass already faces these divergent cases so that a proper computation of the gradients becomes impossible. In the simpler active displacement cost function, the numerical issues only start after 200 iterations.

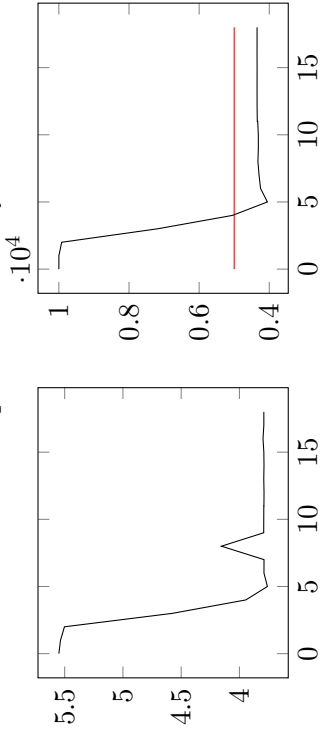
In future work we analyze the described numerical issue in greater detail. A numerically more accurate solver might already prevent the problem. Since the numerical instabilities only arise in large test cases, they don't hamper real-time applications with fewer degrees of freedom than the dragon benchmark.



(a) Ground Truth configuration; initial position and every twentieth time step

(b) Initial configuration for the optimization; initial position and every twentieth time step

(c) Reconstructed result; initial position and every twentieth time step



(d) Cost Function Value

(e) Young's Modulus  $k$

Figure 5.12.: Results of the bunny test case



## 6. Conclusion and Future Work

In this thesis we presented a new discretization method for the elasticity simulation on a regular grid with partially filled cells. In contrast to previous methods, this method does not require a tetrahedral mesh, but instead works on an implicit representation of the object via a signed distance function stored on a regular hexahedral grid. The advantage is that it allows the use of 3D scans as inputs, which are often available directly as a SDF (e.g. with Kinect Fusion [55]). As a disadvantage, the implementation becomes more complicated because the integrals have to be evaluated over parts of the cell. Therefore, the evaluation of a single partially filled hexahedral element takes about 5-6 times longer than the evaluation of a single completely filled tetrahedral element. This is compensated by the fact that much more tetrahedral elements are required as hexahedral elements to achieve the same resolution of the discretization.

Furthermore, we described how to efficiently compute the element matrices and load vector including the corotation correction on the GPU. Our method makes heavy use of warp exchange operations to facilitate the best parallelization of the tasks and an optimal occupancy. The stiffness matrix assembly only requires 1-5% of the time to solve the linear system. Together with a fast conjugate gradient solver on the GPU, this allows the simulation to be executed with up to 10,000 elements at interactive frame rates.

Next, we embedded the regular grid discretization in an adjoint framework that allows us to compute the gradients of some cost functions with respect to the simulation parameters. These parameters include the Young's Modulus, Poisson ratio, gravity as the main body force, Rayleigh damping parameters and the ground plane position for collisions. The gradients are plugged into a black-box gradient descent or LBFGS-algorithm, which enables us to recover these parameters from observations. This optimization, however, is prone to get stuck in local minima. We further showed how to reduce the memory required for the adjoint simulation by recomputing the stiffness matrix and collision forces in the adjoint step. Because the execution time is dominated by the conjugate gradient solver, this method allows to reduce the memory by 95% without a measurable increase of the computation time.

Lastly, we presented a new cost function that allows us to use sparse point cloud observations directly in the optimization. These point clouds can e.g. be obtained from 3D scans. This method allows us to avoid the explicit reconstruction of a signed distance field or even an optical flow tracing to get the actual displacements on the nodes. We further showed that this cost function is very robust against noise.

The proposed adjoint simulation, however, still faces several shortcomings that are addressed in future work. First, the optimization is prone to get stuck in local minima. A

proper choice of priors on the simulation parameters might help to avoid these local minima and guides the optimization towards the global optimum. The choice of priors was not covered in this thesis.

Furthermore, numerical instabilities can occur in the CG solver that lead to divergences and wrong results in about one in every hundred to one in every thousand timesteps, depending on the problem size. This seems to be a purely numerical issue in the linear solver and not a limitation of the method. The linear system can still be solved by an accurate offline solver on the CPU. In future work we address this problem with a more accurate solver.

Connected to the above problem, the whole simulation could be accelerated by replacing the conjugate gradient solver with a multigrid solver. Especially in a multigrid hierarchy, the partially filled grid cells might be of further advantage since the downscaled SDF can again be represented by coarser partially filled cells. In addition to a multigrid hierarchy to solve the elasticity problem per timestep, a multigrid approach over space and time might also be used. In this approach, the simulation is first solved on a coarse grid in space and a larger step in time and then refined in a multigrid fashion until the full resolution in space and time is reached. This might also help to reduce numerical instabilities because a smaller system in the degrees of freedom and the number of timesteps is used in early stages when the current guess is still far away from the final solution.

Moreover, we would like to embed this method in a full 3d reconstruction procedure. This includes the reconstruction of the reference configuration from 3D scans and apply the sparse point cloud cost function on real-world observations. Another possible application is in data compression for 3D reconstruction. Instead of storing a full 3D scan for every timestep, only the first timestep has to be stored as reference configuration and all other timesteps are predicted by the elasticity simulation.

Furthermore, it might be possible to improve the initial reconstruction with observations from later timestep. This would be helpful if the initial scan is also affected by some uncertainty. Moreover, the adjoint simulation could be used in a texture superresolution framework, in which color observations in later timesteps are mapped back to the reference configuration with the adjoint of the elasticity simulation and then adds color details to the reference configuration.

# Appendix

## A. Additional Algorithms

---

**Algorithm A.1** Forward code for PLACEINTOMATRIX, see Sec. 2.5.1

---

```
1: function PLACEINTOMATRIX( $K, f, K_e, f_e, e$ )
2:   for  $i \in \{1, 2, 3\}$  do
3:      $i' = s(e, i)$  ▷ Global index of local node  $i$  of element  $e$ 
4:     for  $j \in \{1, 2, 3\}$  do
5:        $j' = s(e, j)$ 
6:       if isDirichlet( $i$ ) then
7:         pass ▷  $v_i$  is zero by definition of  $V_0$ 
8:       else if isDirichlet( $j$ ) then
9:          $F(2i' : 2i' + 1) += K_e(2i : 2i + 1, 2j : 2j + 1)u_D$  ▷ Add Dirichlet boundary to
the force
10:      else
11:         $K(2i' : 2i' + 1, 2j' : 2j' + 1) += K_e(2i : 2i + 1, 2j : 2j + 1)$  ▷ regular free node
12:      end if
13:    end for
14:    if !isDirichlet( $i$ ) then
15:       $F(2i' : 2i' + 1) += F_e(2i : 2i + 1)$ 
16:    end if
17:  end for
18:  return  $K, f$ 
19: end function
```

---

---

**Algorithm A.2** Implementation of the Direct Forward Advection method (see Sec. 2.5.2.5)

---

```

1: function DIRECTFORWARDADVECTION( $\phi^{\text{old}}, u, \tau$ )
2:    $\phi^{\text{new}} = \mathbf{0}$ 
3:    $w = \mathbf{0}$  ▷ Accumulated blurring weights
4:   for  $x = 1, \dots, \text{width}; y = 1, \dots, \text{height}$  do
5:      $v = \phi^{\text{old}}(x, y)$  ▷ The current value of the levelset
6:      $p = \begin{pmatrix} x \\ y \end{pmatrix} + \Delta t u(x, y)$  ▷ The target position
7:     for  $x' = \max\{1, \lfloor p_x - \tau \rfloor\}, \dots, \min\{\text{width}, \lceil p_x + \tau \rceil\}$  do
8:       for  $y' = \max\{1, \lfloor p_y - \tau \rfloor\}, \dots, \min\{\text{height}, \lceil p_y + \tau \rceil\}$  do
9:         Loop over all cells around the target position that are affected by the blurring kernel
10:         $d = \left\| \begin{pmatrix} x' \\ y' \end{pmatrix} - p \right\|^2$  ▷ Distance to the target
11:        if  $d \leq \tau^2$  then ▷ Cell is affected by the blurring kernel
12:           $w' = \exp\left(\frac{d}{2(\tau/3)^2}\right)$ 
13:          Gaussian kernel, variance is  $\frac{\tau}{3}$  so that  $\approx 98\%$  of the mass are inside the kernel radius.
14:          Note that no normalization is needed, this is done afterwards.
15:           $\phi^{\text{new}}(x', y') += w'v$ 
16:           $w(x', y') += w'$ 
17:        end if
18:      end for
19:    end for
20:     $\phi^{\text{new}} = \phi^{\text{new}} / w$  ▷ Normalize
21:    Optionally: fill cells with  $w = 0$ 
22:  end function

```

---



---

**Algorithm A.3** Forward code for the rotation correction / Corotational formulation on a 2D mesh (see Sec. 2.7)

---

```

1: function CORRECTROTATION( $K_{e,\text{stiffness}}, \mathbf{u}_e$ )
    ▷  $K_{e,\text{stiffness}}$  is the stiffness matrix,  $\mathbf{u}_e$  the displacement of this triangle from the previous time step
    ▷  $\mathbf{x}$  is the reference position of the vertices
2:  $\mathbf{d}_i = \mathbf{x}_i + \mathbf{u}_{e,i}, i \in \{1, 2, 3\}$ 
3:  $D_{\text{def}} = \begin{pmatrix} \mathbf{d}_{1,x} - \mathbf{d}_{3,x} & \mathbf{d}_{2,x} - \mathbf{d}_{3,x} \\ \mathbf{d}_{1,y} - \mathbf{d}_{3,y} & \mathbf{d}_{2,y} - \mathbf{d}_{3,y} \end{pmatrix}$ 
4:  $D_{\text{ref}} = \begin{pmatrix} \mathbf{x}_{1,x} - \mathbf{x}_{3,x} & \mathbf{x}_{2,x} - \mathbf{x}_{3,x} \\ \mathbf{x}_{1,y} - \mathbf{x}_{3,y} & \mathbf{x}_{2,y} - \mathbf{x}_{3,y} \end{pmatrix}$ 
5:  $J = D_{\text{def}} D_{\text{ref}}^{-1}$ 
    ▷ The Jacobi of the deformation
6: if  $\det(J) > \epsilon$  then
    ▷ To avoid numerical problems in the case of very small or no rotations
7:  $R' = J + \text{sign}(\det(J)) \begin{pmatrix} J_{2,2} & -J_{2,1} \\ -J_{1,2} & J_{1,1} \end{pmatrix}$ 
    ▷ Polar decomposition
8:  $s = 1/\sqrt{R_{1,1}^2 + R_{2,1}^2}$ 
9:  $R = sR'$ 
    ▷ Needed to ensure that  $R$  is normalized
10: else
11:  $R = \mathbf{1}^2$ 
12: end if
13:  $R_e = \begin{pmatrix} R & & \\ & R & \\ & & R \end{pmatrix}$ 
14:  $x_e = (\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3)$ 
15:  $K_e = R_e K_{e,\text{stiffness}} R_e^T$ 
16:  $F_e = -R_e K_{e,\text{stiffness}} (R_e^T x_e - x_e)$ 
17: return  $K_e, F_e$ 
18: end function

```

Note that  $K_{e,\text{stiffness}}$  and  $D_{\text{ref}}$  stays constant for all time steps, are precomputed and treated as external parameter.

---

**Algorithm A.4** Adjoint of the Direct Forward Advection method (compare to Alg. A.2, see Sec. 5.5.1)

---

```

1: function ADJOINTDIRECTFORWARDADVECTION( $\hat{\phi}^{\text{new}}, \phi^{\text{old}}, \phi^{\text{new}}, u, \tau, w$ )
   The input, output and normalization weights from the forward version are needed
2:    $\hat{w} = -(\hat{\phi}^{\text{new}} \odot \phi^{\text{old}}) / (w \odot w)$  ▷ Adjoint of the normalization step
3:    $\hat{\phi}^{\text{new}} = \hat{\phi}^{\text{new}} / w$ 
4:   for  $x = \text{width}, \dots, 1; y = \text{height}, \dots, 1$  do
5:      $v = \phi^{\text{old}}(x, y)$  ▷ The current value of the levelset
6:      $p = \begin{pmatrix} x \\ y \end{pmatrix} + \Delta t u(x, y)$  ▷ The target position
7:      $\hat{v} = 0, \hat{p} = \mathbf{0}$  ▷ Assemble the adjoint values here
8:     for  $x' = \min\{\text{width}, \lceil p_x + \tau \rceil\}, \dots, \max\{1, \lfloor p_x - \tau \rfloor\}$  do
9:       for  $y' = \min\{\text{height}, \lceil p_y + \tau \rceil\}, \dots, \max\{1, \lfloor p_y - \tau \rfloor\}$  do
         Loop over all cells around the target position that are affected by the blurring kernel
         in reversed order
10:         $d = \left\| \begin{pmatrix} x' \\ y' \end{pmatrix} - p \right\|^2$  ▷ Distance to the target
11:        if  $d \leq \tau^2$  then ▷ Cell is affected by the blurring kernel
12:           $w' = \exp\left(\frac{d}{2(\tau/3)^2}\right)$ 
13:           $\hat{w}' = \hat{w}(x', y')$  ▷ Adjoint of  $w(x', y') += w'$ 
14:           $\hat{v} += w \hat{\phi}^{\text{new}}(x', y') ; \hat{w}' += v \hat{\phi}^{\text{new}}(x', y')$  ▷ Adjoint of  $\phi^{\text{new}}(x', y') += w' v$ 
15:           $\hat{d} = -\exp\left(\frac{d}{2(\tau/3)^2}\right) \frac{\hat{w}'}{2(\tau/3)^2}$  ▷ Adjoint of  $w' = \exp\left(\frac{d}{2(\tau/3)^2}\right)$ 
16:           $\hat{p} += 2(p - \begin{pmatrix} x' \\ y' \end{pmatrix}) \hat{d}$  ▷ Adjoint of  $d = \left\| \begin{pmatrix} x' \\ y' \end{pmatrix} - p \right\|^2$ 
17:        end if
18:      end for
19:    end for
20:     $\hat{u}(x, y) = \Delta t \hat{p}$  ▷ Adjoint of  $p = \begin{pmatrix} x \\ y \end{pmatrix} + \Delta t u(x, y)$ 
21:     $\hat{\phi}^{\text{new}}(x, y) = \hat{v}$  ▷ Adjoint of  $v = \phi^{\text{old}}(x, y)$ 
22:  end for
23: end function

```

---

## B. Tables

Size	cuBlas	cuMat	Eigen	Numpy
100	0.0001488	0.0037088	0.0003004	0.00495774
1000	0.0001456	0.00369664	0.0004506	0.00573897
10000	0.0001504	0.00368832	0.0055286	0.0202516
100000	0.00015808	0.0037232	0.0484356	0.160451
1000000	0.00020032	0.00370272	1.614	6.66087
10000000	0.00022208	0.00377824	18.236	73.0739
50000000	0.00022336	0.00571136	93.9108	370.887

Table B.1.: *cuMat* benchmark: Linear combination with 2 combinations

Combinations	cuBlas	cuMat	Eigen	Numpy
1	0.00014336	0.00361568	1.38066	2.96753
2	0.00021216	0.00368544	1.58203	6.78211
3	0.00194784	0.00397888	1.97174	10.4266
4	0.166469	0.00422528	2.0635	14.199
5	0.167558	0.00430688	2.40177	19.0094
6	0.325902	0.00459712	2.65564	21.729
7	0.330418	0.0045984	2.95554	25.8566
8	0.488936	0.00764864	3.31724	29.1512
9	0.48627	0.00508224	3.63084	33.2603
10	0.643359	0.00545568	4.14723	36.7435

Table B.2.: *cuMat* benchmark: Linear combination with a constant vector size of  $n = 10^6$

## B. Tables

Size	cuBlas	cuMat	Eigen
10	0.010252	0.00955791	0.0013821
100	0.0121149	0.012815	0.0651417
1000	0.491989	0.405348	2.85587
10000	49.2694	37.1613	302.804

Table B.3.: *cuMat* benchmark: Sparse matrix - vector multiplication, 2D Poisson matrix

Size	cuMat	Eigen
10	0.350014	0.0264716
100	3.79062	17.7015
1000	1160.36	15700.8

Table B.4.: *cuMat* benchmark: Conjugate Gradient benchmark on a 2D diffusion problem

Resolution	TetNumFreeNodes	GridNumFreeNodes	TetNumFixedNodes	GridNumEmptyNodes	TetNumElements	GridNumElements	TetAvgEntriesPerRow	GridAvgEntriesPerRow	TetMatrixAssembly	GridMatrixAssembly	TetCollisionForces	GridCollisionForces	TetCGIterations	GridCGIterations	TetCGTime	GridCGTime	GridBoundingBox	GridDiffusion	GridAdvection
5	605	675	25	2175	2496	448	11.733884	20.703704	0.000176	0.000123	0.000060	0.000057	67	42	0.015278	0.009479	0.000128	0.016226	0.000417
6	1014	1309	42	2563	4500	960	12.191321	22.048128	0.000280	0.000180	0.000056	0.000057	83	50	0.016990	0.012588	0.000130	0.015342	0.000441
7	1554	1729	42	3671	7128	1296	12.619048	22.362637	0.000405	0.000225	0.000044	0.000058	99	58	0.019589	0.015888	0.000121	0.016232	0.000437
8	2044	3105	112	3564	9828	2464	12.712329	23.196457	0.000549	0.000384	0.000049	0.000056	102	64	0.022255	0.019141	0.000126	0.014513	0.000466
9	2736	3825	144	4995	13524	3072	12.925439	23.379085	0.000729	0.000454	0.000047	0.000064	113	74	0.027875	0.023716	0.000132	0.016496	0.000617
10	3951	4131	180	6173	19968	3328	13.164515	23.426531	0.001054	0.000449	0.000051	0.000054	136	84	0.034792	0.029373	0.000122	0.017357	0.000581
11	5290	6479	220	6121	27216	5400	13.337996	23.947368	0.001226	0.000663	0.000047	0.000060	144	90	0.043321	0.039636	0.000124	0.015511	0.000605
12	6776	7623	264	7769	35340	6400	13.466942	24.062311	0.001585	0.000783	0.000054	0.000050	140	101	0.046389	0.048389	0.000127	0.016038	0.000620
13	8008	10465	462	7895	42840	8976	13.506993	24.399140	0.001898	0.001039	0.000046	0.000051	154	111	0.056206	0.074547	0.000127	0.016720	0.000848
14	9744	12025	468	10421	52272	10368	13.625205	24.483077	0.002338	0.001207	0.000047	0.000050	166	123	0.075248	0.092076	0.000123	0.021438	0.000829
15	12415	13325	585	11785	67392	11520	13.724124	24.526904	0.003004	0.001333	0.000050	0.000060	170	126	0.088268	0.105245	0.000122	0.026270	0.000944
16	15162	17415	714	11769	83148	15288	13.801741	24.772840	0.003730	0.001747	0.000049	0.000054	174	132	0.108872	0.146607	0.000121	0.034200	0.001178
17	18180	19575	720	14425	99792	17248	13.886029	24.833461	0.004515	0.001944	0.000051	0.000053	169	150	0.123963	0.175925	0.000133	0.027644	0.001411
18	20340	25823	810	13132	112056	23040	13.913471	25.037951	0.005085	0.002580	0.000056	0.000061	171	148	0.130449	0.219857	0.000134	0.038842	0.001591
19	23584	28611	1216	16973	132300	25600	13.958446	25.084862	0.005981	0.002880	0.000053	0.000054	170	162	0.144307	0.267637	0.000144	0.045499	0.001773
20	28373	29733	1360	20031	159744	26624	14.022310	25.097400	0.007200	0.002984	0.000053	0.000063	176	167	0.176455	0.288962	0.000132	0.051075	0.001936

Table B.5.: Benchmark of the Cuboid testcase with Dirichlet boundary conditions

Resolution	TetNumFreeNodes	GridNumFreeNodes	TetNumFixedNodes	GridNumEmptyNodes	TetNumElements	GridNumElements	TetAvgEntriesPerRow	GridAvgEntriesPerRow	TetMatrixAssembly	GridMatrixAssembly	TetCollisionForces	GridCollisionForces	TetCGIterations	GridCGIterations	TetCGTime	GridCGTime	GridBoundingBox	GridDiffusion	GridAdvection
5	630	675	0	2175	2496	448	12.114286	20.703704	0.000152	0.000110	0.000052	0.000093	51	40	0.010400	0.008629	0.000126	0.016145	0.000327
6	1056	1309	0	2563	4500	960	12.562500	22.048128	0.000245	0.000162	0.000059	0.000088	62	46	0.013219	0.011398	0.000125	0.015510	0.000331
7	1596	1729	0	3671	7128	1296	12.863409	22.362637	0.000371	0.000221	0.000058	0.000098	72	54	0.014821	0.014650	0.000124	0.016845	0.000389
8	2156	3105	0	3564	9828	2464	13.000928	23.196457	0.000564	0.000372	0.000058	0.000103	75	58	0.017998	0.017957	0.000124	0.014743	0.000414
9	2880	3825	0	4995	13524	3072	13.197917	23.379085	0.000761	0.000434	0.000062	0.000100	80	66	0.019676	0.020876	0.000123	0.017118	0.000539
10	4131	4131	0	6173	19968	3328	13.395062	23.426531	0.001065	0.000449	0.000061	0.000097	84	61	0.024543	0.021316	0.000122	0.022711	0.000338
11	5510	6479	0	6121	27216	5400	13.544828	23.947368	0.001277	0.000635	0.000056	0.000110	87	81	0.027262	0.035425	0.000125	0.016463	0.000548
12	7040	7623	0	7769	35340	6400	13.658239	24.062311	0.001635	0.000743	0.000062	0.000112	89	85	0.030039	0.041485	0.000122	0.016788	0.000590
13	8470	10465	0	7895	42840	8976	13.712397	24.399140	0.001986	0.001040	0.000060	0.000123	90	89	0.037797	0.060018	0.000143	0.017039	0.000733
14	10212	12025	0	10421	52272	10368	13.797297	24.483077	0.002416	0.001178	0.000062	0.000134	91	92	0.042818	0.070319	0.000137	0.023863	0.000816
15	13000	13325	0	11785	67392	11520	13.888462	24.526904	0.003135	0.001289	0.000071	0.000145	93	86	0.052413	0.072414	0.000139	0.027782	0.000875
16	15876	17415	0	11769	83148	15288	13.962585	24.772840	0.003872	0.001714	0.000065	0.000159	95	94	0.063794	0.105669	0.000145	0.034047	0.000997
17	18900	19575	0	14425	99792	17248	14.021693	24.833461	0.004671	0.001899	0.000071	0.000167	96	94	0.072698	0.111150	0.000136	0.028838	0.001150
18	21150	25823	0	13132	112056	23040	14.047092	25.037951	0.005255	0.002549	0.000070	0.000205	96	95	0.076682	0.142527	0.000141	0.027264	0.001351
19	24800	28611	0	16973	132300	25600	14.097500	25.084862	0.006179	0.002808	0.000079	0.000212	97	97	0.088623	0.160768	0.000151	0.033257	0.001601
20	29733	29733	0	20031	159744	26624	14.149834	25.097400	0.007484	0.002937	0.000076	0.000213	99	82	0.106839	0.143439	0.000145	0.051823	0.001756

Table B.6.: Benchmark of the Cuboid testcase with ground collision

## C. Additional Equations

### Analytic solutions to the 3D partial integrals

Let  $v_i$  be the value to integrate and  $\phi_i$  be the SDF value at node  $i$ .  $\Omega^e$  is the part of the cell that is within the object described by the SDF. It given by

$$\Omega^e := \left\{ x \in [0, 1]^3 : \text{trilinear-interpolation}(x, \phi_1, \dots, \phi_8) \leq 0 \right\}.$$

The surface integrals are evaluated by discretizing the current cell using the Marching Cubes algorithm and evaluating the integral over the triangles.

For the volume integrals, we are interested in the value of  $\int_{\Omega^e} f(x) dx$  with  $f$  being  $N_1, \dots, N_8$ . The object surface is defined by the zero isosurface of the trilinear interpolation, which is a polynomial of degree one in three variables. We couldn't find an analytic solution for the exact surface. Therefore, we approximate the integral by a sum of piecewise integrals with planar surfaces. The basic subdivisions are described in Tab. C.1. The integrals are then analytically solved by a symbolic programming toolbox (Mathematica), with  $f$  replaced by  $N_1$  to  $N_8$  respectively. All other cases are rotations or inversion of the described subdivisions. If the object surface intersect edge  $i$  and  $j$ , then  $c_{ij}$  denotes the position along the edge where the intersection occurs.

There are some few, rare cases that are not handled by the analytic integrals. These are then solved with numerical sampling. Not all cases are solved numerically for performance reasons. A few thousand samples are required per cell to accurately approximate the integrals if the cell is only filled to a small amount.

The approximation error introduced by these piecewise planar integrals was measured to be a relative error of 5.26% or an absolute error of 1.02% on average.

### C. Additional Equations

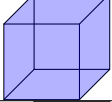
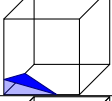
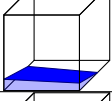
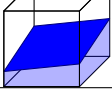
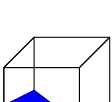
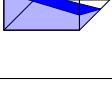

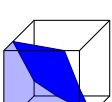
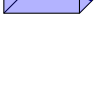
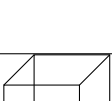
I		$\int_{\Omega^e} f(x) dx = \int_0^1 \int_0^1 \int_0^1 f(x, y, z) dz dy dx$
II		$\int_{\Omega^e} f(x) dx = \int_0^{c_{12}} \int_0^{c_{13}(1-\frac{x}{c_{12}})} \int_0^{c_{15}(1-\frac{x}{c_{12}})(1-\frac{y}{c_{13}})} f(x, y, z) dz dy dx$
III		$\int_{\Omega^e} f(x) dx = \int_0^1 \int_0^{(1-x)c_{13}+xc_{24}} \int_0^{((1-x)c_{15}+xc_{26})(1-\frac{y}{(1-x)c_{13}+xc_{24}})} f(x, y, z) dz dy dx$
IV		$\int_{\Omega^e} f(x) dx = \int_0^1 \int_0^1 \int_0^{(1-x)(1-y)c_{15}+x(1-y)c_{26}+(1-x)yc_{37}+xy c_{47}} f(x, y, z) dz dy dx$
V		$\int_{\Omega^e} f(x) dx = \int_0^{c_{34}} \int_0^1 \int_0^{(1-x)(1-y)c_{15}+x(1-y)c_{26}+(1-\frac{x}{c_{14}})yc_{37}+y} f(x, y, z) dz dy dx$ $+ \int_{c_{34}}^1 \int_0^{x^*} \int_0^{(1-\frac{x-c_{34}}{1-c_{34}})((1-y)((1-c_{34})c_{15}+c_{34}c_{26})) + \frac{x-c_{34}}{1-c_{34}}(1-\frac{y}{x^*})c_{26}} f(x, y, z) dz dy dx$ with $x^* = 1 - \frac{x - c_{34}}{1 - c_{34}} + \frac{x - c_{34}}{1 - c_{34}} c_{24}$
VI		$\int_{\Omega^e} f(x) dx = \int_0^{c_{56}} \int_0^{(1-\frac{x}{c_{56}})c_{57}} \int_0^1 f(x, y, z) dz dy dx$ $+ \int_0^{c_{56}} \int_{(1-\frac{x}{c_{56}})c_{57}}^{(1-\frac{x}{c_{56}})} \int_0^{(1-\frac{x}{c_{56}})((1-\frac{y-c_{57}}{1-c_{57}})+\frac{y-c_{57}}{1-c_{57}}c_{37}) + \frac{x}{c_{56}}} f(x, y, z) dz dy dx$ $+ \int_0^1 \int_{(1-y)c_{56}}^{(1-y)c_{56}+yc_{34}} \int_0^{(1-y)+y(1-\frac{x}{c_{34}})c_{37}} f(x, y, z) dz dy dx$ $+ \int_0^1 \int_{(1-y)c_{56}+yc_{34}}^{(1-y)+yc_{34}} \int_0^{(1-y)((1-\frac{x-c_{56}}{1-c_{56}})+\frac{x-c_{56}}{1-c_{56}}c_{26})} f(x, y, z) dz dy dx$ $+ \int_{c_{34}}^1 \int_{1-\frac{x-c_{34}}{1-c_{34}}}^{(1-\frac{x-c_{34}}{1-c_{34}})+\frac{x-c_{34}}{1-c_{34}}c_{24}} \int_0^{\frac{x-c_{34}}{1-c_{34}}(1-\frac{y}{x_{24}})c_{26}} f(x, y, z) dz dy dx$
VIIa		$\int_{\Omega^e} f(x) dx = \text{II} + \text{II}$
VIIb		$\int_{\Omega^e} f(x) dx = \text{II} + \text{II}$
VIII		$\int_{\Omega^e} f(x) dx = \text{II} + \text{III}$
IX		$\int_{\Omega^e} f(x) dx = \text{II} + \text{V}$

Table C.1.: Base cases for the partial integrals

---





# Bibliography

- [1] Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., ... & Zheng, X. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.
- [2] Abe, Y., da Silva, M., & Popovic, J. Multiobjective control with frictional contacts. *Eurographics/ ACM SIGGRAPH Symposium on Computer Animation*, 2007.
- [3] Alterovitz, R., Goldberg, K., Pouliot, J., Hsu, I. C. J., Kim, Y., Noworolski, S. M., & Kurhanewicz, J. Registration of mr prostate images with biomechanical modeling and nonlinear parameter estimation. *Medical physics*, 33(2):446–454, 2006.
- [4] Anitescu, M., & Potra, F. A. Formulating dynamic multi-rigid-body contact problems with friction as solvable linear complementarity problems. *Nonlinear Dynamics*, 14:231–247, 1997.
- [5] Baraff, D. An introduction to physically based modeling: rigid body simulation I - unconstrained rigid body dynamics. *SIGGRAPH Course Notes*, 1997.
- [6] Barbič, J., da Silva, M., & Popović, J. Deformable object animation using reduced optimal control. *ACM Transactions on Graphics*, 28(3):1, 2009.
- [7] Barbič, J., Sin, F., & Grinspun, E. Interactive editing of deformable simulations. *ACM Transactions on Graphics*, 31(4):1–8, 2012.
- [8] Benk, J., Ulbrich, M., & Mehl, M. The Nitsche method of the Navier-Stokes Equations for Immersed Boundaries. *Seventh International Conference on Computation Fluid Dynamics (ICCFD7)*, 2012.
- [9] Bergou, M., Mathur, S., Wardetzky, M., & Grinspun, E. Tracks: Toward directable thin shells. *ACM Transactions on Graphics*, 26(99):50, 2007.
- [10] Bochud, N., & Rus, G. Probabilistic inverse problem to characterize tissue-equivalent material mechanical properties. *IEEE transactions on ultrasonics, ferroelectrics, and frequency control*, 59(7):1443–1456, 2012.
- [11] Bourke, P. Polygonising a scalar field. <http://paulbourke.net/geometry/polygonise/>, 1994.
- [12] Bridson, R. *Fluid simulation for computer graphics*. AK Peters/CRC Press, 2015.

- [13] Bridson, R., Fedkiw, R., & Anderson, J. Robust treatment of collisions, contact and friction for cloth animation. *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, 2002.
- [14] Burman, E., Hansbo, P., Larson, M. G., & Zahedi, S. Cut finite element methods for coupled bulk–surface problems. *Numerische Mathematik*, 133(2):203–231, 2016.
- [15] Burman, E., Hansbo, P., Larson, M. G., Massing, A., & Zahedi, S. Full gradient stabilized cut finite element methods for surface partial differential equations. *Computer Methods in Applied Mechanics and Engineering*, 310:278–296, 2016.
- [16] Byrd, R., Lu, P., Nocedal, J., & Zhu, C. A limited memory algorithm for bound constrained optimization. *SIAM Journal on Scientific Computing*, 16(5):1190–1208, 1995.
- [17] Chen, X., Zheng, C., Xu, W., & Zhou, K. An asymptotic numerical method for inverse elastic shape design. *ACM Transactions on Graphics*, 33(4):1–11, 2014.
- [18] Cook, J. D. Soft maximum. <https://www.johndcook.com/blog/2010/01/13/soft-maximum/>, 2010. Accessed 07/26/2018.
- [19] Cook, S. *CUDA programming: a developer’s guide to parallel computing with GPUs*. Newnes, 2012.
- [20] Coros, S., Martin, S., Thomaszewski, B., Schumacher, C., Sumner, R., & Gross, M. Deformable objects alive! *ACM Transactions on Graphics*, 31(4):1–9, 2012.
- [21] Courtecuisse, H., Allard, J., Kerfriden, P., Bordas, S. P., Cotin, S., & Duriez, C. Real-time simulation of contact and cutting of heterogeneous soft-tissues. *MEDICAL IMAGE ANALYSIS*, 18(2):394–410, FEB 2014.
- [22] Crum, W. R., Camara, O., & Hawkes, D. J. Methods for inverting dense displacement fields: Evaluation in brain image registration. *International Conference on Medical Image Computing and Computer-Assisted Intervention*, pages 900–907, 2007.
- [23] Davatzikos, C. Spatial transformation and registration of brain images using elastically deformable models. *Computer Vision and Image Understanding*, 66(2):207–222, 1997.
- [24] Demidov, D. VexCL. <https://github.com/ddemidov/vexcl>, 2018.
- [25] Dick, C. *Computational Steering for Implant Planning in Orthopedics*. Dissertation, Technische Universität München, München, 10.01.2012.
- [26] El Foutayenia, Y., El Bouananib, H., & Khaladic, M. The linear complementarity problem and a method to find all its solutions. *Information in Sciences and Computing*, 3, 2014.

- 
- [27] Embar, A., Dolbow, J., Harari, I. Imposing dirichlet boundary conditions with nitsche's method and spline-based finite elements. *International Journal for Numerical Methods in Engineering*, 194(3), 2010.
- [28] Feppon, F. *Riemannian Geometry of Matrix Manifolds for Lagrangian Uncertainty Quantification of Stochastic Fluid Flows*. Master's thesis, Massachusetts Institute of Technology, 02 2017.
- [29] Fletcher, R. On the barzilai-borwein method. In Liqun Qi, Koklay Teo, and Xiaoqi Yang, editors, *Optimization and Control with Applications*, pages 235–256, Boston, MA, 2005. Springer US.
- [30] Fratarcangeli, M., Tibaldi, V., Pellacini, F. Vivace: a Practical Gauss-Seidel Method for Stable Soft Body Dynamics. *ACM TRANSACTIONS ON GRAPHICS*, 35(6), NOV 2016. ACM SIGGRAPH Asia Conference, Macao, PEOPLES R CHINA, 2016.
- [31] Galoppo, N., Otaduy, M. A., Tekin, S., Gross, M., & Lin, M. C. Soft articulated characters with fast contact handling. *Computer Graphics Forum*, 26(3):243–253, 2007.
- [32] Gavin, H. P. Numerical integration in structural dynamics, 2016.
- [33] Georgii, J., Krüger, J., & Westermann, R. Interactive collision detection for deformable and gpu objects. *IADIS International Journal on Computer Science and Information Systems*, 2(2):162–180, 2007.
- [34] Gokhale, N. H., Barbone, P. E., & Oberai, A. A. Solution of the nonlinear elasticity imaging inverse problem: the compressible case. *Inverse Problems*, 24(4):045010, 2008.
- [35] Gould, H., Tobochnik, J., & Christian, W. *An introduction to computer simulation methods*, volume 1. Addison-Wesley New York, 1988.
- [36] Greenough, C. Newmark's method of direct integration. <http://www.softeng.rl.ac.uk/st/projects/felib3/Docs/html/Intro/intro-node52.html>. Accessed: 05/17/2018.
- [37] Guennebaud, G., & Jacob, B. Eigen v3. <http://eigen.tuxfamily.org>, 2010.
- [38] Hansbo, P., Larson, M. G., & Zahedi, S. A cut finite element method for coupled bulk-surface problems on time-dependent domains. *Computer Methods in Applied Mechanics and Engineering*, 307:96–116, 2016.
- [39] Hansbo, P., Larson, M. G., Zahedi, S. A cut finite element method for a stokes interface problem.
- [40] Hauth, M., & Strasser, W. Corotational simulation of deformable solids. *Journal of WSCG*, 12(1-3), 2003.

- [41] Higham, N. J. Computing the polar decompositions - with applications. *SIAM Journal on Scientific and Statistical Computing*, 7(4), 1986.
- [42] Higham, N. J., & Schreiber, R. S. Fast polar decomposition of an arbitrary matrix. *SIAM Journal on Scientific and Statistical Computing*, 2(4):648–655, 1990.
- [43] Hodgins, J. A multiresolution framework for dynamic deformations. *ACM SIGGRAPH*, 2002.
- [44] Juntunen, M., & Stenberg, R. Nitsche’s method for general boundary conditions. *Mathematics of Computation*, 78(267):1353–1374, 2009.
- [45] Kim, M., Pons-Moll, G., Pujades, S., Bang, S., Kim, J., Black, M. J., & Lee, S.-H. Data-driven physics for human soft tissue animation. *ACM Transactions on Graphics*, 36(4):1–12, 2017.
- [46] Kirk, D. B., & Wen-Mei, W. H. *Programming massively parallel processors: a hands-on approach*. Morgan kaufmann, 2016.
- [47] Lamecki, A., Dziekonski, A., Balewski, L., Fotyga, G., & Mrozowski, M. GPU-Accelerated 3D Mesh Deformation for Optimization Based on the Finite Element Method. *RADIOENGINEERING*, 26(4):924–929, DEC 2017.
- [48] Liu, Y., Jiao, S., Wu, W., & De, S. GPU accelerated fast FEM deformation simulation. In *Circuits and Systems, 2008. APCCAS 2008. IEEE Asia Pacific Conference on*, pages 606–609. IEEE, 2008.
- [49] Lorensen, W. E., & Cline, H. E. Marching cubes: A high resolution 3d surface construction algorithm. *ACM siggraph computer graphics*, 21(4):163–169, 1987.
- [50] Mächler, M. Accurately computing  $\log(1 - \exp(\cdot))$  – assessed by the rmpfr package, 2012.
- [51] McNamara, A., Treuille, A., Popović, Z., & Stam, J. Fluid control using the adjoint method. *ACM Transactions on Graphics (TOG)*, 23(3):449–456, 2004.
- [52] Mittal, R., & Iaccarino, G. Immersed boundary methods. *Annual Review of Fluid Mechanics*, 37(1):239–261, 2005.
- [53] Monzpart, A., Thuerey, N., & Mitra, N. J. Smash. *ACM Transactions on Graphics*, 35(6):1–14, 2016.
- [54] Mordatch, I., Todorov, E., & Popović, Z. Discovery of complex behaviors through contact-invariant optimization. *ACM Transactions on Graphics*, 31(4):1–8, 2012.

- 
- [55] Newcombe, R. A., Izadi, S., Hilliges, O., Molyneaux, D., Kim, D., Davison, A. J., ... & Fitzgibbon, A. Kinectfusion: Real-time dense surface mapping and tracking. In *2011 10th IEEE International Symposium on Mixed and Augmented Reality*, pages 127–136. IEEE, 26.10.2011 - 29.10.2011.
- [56] Nvidia Corporation. cuBLAS. <https://docs.nvidia.com/cuda/cublas/index.html>, 2018.
- [57] Nvidia Corporation. cuSPARSE. <https://docs.nvidia.com/cuda/cusparse/index.html>, 2018.
- [58] Nvidia Corporation. Nvidia cuda toolkit documentation v9.2.148. <https://docs.nvidia.com/cuda/index.html>, 2018.
- [59] Oberhuber, T. Numerical recovery of the signed distance function. *Proceedings of Czech–Japanese Seminar in Applied Mathematics 2004*, pages 148–164, 2004.
- [60] Ovsyannikov, A., Sabel’nikov, V., & Gorokhovski, M. A new level set equation and its numerical assessments. *Center for Turbulence Research, Proceedings of the Summer Program 2012*, 2012.
- [61] Pan, Z., & Manocha, D. Efficient solver for spacetime control of smoke. *ACM Transactions on Graphics (TOG)*, 36(5):68a, 2017.
- [62] Pan, Z., & Manocha, D. Active animations of reduced deformable models with environment interactions. *ACM Transactions on Graphics (TOG)*, 37(3):36, 2018.
- [63] Petersen, K. B., & Pedersen, M. S. The matrix cookbook, nov 2012. Version 20121115.
- [64] Qiuv, Y. Lbfgspp. <https://github.com/yixuan/LBFGSpp>, 2016.
- [65] Risholm, P., Ross, J., Washko, G. R., & Wells, W. M. Probabilistic elastography estimating lung elasticity. *Inf Process Med Imaging*, pages 699–710, 2011.
- [66] Risholm, P., Samset, E., & Wells, W. Bayesian estimation of deformation and elastic parameters in non-rigid registration. In Bernd Fischer, Benoît M. Dawant, and Cristian Lorenz, editors, *Biomedical Image Registration*, pages 104–115, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.
- [67] Rupp, K. Viennacl. <http://viennacl.sourceforge.net/>, 2016.
- [68] Rupp, K., Tillet, P., Rudolf, F., Weinbub, J., Morhammer, A., Grasser, T., JÄ¼ngel, A., & Selberherr, S. Viennacl—linear algebra library for multi-and many-core architectures. *SIAM Journal on Scientific Computing*, 38(5):S412–S439, 2016.
- [69] Schulz, C., von Tycowicz, C., Seidel, H. P., & Hildebrandt, K. . Animating deformable objects using sparse spacetime constraints. *ACM Transactions on Graphics*, 33(4):1–10, 2014.

- [70] Shi, P., & Liu, H. Stochastic finite element framework for simultaneous estimation of cardiac kinematic functions and material parameters. *Medical Image Analysis*, 7(4):445–464, 2003.
- [71] Shoemake, K., & Duff, T. Matrix animation and polar decomposition. In *Proceedings of the conference on Graphics interface*, volume 92, pages 258–264, 1992.
- [72] Sifakis, E. D. Fem simulation of 3d deformable solids: A practitioner’s guide to theory, discretization and model reduction: Part one: The classical fem method and discretization methodology, 2012.
- [73] Skeel, K. L. Introduction to soft body physics. <http://cg.skeelogy.com/?download=SoftBodyPhysicsTutorial>. Accessed: 29/30/2018.
- [74] Smith, O. K. Eigenvalues of a symmetric 3x3 matrix. *Commun. ACM*, 4(4):168–, April 1961.
- [75] Stam, J. Notes on adjoint control of graphical simulations. *Course Notes SIGGRAPH Course Note*, 2004.
- [76] Stewart, D., & Trinkle, J. C. An implicit time-stepping scheme for rigid body dynamics with coulomb friction. In *Proceedings 2000 ICRA. Millennium Conference. IEEE International Conference on Robotics and Automation. Symposia Proceedings (Cat. No.00CH37065)*, pages 162–169. IEEE, 24-28 April 2000.
- [77] Tan, J., Turk, G., & Liu, C. K. Soft body locomotion. *ACM Transactions on Graphics*, 31(4):1–11, 2012.
- [78] Tejada, E., & Ertl, T. Large steps in gpu-based deformable bodies simulation. *Simulation Modelling Practice and Theory*, 13(8):703–715, 2005.
- [79] Teschner, M., Kimmerle, S., Heidelberger, B., Zachmann, G., Raghupathi, L., Fuhrmann, A., ... & Volino, P. Collision detection for deformable objects. *Computer Graphics Forum*, 24(1):61–81, 2005.
- [80] Tillet, P., Rupp, K., & Selberherr, S. An automatic opencl compute kernel generator for basic linear algebra operations. In *Proceedings of the 2012 Symposium on High Performance Computing*, page 4. Society for Computer Simulation International, 2012.
- [81] Tomov, S., Dongarra, J., & Baboulin, M. Towards dense linear algebra for hybrid GPU accelerated manycore systems. *Parallel Computing*, 36(5-6):232–240, June 2010.
- [82] Treuille, A., McNamara, A., Popović, Z., & Stam, J. Keyframe control of smoke simulations. In Alyn P. Rockwood, editor, *ACM SIGGRAPH 2003 Papers on - SIGGRAPH '03*, page 716, New York, New York, USA, 2003. ACM Press.

- [83] Wang, B., Wu, L., Yin, K., Ascher, U., Liu, L., & Huang, H. Deformation capture and modeling of soft objects. *ACM Transactions on Graphics*, 34(4):94:1–94:12, 2015.
- [84] Weiss, S. cuMat. <https://gitlab.com/shaman42/cuMat>, 2018.
- [85] Wiemann, P., Wenger, S., & Magnor, M. Cuda expression templates. *WSCG '2011: Communication Papers Proceedings*, pages 185–192, 2011.
- [86] Yadav, P., Suresh, K. Large Scale Finite Element Analysis Via Assembly-Free Deflated Conjugate Gradient. *JOURNAL OF COMPUTING AND INFORMATION SCIENCE IN ENGINEERING*, 14(4), DEC 2014.
- [87] Zhao, Q., Pizer, S., Alterovitz, R., Niethammer, M., & Rosenman, J. Orthotropic thin shell elasticity estimation for surface registration. In Marc Niethammer, Martin Styner, Stephen Aylward, Hongtu Zhu, Ipek Oguz, Pew-Thian Yap, and Dinggang Shen, editors, *Information Processing in Medical Imaging*, volume 10265 of *Lecture Notes in Computer Science*, pages 493–504. Springer International Publishing, Cham, 2017.