Differentiation and High-Performance Infrastructure for Multiphysics Simulation

Von der Fakultät für Informatik der RWTH Aachen University
zur Erlangung des akademischen Grades
eines Doktors der Naturwissenschaften
genehmigte Dissertation

vorgelegt von

José Antonio Fernández-Fernández
M.Sc. RWTH

Berichter: Prof. Dr. Jan Bender
Prof. Dr. Mark Pauly

Tag der mündlichen Prüfung: 04.09.2026

Diese Dissertation ist auf den Internetseiten der Universitätsbibliothek verfügbar.

Abstract

Modern physical simulation is expected to handle increasingly complex requirements: heterogeneous object representations, nonlinear materials, robust contact and friction, differentiability, and runtime efficiency. Optimization-based simulation provides a common language for many of these problems by formulating the balance of forces as the minimization of a global objective, which can be robustly solved with Newton-type methods. However, the path from a compact mathematical model to a rich, robust, fast, and differentiable program remains difficult.

This thesis presents the work developed during my doctoral studies on simulation infrastructure to bridge this gap along two connected threads. The first builds on symbolic representation, differentiation, assembly, and solution of variational multiphysics problems. The second concerns high-performance data structures and algorithms for simulation, with emphasis on concurrency, vectorization, pipelining, and task decomposition. Together, these threads support the design of simulation systems that meet demanding modern requirements while remaining approachable, robust, and efficient.

The thesis first introduces SymX, a simulation-native symbolic framework that turns concise mechanical model definitions into compiled kernels and assembled global derivatives for Newton-type solvers. This infrastructure supports follow-up work, including a strongly coupled simulator for rigid and deformable bodies with frictional contact, a selective strategy for stabilizing Newton solves through local Hessian projection, and differentiable pipelines for inverse simulation. Applications in robotics and digital humans accompany the presentation and validate the methods. Finally, the thesis studies high-performance proximity detection in the context of fluid simulation.

Together, these contributions show that expressive, robust, differentiable, and efficient simulation emerges not from any single technique, but from the careful integration of modeling, optimization, differentiation, and high-performance infrastructure.

Zusammenfassung

Titel: *Differentiation und Hochleistungsinfrastruktur für multiphysikalische Simulationen*

Moderne physikalische Simulationen müssen zunehmend komplexe Anforderungen erfüllen: heterogene Objektrepräsentationen, nichtlineare Materialien, robuste Kontakt- und Reibungsbehandlung, Differenzierbarkeit sowie hohe Laufzeiteffizienz. Optimierungsbasierte Simulationen bieten für viele dieser Probleme eine gemeinsame Grundlage, indem sie das Kräftegleichgewicht als Minimierung einer globalen Zielfunktion formulieren, die sich mit Newton-Verfahren robust lösen lässt. Der Weg von einem kompakten mathematischen Modell zu einem vielseitigen, robusten, schnellen und differenzierbaren Programm bleibt jedoch schwierig.

Diese Dissertation stellt die während meiner Promotion entwickelten Arbeiten an Simulationsinfrastruktur vor, die diese Lücke entlang zweier miteinander verbundener Forschungsrichtungen schließen. Die erste baut auf der symbolischen Darstellung, Differenzierung, Assemblierung und Lösung variationeller Mehrphysikprobleme auf. Die zweite befasst sich mit leistungsfähigen Datenstrukturen und Algorithmen für Simulationen, insbesondere mit nebenläufiger Ausführung, Vektorisierung, Pipelining und der Zerlegung in Teilaufgaben. Zusammen unterstützen beide Forschungsrichtungen die Entwicklung von Simulationssystemen, die anspruchsvolle moderne Anforderungen erfüllen und zugleich zugänglich, robust und effizient bleiben.

Zunächst stellt die Dissertation SymX vor, ein auf Simulationen zugeschnittenes symbolisches Framework, das kompakte Definitionen mechanischer Modelle in kompilierte Rechenkerne und assemblierte globale Ableitungen für Newton-Verfahren überführt. Diese Infrastruktur unterstützt weitere Arbeiten, darunter einen stark gekoppelten Simulator für starre und deformierbare Körper mit Reibungskontakt, ein selektives Verfahren zur Stabilisierung von Newton-Verfahren durch Projektion lokaler Hessematrizen sowie differenzierbare Abläufe für inverse Simulationen. Anwendungen in der Robotik und für digitale Menschen begleiten die Darstellung und validieren die Verfahren. Abschließend untersucht die Dissertation leistungsfähige Verfahren zur Nachbarschaftssuche im Kontext der Fluidsimulation.

Zusammen zeigen diese Beiträge, dass ausdrucksstarke, robuste, differenzierbare und effiziente Simulationen nicht aus einer einzelnen Technik entstehen, sondern aus dem sorgfältigen Zusammenspiel von Modellierung, Optimierung, Differenzierung und leistungsfähiger Simulationsinfrastruktur.

Acknowledgements

I would first like to thank my parents, Celia and Diego, and my sister, Maria, for the foundation on which so much of my life has been built. Thank you for your unconditional love, your patience, your advice, and your care. Thank you for helping me through difficult moments, and for always reminding me where I come from.

To Maria, my love, thank you for walking beside me through the good and the bad, and for bringing love, patience, and warmth to this journey. I am also deeply grateful to the rest of my family and friends for their love, for being there, and for filling life with joy.

I would like to express my deepest gratitude to my supervisor, Prof. Jan Bender, for believing in me throughout all these years. You took me in as a student, gave me the environment to grow, and helped me reach much further than I would ever have expected. This thesis would not exist without your trust, your help, and your support.

I am equally thankful to my colleagues at RWTH Aachen University: Andreas, Fabian, Lukas, Marcel, Stefan, Tassilo, and Timna. Thank you for the discussions, the advice, the laughter, and the many moments we shared along the way. It was a rollercoaster, and you endured more than a fair amount of ranting with more patience than I probably deserved. More broadly, I will always be grateful to RWTH Aachen University, and to Germany as a whole, for welcoming me, supporting my research, and giving me a place to grow.

I would also like to thank Dr. Philipp Herholz and Prof. Miguel Otaduy for giving me the opportunity to intern at Meta Reality Labs. Working with you has had a deep and lasting impact on me. From you I learned to push harder, to raise my standards, and to keep looking for the next meaningful milestone.

To David, German, and Matheus, my intern brothers, thank you for teaching me to keep going, for making difficult moments lighter, and for turning those experiences into something I will always remember with joy.

My sincere thanks also go to Prof. Kai Arras, Dr. Ralph Lange, and the entire team at Bosch Research, who supported the first years of my doctoral studies. Thank you for staying with the project and seeing it through all the way to the end.

Finally, I would like to thank Prof. Mark Pauly for co-supervising this thesis and for his valuable time, attention, and support.

Contents

Abstract i

Zusammenfassung iii

Acknowledgements v

I Introduction 1

1 Introduction 3
1.1 Research Questions . . . 4
1.2 Outline . . . 4
1.3 Publications . . . 5

II Foundations 7

2 Optimization-Based Simulation 9
2.1 Equations of Motion . . . 9
2.2 Optimization Time Integration . . . 10
2.3 Discrete Potentials and Assembly . . . 11
2.4 Newton's Method . . . 11
2.5 Hessian Definiteness and Stabilization . . . 12
2.6 Line Search and Globalization . . . 13
2.7 Convergence and Inexact Newton . . . 14
2.8 Constraints, Penalties, and Barriers . . . 16
2.9 Chapter Summary . . . 17

3 Variational Models for Multiphysics Simulation 19
3.1 From Continuum Models to Discrete Potentials . . . 19
3.2 Kinematics of Deformation . . . 20
3.3 Constitutive Models . . . 21
3.4 The Finite Element Method for Elasticity . . . 22
3.5 Rayleigh Damping . . . 24
3.6 Contact and Friction Potentials . . . 24
3.7 Shells and Cloth . . . 26

3.8 Rigid Bodies . . . 27
3.9 Chapter Summary . . . 28

4 Differentiable Simulation and Sensitivity Analysis 31
4.1 Inverse Problem Setting . . . 31
4.2 Implicit Function Theorem . . . 32
4.3 The Adjoint Method . . . 32
4.4 Optimization in Parameter Space . . . 33

III Scientific Contributions 35

5 SymX: Simulation from Symbolic Expressions 37
5.1 Introduction . . . 38
5.2 Related Work . . . 39
5.3 Problem Definition . . . 41
5.4 SymX Framework . . . 44
5.5 Applications . . . 51
5.6 Benchmarks . . . 60
5.7 Limitations and Future Work . . . 64
5.8 Impact and Adoption . . . 66

6 STARK: Strongly Coupled Rigid and Deformable Simulation 69
6.1 Introduction . . . 70
6.2 Related Work . . . 71
6.3 Concept . . . 73
6.4 Experiments . . . 75
6.5 Conclusions . . . 79
6.6 Impact and Adoption . . . 79

7 Progressively Projected Newton's Method 81
7.1 Introduction . . . 82
7.2 Related Work . . . 83
7.3 Projected Newton and Over-Projection . . . 85
7.4 Progressively Projected Newton's Method . . . 85
7.5 Results . . . 88
7.6 Limitations and Future Work . . . 95
7.7 Conclusion . . . 97

8 Interactive Facial Animation via Differentiable Shell Simulation 99
8.1 Introduction . . . 100
8.2 Related Work . . . 102
8.3 Forward Simulation Pipeline . . . 104
8.4 Material Stiffness Reconstruction . . . 109
8.5 Results . . . 112
8.6 Limitations and Future Work . . . 118
8.7 Conclusions . . . 118

9 Fast Octree Neighborhood Search for SPH Simulations **119**
9.1 Introduction . . . 120
9.2 Related Work . . . 121
9.3 Method . . . 123
9.4 Results . . . 131
9.5 Limitations . . . 138
9.6 Conclusion and Future Work . . . 140

IV Conclusions and Outlook **141**

10 Concluding Remarks **143**
10.1 Conclusion . . . 143
10.2 Outlook . . . 144

Bibliography **147**

Part I

Introduction

CHAPTER 1

Introduction

Physical simulation is a central tool in computer graphics, visual computing, robotics, and related fields. It allows engineers, scientists, artists, and autonomous systems to approximate the behavior of matter under changing conditions in virtual environments. In these environments, digital objects can move, deform, collide, slide, and interact according to physical principles. Such simulations are used to gain insight, communicate ideas, create entertainment media, support manufacturing, and enable learning. In visual computing, these methods bring life to animations, digital humans, virtual environments, and interactive applications. In robotics, they provide a way to study the interaction between machines and the physical world, and for autonomous agents to learn actions such as locomotion and manipulation.

The practical difficulty is that realistic scenes are rarely composed of a single isolated physical model. The simulations considered in this thesis combine rigid bodies, deformable solids, shells, cloth, particles, contact, friction, joints, motors, damping, and actuation. Moreover, it is not sufficient for each subsystem to behave plausibly by itself, the interactions between them must also be resolved consistently. These effects are strongly coupled: the deformation of a soft object changes contact forces, contact forces alter rigid body motion, and rigid body motion in turn changes the loading and deformation of the surrounding objects. If forces and constraints are not transferred correctly between subsystems, the resulting motion may look plausible locally while violating the global mechanics of the coupled system. This motivates strongly coupled multiphysics formulations in which simulations are defined and solved globally.

In addition to physical fidelity, these models must also be compatible with robust and efficient numerical solution. Properties such as continuity, differentiability, conditioning, and definiteness directly affect the behavior of nonlinear solvers. At the same time, the discretized system should retain mechanical correctness, including meaningful treatment of mass, energy, momentum, dissipation, and constraints. Powerful simulation environments therefore need to address modeling, numerical robustness, and implementation together.

This thesis studies this broader problem through the lens of optimization-based multiphysics simulation, with particular emphasis on differentiability and high-performance simulation infrastructure. The central viewpoint is that many physical effects can be expressed as local contributions to a global objective. The derivatives of this objective define the nonlinear systems solved during simulation and the sensitivities required for inverse problems. This

formulation provides a common language for modeling, coupling, solving, and differentiating complex scenes. At the same time, it exposes the infrastructure needed to make these methods practical: derivatives of complex mechanical models, sparse assembly, dynamic topology handling, nonlinear optimization, stabilization and globalization, data management, vectorization and parallelization, and more.

1.1 Research Questions

The thesis is organized around two connected threads. The first is the optimization-based formulation of multiphysics simulation: local physical models are assembled into a global objective, solved with Newton-type methods, and differentiated when inverse problems require sensitivities. The second is performance-oriented simulation infrastructure: the practical value of these formulations depends on derivative-generation tools, sparse assembly, solver design, data structures, task decomposition, and hardware-aware kernels. Specifically, this thesis lays out the answers to the following research questions:

RQ1 How can complex variational simulation models be specified symbolically so that differentiation, compilation, and assembly for Newton-type multiphysics simulation can be automated?

RQ2 How can rigid bodies, deformables, joints, motors, contact, and friction be solved robustly in one strongly coupled variational simulator, with results validated in robotics lab experiments?

RQ3 How can Newton-type solvers stabilize indefinite Hessians while avoiding most local eigendecompositions and preserving useful curvature information?

RQ4 How can differentiable shell simulation recover facial stiffness fields that preserve existing rigs while enabling real-time contact response?

RQ5 How can performance-critical simulation kernels, such as neighborhood and proximity search, be designed around modern CPU architectures rather than theoretical complexity alone?

1.2 Outline

The remainder of this thesis is organized into three parts.

Foundations. Part II introduces the mathematical and numerical background used in the rest of this thesis. Chapter 2 presents optimization-based simulation, deriving implicit time stepping as the minimization of an incremental potential and introducing Newton's method, line search, Hessian stabilization, and convergence criteria. Chapter 3 introduces variational models for multiphysics simulation, including deformation kinematics, constitutive models, finite elements, contact and friction potentials, shells, cloth, and rigid bodies. Chapter 4 introduces differentiable simulation and sensitivity analysis, with emphasis on the implicit function theorem, adjoint gradients, and outer optimization in parameter space.

Scientific contributions. Part III contains the main scientific contributions and answers the research questions listed above. Chapter 5 presents SymX, a simulation-native symbolic differentiation framework that addresses RQ1 by defining complex multiphysics problems with concise expressions and automatically carrying out differentiation, compilation, evaluation, and assembly. Chapter 6 presents STARK, a strongly coupled simulation framework that addresses RQ2 by integrating rigid and deformable bodies with frictional contact, joints, motors, and robotics-oriented validation. Chapter 7 introduces Progressively Projected Newton's Method, which addresses RQ3 through selective Hessian stabilization for Newton-type solvers in dynamic simulation. Chapter 8 presents an interactive facial animation pipeline that addresses RQ4 through differentiable shell simulation, inverse stiffness reconstruction, contact-aware forward simulation, and fine-detail restoration. Chapter 9 presents TreeNSearch, a hardware-aware neighborhood-search method that addresses RQ5 by designing data structures and algorithms for proximity queries around SIMD throughput, branchless computation, and balanced parallel tasks.

Synthesis and outlook. Part IV summarizes the thesis, relates the contributions to the common themes of differentiation and high-performance infrastructure, and discusses future directions.

1.3 Publications

This thesis is primarily based on the following first-authored peer-reviewed publications:

1. **José Antonio Fernández-Fernández**, Fabian Löschner, Lukas Westhofen, Andreas Longva, and Jan Bender. *SymX: Energy-based Simulation from Symbolic Expressions* [Fernández-Fernández et al. 2025b]. Published in *ACM Transactions on Graphics*. Presented at SIGGRAPH.

2. **José Antonio Fernández-Fernández**, Ralph Lange, Stefan Laible, Kai O. Arras, and Jan Bender. *STARK: A Unified Framework for Strongly Coupled Simulation of Rigid and Deformable Bodies with Frictional Contact* [Fernández-Fernández et al. 2024]. Published at the *IEEE International Conference on Robotics and Automation (ICRA)*.

3. **José Antonio Fernández-Fernández**, Fabian Löschner, and Jan Bender. *Progressively Projected Newton's Method* [Fernández-Fernández et al. 2026]. Published in *Computer Graphics Forum* (Eurographics).

4. **José Antonio Fernández-Fernández**, Ryan Goldade, Ladislav Kavan, Jan Bender, and Philipp Herholz. *Interactive Facial Animation: Enhancing Facial Rigs with Real-Time Shell and Contact Simulation* [Fernández-Fernández et al. 2025a]. Published in *Proceedings of the ACM on Computer Graphics and Interactive Techniques*.

5. **José Antonio Fernández-Fernández**, Lukas Westhofen, Fabian Löschner, Stefan Rhys Jeske, Andreas Longva, and Jan Bender. *Fast Octree Neighborhood Search for SPH Simulations* [Fernández-Fernández et al. 2022]. Published in *ACM Transactions on Graphics* (SIGGRAPH Asia).

The following co-authored publications are related to the broader research context of this thesis but are not included in this document:

1. Tassilo Kugelstadt, Jan Bender, **José Antonio Fernández-Fernández**, Stefan Rhys Jeske, Fabian Löschner, and Andreas Longva. *Fast Corotated Elastic SPH Solids with Implicit Zero-Energy Mode Control* [Kugelstadt et al. 2021]. Published in *Proceedings of the ACM on Computer Graphics and Interactive Techniques*.

2. Andreas Longva, Fabian Löschner, Tassilo Kugelstadt, **José Antonio Fernández-Fernández**, and Jan Bender. *Higher-Order Finite Elements for Embedded Simulation* [Longva et al. 2020]. Published in *ACM Transactions on Graphics*.

3. Stefan Rhys Jeske, Lukas Westhofen, Fabian Löschner, **José Antonio Fernández-Fernández**, and Jan Bender. *Implicit Surface Tension for SPH Fluid Simulation* [Jeske et al. 2023]. Published in *ACM Transactions on Graphics*.

4. Andreas Longva, Fabian Löschner, **José Antonio Fernández-Fernández**, Egor Larionov, Uri M. Ascher, and Jan Bender. *Pitfalls of Projection: A Study of Newton-Type Solvers for Incremental Potentials* [Longva et al. 2026]. Published in *ACM Transactions on Graphics*.

5. Fabian Löschner, **José Antonio Fernández-Fernández**, Stefan Rhys Jeske, Andreas Longva, and Jan Bender. *Micropolar Elasticity in Physically-Based Animation* [Löschner et al. 2023]. Published in *Proceedings of the ACM on Computer Graphics and Interactive Techniques*.

6. Fabian Löschner, **José Antonio Fernández-Fernández**, Stefan Rhys Jeske, and Jan Bender. *Curved Three-Director Cosserat Shells with Strong Coupling* [Löschner et al. 2024]. Published in *Computer Graphics Forum*.

7. Lukas Westhofen, **José Antonio Fernández-Fernández**, Stefan Rhys Jeske, and Jan Bender. *Strongly Coupled Simulation of Magnetic Rigid Bodies* [Westhofen et al. 2024]. Published in *Computer Graphics Forum*.

Part II

Foundations

CHAPTER 2

Optimization-Based Simulation

This chapter introduces the optimization-based viewpoint to simulation used throughout much of this thesis. The main idea is to express a discrete time step force balance as the minimization of an incremental potential whose stationary points satisfy the equations of motion. This approach is especially useful in multiphysics settings, where different physical effects such as inertia, elasticity or contact can be written as additive contributions to the same scalar objective and solved in a fully coupled manner. Optimization-based time integration has a long history [Radovitzky and Ortiz 1999] and is now well established in computer graphics [Kharevych et al. 2006; Martin et al. 2011; Liu et al. 2013]. This route includes first-order and quasi-Newton approaches such as Projective Dynamics [Bouaziz et al. 2014; Narain et al. 2016; Overby et al. 2017; Chen et al. 2024a], second-order methods for robust nonlinear solves [Gast et al. 2015], and contact-oriented formulations such as IPC [Li et al. 2020]. It has also been applied across cloth and rods, rigid bodies, advanced materials, fluids, and granular media [Li et al. 2021; Chen et al. 2023; Ferguson et al. 2021; Lan et al. 2022; Löschner et al. 2023, 2024; Xie et al. 2023; Li et al. 2024b]. This chapter introduces the common variational and numerical structure used by the later contribution chapters.

The chapter proceeds from the discrete equations of motion to their variational reformulation, then develops the local-to-global structure of discrete potentials and the main ingredients of Newton-based nonlinear solvers.

2.1 Equations of Motion

We begin with the equations of motion given by Newton's second law for a spatially discretized system:

$$\mathbf{M}\dot{\mathbf{v}} = \mathbf{f}(\mathbf{x}, \mathbf{v}), \qquad \dot{\mathbf{x}} = \mathbf{v}, \tag{2.1}$$

where $\mathbf{x} \in \mathbb{R}^{3n}$ stacks the nodal positions, $\mathbf{v} \in \mathbb{R}^{3n}$ stacks the nodal velocities, $\mathbf{M} \in \mathbb{R}^{3n\times 3n}$ is the mass matrix, and $\mathbf{f} = \mathbf{f}_{\text{int}} + \mathbf{f}_{\text{ext}}$ collects all internal and external forces. Since $\mathbf{f}$ generally depends nonlinearly on the state, this system admits no closed-form solution.

To advance the system in time, we discretize it into a sequence of time steps. We restrict the exposition to Euclidean configuration spaces; rigid body systems admit the same variational viewpoint but require a kinematic update consistent with the chosen rotational

representation. A common time integration scheme is Backward Euler, which approximates the time derivatives at the end of the step as

$$\dot{\mathbf{v}} \approx \frac{\mathbf{v} - \mathbf{v}^{\text{prev}}}{\Delta t}, \qquad \dot{\mathbf{x}} \approx \frac{\mathbf{x} - \mathbf{x}^{\text{prev}}}{\Delta t} = \mathbf{v}. \tag{2.2}$$

Here Δt is the time step size and $(\mathbf{x}^{\text{prev}}, \mathbf{v}^{\text{prev}})$ is the state at the previous time step.

Eliminating $\mathbf{v}$ yields the nonlinear system

$$\frac{1}{\Delta t^2}\mathbf{M}\left(\mathbf{x} - \mathbf{x}^{\text{prev}} - \Delta t\, \mathbf{v}^{\text{prev}}\right) - \mathbf{f}\left(\mathbf{x}\right) = \mathbf{0}. \tag{2.3}$$

A state $\mathbf{x}$ that solves Eq. (2.3) fulfills the discrete balance of forces for the current time step. This system can be solved directly, for instance with Newton's method, but nonlinear root-finding is often fragile: steps may overshoot, enter invalid intermediate states, or fail to converge when started far from the solution [Gast et al. 2015]. These difficulties are particularly pronounced in stiff and strongly nonlinear problems.

2.2 Optimization Time Integration

For an important class of problems, the same step can be recast as a nonlinear minimization, which unlocks robust mechanisms such as sufficient-decrease conditions that help guarantee a solution is found.

Assume the forces aggregate from multiple sources as $\mathbf{f} = \sum_i \mathbf{f}_i$ and are conservative, so that

$$\mathbf{f}_i(\mathbf{x}) = -\nabla \phi_i(\mathbf{x}), \tag{2.4}$$

for some potential energy ϕ_i. Then, the Backward Euler step is equivalent to minimizing the incremental potential

$$E(\mathbf{x}) = \frac{1}{2\Delta t^2}(\mathbf{x} - \tilde{\mathbf{x}})^T \mathbf{M}(\mathbf{x} - \tilde{\mathbf{x}}) + \sum_i \phi_i(\mathbf{x}), \tag{2.5}$$

where $\tilde{\mathbf{x}} = \mathbf{x}^{\text{prev}} + \Delta t \mathbf{v}^{\text{prev}}$ is the inertial position. This equivalence follows from the stationarity condition $\nabla E(\mathbf{x}) = \mathbf{0}$, which recovers Eq. (2.3) for conservative forces. The first term in Eq. (2.5) is often called the *inertial potential*.

The time step update can therefore be written as

$$\mathbf{x} = \arg\min_{\mathbf{x}} E(\mathbf{x}), \qquad \mathbf{v} = \frac{\mathbf{x} - \mathbf{x}^{\text{prev}}}{\Delta t}, \tag{2.6}$$

which is typically solved with Newton-style methods. These require the gradient and Hessian of E, which provide the residual and curvature information needed by the solver.

This *variational* form is very useful in practice. Rather than solving a nonlinear root-finding problem directly, we minimize a scalar objective. This enables the use of globalization techniques such as backtracking line search or trust-region strategies [Nocedal and Wright 2006]. Further, using a symmetric positive definite (SPD) approximation of the Hessian ensures that every step decreases the objective. Together, they make Newton-like methods very robust and widely used.

The energy-based viewpoint also provides a natural interface for multiphysics simulation. Each physical effect contributes an additive term to the same objective, and the total gradient and Hessian are the sums of the corresponding per-effect contributions. This leads to a conceptually simple and implementation-friendly framework in which inertia, elasticity, contact, and other effects can be handled in a uniform manner.

Non-conservative effects, however, do not in general derive from a scalar potential in the unknown state variables. Important examples are friction and damping, which require special treatment such as lagging selected terms.

2.3 Discrete Potentials and Assembly

The variational form introduced above becomes useful only after the physical model has been discretized in space. In practice, continuum quantities such as inertia, elasticity, contact, or boundary interactions are converted into a finite collection of discrete contributions, each depending only on a small subset of the global degrees of freedom. Figure 2.1 illustrates this local-to-global view for a simple discretized body. The resulting objective therefore takes the generic form

$$E(\mathbf{x}) = \sum_{p \in \mathcal{P}} \sum_{e \in \mathcal{T}_p} \phi_p^e(\mathbf{x}^e), \tag{2.7}$$

where $\mathcal{P}$ is the set of different physical effects, $\mathcal{T}_p$ is the set of discrete contributions associated with effect p, and ϕ_p^e is the corresponding local potential. Here $\mathbf{x}^e$ denotes the subset of degrees of freedom on which contribution e depends.

It is convenient to express this dependence through a local extraction operator

$$\mathbf{x}^e = \mathbf{P}^e \mathbf{x}, \tag{2.8}$$

where $\mathbf{P}^e$ selects the relevant entries of the global state vector. For instance, an elastic finite element depends only on the nodes of that element, a contact term depends only on the degrees of freedom of the contacting primitives, and a lumped inertial term may depend only on a single node or body.

This local structure directly induces the assembly formulas for the global derivatives needed to solve the nonlinear problem,

$$\mathbf{g} = \nabla E(\mathbf{x}) = \sum_{p \in \mathcal{P}} \sum_{e \in \mathcal{T}_p} (\mathbf{P}^e)^T \nabla \phi_p^e(\mathbf{x}^e), \tag{2.9}$$

and

$$\mathbf{H} = \nabla^2 E(\mathbf{x}) = \sum_{p \in \mathcal{P}} \sum_{e \in \mathcal{T}_p} (\mathbf{P}^e)^T \nabla^2 \phi_p^e(\mathbf{x}^e) \mathbf{P}^e. \tag{2.10}$$

This additive local-to-global structure is central to optimization-based simulation. It allows different physical effects to be expressed in a uniform way, while enabling efficient evaluation, sparse assembly, and second-order optimization methods.

2.4 Newton's Method

We minimize the incremental potential $E(\mathbf{x})$ with Newton's method. At iteration k, let $\mathbf{x}^k$ denote the current state. Newton's method computes a correction $\Delta \mathbf{x}^k$ by solving

$$\mathbf{H}^k \Delta \mathbf{x}^k = -\mathbf{g}^k, \tag{2.11}$$

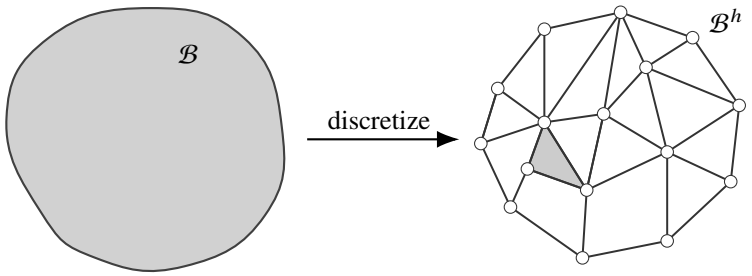


Figure 2.1: *Local-to-global structure of a discretized variational problem. A continuous body $\mathcal{B}$ is approximated by a finite discretization $\mathcal{B}^h$ composed of nodes, edges, and surface elements. Local energy potentials can be defined on these elements, or on selected interaction pairs, and accumulated into the global objective.*

where

$$\mathbf{g}^k = \nabla_{\mathbf{x}} E(\mathbf{x}^k), \qquad \mathbf{H}^k = \nabla^2_{\mathbf{x}} E(\mathbf{x}^k). \tag{2.12}$$

The iterate is then updated as $\mathbf{x}^{k+1} = \mathbf{x}^k + \Delta\mathbf{x}^k$. This process is repeated until a suitable convergence criterion is satisfied.

Equation (2.11) is obtained by minimizing the local quadratic model of the objective around the current iterate. For clarity, we omit the superscript k hereafter. The quadratic model reads

$$m(\Delta\mathbf{x}) = E + \Delta\mathbf{x}^T\mathbf{g} + \frac{1}{2}\Delta\mathbf{x}^T\mathbf{H}\,\Delta\mathbf{x}, \tag{2.13}$$

whose stationary point, given by vanishing first-order derivatives, recovers Eq. (2.11).

Classical Newton's method relies on regularity of the objective: it requires E to be twice differentiable near the iterates, so that both $\mathbf{g} = \nabla E$ and $\mathbf{H} = \nabla^2 E$ are well defined. Under C^2 regularity and a nonsingular Hessian, the method converges superlinearly near the solution. In practice this is not strict, as Newton's method often converges for objectives with only piecewise continuous second derivatives, albeit at a reduced rate.

2.5 Hessian Definiteness and Stabilization

A central requirement for Newton's method is that the search direction be a *descent* direction. Consider again the Newton system in Eq. (2.11). If the Hessian $\mathbf{H}$ is symmetric positive definite (SPD), that is,

$$\mathbf{p}^T\mathbf{H}\,\mathbf{p} > 0, \qquad \forall\,\mathbf{p} \neq \mathbf{0}, \tag{2.14}$$

then the local quadratic model is strictly convex and the Newton direction is guaranteed to be a descent direction. Indeed, multiplying the Newton equation (2.11) by $\Delta\mathbf{x}^T$ gives

$$\Delta\mathbf{x}^T\mathbf{g} = -\Delta\mathbf{x}^T\mathbf{H}\,\Delta\mathbf{x}. \tag{2.15}$$

If $\mathbf{H}$ is SPD, the right-hand side is strictly negative, hence

$$\Delta\mathbf{x}^T\mathbf{g} < 0, \tag{2.16}$$

and $\Delta\mathbf{x}$ is a descent direction.

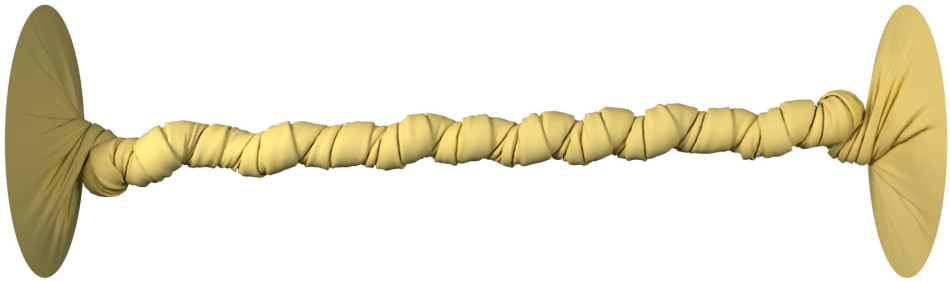


Figure 2.2: *A cylindrical cloth is twisted by rotating both ends in opposite directions. Despite the strong nonlinearities and intricate contact geometry, line search maintains admissibility throughout the solve and enables robust convergence of Newton's method.*

In practice, however, Hessians arising in physical simulation are often indefinite due to material nonlinearities, contact, or large deformations. For this reason, it is common to replace the true Hessian by a positive definite approximation, guaranteeing descent directions while preserving as much of the original curvature as possible. This process is referred to as *projection to positive definiteness*.

To achieve this efficiently, a standard approach is to filter, for instance by clamping, the negative eigenvalues of the local Hessian contributions prior to assembly. Given the eigendecomposition of a local contribution

$$\mathbf{H}^e = \mathbf{Q}^e \, \mathbf{D}^e \left(\mathbf{Q}^e\right)^T, \tag{2.17}$$

where $\mathbf{Q}^e$ is an orthogonal matrix whose columns are the eigenvectors of $\mathbf{H}^e$ and $\mathbf{D}^e$ is a diagonal matrix containing the corresponding eigenvalues. Its stabilized counterpart is

$$\widehat{\mathbf{H}}^e = \mathbf{Q}^e \, \widehat{\mathbf{D}}^e \left(\mathbf{Q}^e\right)^T, \tag{2.18}$$

with

$$\widehat{D}^e_{ii} = \max(D^e_{ii}, \, \varepsilon), \tag{2.19}$$

for some small $\varepsilon > 0$. Since a sum of SPD matrices is SPD, the assembled global matrix is also SPD. This approach is commonly referred to as *Projected Newton* [Teran et al. 2005].

This stabilization is essential for robustness, but it also modifies the original curvature of the problem. More selective strategies therefore try to apply projection only when needed, while others replace simple eigenvalue clamping with more refined spectral filters. A more detailed discussion on this matter is given in Chapter 7, together with the presentation of the *Progressively Projected Newton* method.

2.6 Line Search and Globalization

Even when $\Delta\mathbf{x}$ is a descent direction, the full Newton step may not be reliable. Far from the minimizer, the quadratic model can be inaccurate, and the update may either increase the objective or leave the admissible set. This phenomenon is often referred to as *overshooting*, and is particularly common in strongly nonlinear simulation problems.

To control the update, we replace the full Newton step by

$$\mathbf{x}^{k+1} = \mathbf{x}^k + \alpha^k \Delta \mathbf{x}^k, \tag{2.20}$$

where the step length $\alpha^k \in (0, 1]$ is selected by line search. A common strategy is *backtracking*: starting from $\alpha^k = 1$, corresponding to the full Newton step, the step length is repeatedly reduced until suitable acceptance conditions are satisfied. The full Newton's method logic can be found in Algorithm 1.

Admissibility. The first requirement is that the steps remain in the domain where the objective is meaningful and the physical model remains valid. In simulation, inadmissible states may include interpenetrating configurations, inverted elements, or arguments outside the domain of barrier or logarithmic terms. Admissibility can be enforced by backtracking, but also by *truncation*, where the step is clipped explicitly using the intersection between the step and the admissible domain.

Sufficient decrease. Among admissible steps, we seek one that decreases the objective by an amount consistent with the predicted local descent. A common criterion is the *Armijo condition*

$$E(\mathbf{x} + \alpha \Delta \mathbf{x}) \leq E + c_1 \, \alpha \, \Delta \mathbf{x}^T \mathbf{g}, \tag{2.21}$$

where $c_1 \in (0, 1)$ is a small constant. Since $\Delta \mathbf{x}$ is a descent direction, Eq. (2.21) requires the objective to decrease by at least a fixed fraction of the decrease predicted by the local linear model.

Second-order information gives Newton's method fast local convergence, while admissibility and sufficient decrease are the robustness mechanisms that keep iterates valid. Together, they make the optimization-based approach practical, ensuring reliable progress of the nonlinear solve even in highly nonlinear and numerically stiff settings such as the example in Figure 2.2.

Algorithm 1: Newton's method for minimizing the incremental potential.

$\mathbf{x} \leftarrow \mathbf{x}^{\text{prev}} + \Delta t \, \mathbf{v}^{\text{prev}}$
while *not converged* **do**
  assemble $\mathbf{g}(\mathbf{x})$ and $\mathbf{H}(\mathbf{x})$
  solve $\mathbf{H}(\mathbf{x}) \, \Delta \mathbf{x} = -\mathbf{g}(\mathbf{x})$
  find α by line search
  $\mathbf{x} \leftarrow \mathbf{x} + \alpha \Delta \mathbf{x}$
$\mathbf{v} \leftarrow (\mathbf{x} - \mathbf{x}^{\text{prev}})/\Delta t$

2.7 Convergence and Inexact Newton

In simulation, Newton's method is rarely driven to very high accuracy. Instead, the nonlinear solve is terminated once the remaining error is small enough for the intended application. This is a practical trade-off: solving too accurately increases cost with little benefit, while terminating too early may leave significant force imbalance. In multiphysics settings, the choice of convergence criterion is further complicated by large stiffness and density contrasts.

Two criteria are particularly common. The first is the *residual norm*,

$$\left\|\mathbf{g}^k\right\| < \varepsilon_{\text{res}}, \tag{2.22}$$

which measures proximity to stationarity and directly reflects the remaining force imbalance. The second is the *step norm*,

$$\left\|\Delta\mathbf{x}^k\right\| < \varepsilon_{\text{step}}, \tag{2.23}$$

which measures the size of the Newton correction. The residual norm is the more direct criterion, but in heterogeneous systems its scale can be difficult to interpret. This issue is particularly pronounced in strongly coupled scenes with large mass and stiffness contrasts, such as the dryer example in Figure 2.3. The step norm is therefore often more practical, since Eq. (2.11) shows that the Newton correction already incorporates the local scaling induced by mass and stiffness, and thus often provides a more consistent measure of the remaining error.



Figure 2.3: *The drum of a tumble dryer rotates with eight pieces of cloth inside. The nonlinear solve must handle widely different stiffness and density scales within a single coupled system.*

2.7.1 Inexact Newton Solves

When the Newton system is solved iteratively, it is usually unnecessary to solve it to high accuracy at every outer iteration. Far from the minimizer, the quadratic model is only a rough local approximation, so an expensive high-accuracy linear solve is often wasteful. Near the solution, in contrast, more accurate linear solves are desirable in order to recover the fast local convergence of Newton's method.

This idea is commonly formalized through an *inexact Newton* condition of the form

$$\left\|\mathbf{H}^k\Delta\mathbf{x}^k + \mathbf{g}^k\right\| \le \eta^k \left\|\mathbf{g}^k\right\|, \tag{2.24}$$

where $\eta^k \in [0, 1)$ is a *forcing term* controlling the accuracy of the inner linear solve. A common choice is

$$\eta^k = \min\left(0.5, \sqrt{\left\|\mathbf{g}^k\right\|}\right), \tag{2.25}$$

which yields coarse solves far from the minimizer and increasingly accurate solves near convergence [Nocedal and Wright 2006].

In practice, many implementations use a simpler surrogate and terminate the inner iterative solve once its relative residual has been reduced by a fixed factor, for example

$$\left\|\mathbf{r}_j\right\| \leq \eta \left\|\mathbf{r}_0\right\|, \tag{2.26}$$

with a constant such as $\eta = 10^{-4}$.

2.8 Constraints, Penalties, and Barriers

Many simulation problems involve restrictions that cannot be expressed by smooth unconstrained energies alone. Examples include non-penetration, joint limits, unilateral contact, or inequality bounds on deformation. In principle, such conditions can be formulated as a constrained optimization problem. In practice, however, handling constraints directly is often cumbersome within Newton-based time integration, as it introduces complementarity conditions, changing active sets, or saddle-point systems that are more difficult to solve robustly and efficiently.

Within an energy-based formulation, constraints are often incorporated through additional *penalty* or *barrier* terms. This effectively softens their treatment, with restoring forces derived from constraint violations. As long as the resulting objective remains sufficiently smooth, this approach preserves the unconstrained optimization structure developed above, and therefore retains the robustness and convergence properties of Newton-based minimization.

Let $C(\mathbf{x})$ be a scalar constraint function such that $C(\mathbf{x}) = 0$ corresponds to no violation of the constraint. A penalty potential can then be defined as

$$\phi_c(\mathbf{x}) = \frac{k_c}{2} C(\mathbf{x})^2, \tag{2.27}$$

where $k_c > 0$ is the penalty stiffness. The corresponding force is then

$$\mathbf{f}_c(\mathbf{x}) = -k_c\, C(\mathbf{x})\, \nabla C(\mathbf{x}). \tag{2.28}$$

Thus, the larger the constraint violation, the stronger the corrective force.

For inequality constraints, however, corrective forces should act only when the constraint is violated. If the feasible set is defined by $C(\mathbf{x}) \geq 0$, the corresponding penalty may be written as

$$\phi_c(\mathbf{x}) = \begin{cases} 0, & C(\mathbf{x}) \geq 0, \\ -\frac{k_c}{3} C(\mathbf{x})^3, & C(\mathbf{x}) < 0. \end{cases} \tag{2.29}$$

The cubic term provides a C^2-continuous transition at activation, which is generally beneficial for the convergence of Newton's method.

In some cases, the constraint defines an additional *hard limit* of the admissible state, as in contact with non-penetration guarantees. For such constraints, logarithmic barrier potentials are often used, since they can produce arbitrarily large corrective forces as the constraint approaches the boundary of admissibility, at the price of increased nonlinearity.

2.9 Chapter Summary

This chapter introduced the optimization-based viewpoint that underlies the remainder of the thesis. Starting from the discrete equations of motion, we showed how an implicit time step can be reformulated, for conservative systems, as the minimization of an incremental potential. This variational form provides a unified language for multiphysics simulation, in which inertia, elasticity, contact, and other effects contribute through additive local potentials that are assembled into a global objective, gradient, and Hessian.

We then presented the main numerical ingredients used to solve these problems in practice. Newton's method supplies a local second-order model of the objective, Hessian stabilization provides descent directions when the true curvature is indefinite, and line search globalizes the method by enforcing admissibility and sufficient decrease. Finally, convergence criteria, inexact Newton solves, and the use of penalties and barriers were discussed as the practical mechanisms that make the overall approach robust and efficient in complex simulations.

CHAPTER 3

Variational Models for Multiphysics Simulation

The previous chapter introduced the optimization viewpoint at the level of a generic incremental potential. We now turn to the concrete variational models from which such potentials are built. Any physical effect that admits an energy or scalar potential can be combined into a strongly coupled multiphysics simulator by summing its contribution, together with any interaction potentials, into the global incremental potential of Equation (2.5). Such models then inherit the numerical methods and stability properties outlined before. The following gives a brief overview of the models most relevant to this thesis.

3.1 From Continuum Models to Discrete Potentials

Material modeling usually starts from a continuum description: a body occupies a reference domain, its deformation is a map from the reference to the current configuration, and its mechanical response is defined through energies, stresses, or virtual work. In a variational setting, this response is expressed through an energy functional over the reference domain. For an elastic body it typically takes the form

$$\Phi(\mathbf{x}) = \int_{\Omega_0} \psi\big(\mathbf{x}, \mathbf{X}\big)\, \mathrm{d}V, \tag{3.1}$$

where Ω_0 is the reference configuration, $\mathbf{x}$ the deformed (current) state, $\mathbf{X}$ the material (reference) coordinates, and ψ a strain energy density. The functional (3.1) measures the total elastic energy of the object. The dependence of ψ on the state is written abstractly here; in practice it is expressed through kinematic quantities derived from the deformation, most importantly the deformation gradient $\mathbf{F}$ introduced next.

To evaluate this functional and its derivatives in practice, the reference domain is partitioned into a set of finite elements $\mathcal{T}$,

$$\Omega_0 = \bigcup_{e \in \mathcal{T}} \Omega_0^e, \tag{3.2}$$

so that the global energy can be written as a sum of element contributions,

$$\Phi(\mathbf{x}) = \sum_{e \in \mathcal{T}} \int_{\Omega_0^e} \psi\big(\mathbf{x}, \mathbf{X}\big)\, \mathrm{d}V. \tag{3.3}$$

Each element integral is then typically approximated by numerical quadrature.

3.2 Kinematics of Deformation

We now make explicit the kinematic quantities through which the strain energy density is defined. A material point $\mathbf{X}$ is mapped to a spatial position $\mathbf{x}$ by the deformation map

$$\mathbf{x} = \varphi(\mathbf{X}), \tag{3.4}$$

where $\varphi : \Omega_0 \to \Omega$ maps the reference domain to the current configuration Ω. Equivalently, the displacement field is

$$\mathbf{u}(\mathbf{X}) = \varphi(\mathbf{X}) - \mathbf{X}. \tag{3.5}$$

The fundamental local measure of deformation is the *deformation gradient*

$$\mathbf{F} = \frac{\partial \mathbf{x}}{\partial \mathbf{X}} = \nabla_{\mathbf{X}} \varphi(\mathbf{X}). \tag{3.6}$$

It maps an infinitesimal reference segment to its deformed counterpart,

$$\mathrm{d}\mathbf{x} = \mathbf{F}\, \mathrm{d}\mathbf{X}. \tag{3.7}$$

A particularly important quantity is the determinant

$$J = \det(\mathbf{F}), \tag{3.8}$$

which measures the local change of volume,

$$\mathrm{d}v = J\, \mathrm{d}V, \tag{3.9}$$

so that $J > 1$ is local expansion and $0 < J < 1$ compression, while $J < 0$ signals material inversion and typically an inadmissible state. In two dimensions the same relation measures change of area.

Rigid translations do not change $\mathbf{F}$, and rigid rotations change $\mathbf{F}$ without changing the intrinsic strain. Most constitutive models are therefore built from rotation-invariant measures derived from $\mathbf{F}$ rather than from $\mathbf{F}$ itself. A standard one is the *right Cauchy–Green tensor*

$$\mathbf{C} = \mathbf{F}^T \mathbf{F}, \tag{3.10}$$

which compares squared lengths and angles before and after deformation while filtering out rigid rotations. The *Green–Lagrange strain tensor* is then

$$\mathbf{E} = \frac{1}{2}\,(\mathbf{C} - \mathbf{I}) = \frac{1}{2}\left(\mathbf{F}^T \mathbf{F} - \mathbf{I}\right), \tag{3.11}$$

which measures deviation from the undeformed state, vanishes under rigid motion, and is a natural strain measure for nonlinear elasticity.

For small deformations, one can instead use the linearized strain from the displacement gradient,

$$\varepsilon = \frac{1}{2}\left(\nabla_{\mathbf{X}}\mathbf{u} + \nabla_{\mathbf{X}}\mathbf{u}^T\right), \tag{3.12}$$

the infinitesimal strain tensor. It is valid only near the undeformed configuration and is not invariant under large rotations, but is often sufficient for small-strain models.

Further important measures are the following invariants

$$I_1 = \mathrm{tr}(\mathbf{C}), \qquad I_2 = \tfrac{1}{2}\big(\mathrm{tr}(\mathbf{C})^2 - \mathrm{tr}(\mathbf{C}^2)\big), \qquad I_3 = \det(\mathbf{C}) = J^2. \tag{3.13}$$

which are frequently used to define isotropic hyperelastic materials.

3.3 Constitutive Models

The kinematic quantities above describe how a body deforms locally, while a constitutive model specifies how the material responds. This response is encoded by a deformation energy density

$$\psi = \psi(\mathbf{F}), \tag{3.14}$$

from which stresses follow by differentiation. In the following, μ and λ denote the Lamé parameters.

3.3.1 Saint Venant–Kirchhoff Elasticity

A classical nonlinear model is Saint Venant–Kirchhoff elasticity, defined in terms of the Green–Lagrange strain tensor,

$$\psi_{\mathrm{StVK}}(\mathbf{F}) = \mu\ \|\mathbf{E}\|_F^2 + \frac{\lambda}{2}\,\mathrm{tr}(\mathbf{E})^2. \tag{3.15}$$

It is simple and widely used, but remains reliable only in moderate deformation regimes.

3.3.2 Neo-Hookean Elasticity

A standard compressible Neo-Hookean model is

$$\psi_{\mathrm{NH}}(\mathbf{F}) = \frac{\mu}{2}\,(I_1 - 3) - \mu \ln J + \frac{\lambda}{2}(\ln J)^2. \tag{3.16}$$

This model captures nonlinear elastic response in a compact form, but is only defined for $J > 0$.

3.3.3 Stable Neo-Hookean Elasticity

A stabilized Neo-Hookean variant is [Smith et al. 2018]

$$\psi_{\mathrm{SNH}}(\mathbf{F}) = \frac{\mu'}{2}\,(I_1 - 3) + \frac{\lambda'}{2}(J - \alpha)^2 - \frac{\mu'}{2}\ln(I_1 + 1), \tag{3.17}$$

with

$$\mu' = \frac{4}{3}\mu, \qquad \lambda' = \lambda + \frac{5}{6}\mu, \qquad \alpha = 1 + \frac{\mu'}{\lambda'} - \frac{\mu'}{4\lambda'}. \tag{3.18}$$

Compared to the standard Neo-Hookean model, this form improves robustness under strong compression and near-degenerate configurations.

3.3.4 As-Rigid-As-Possible Elasticity

The as-rigid-as-possible model [Sorkine and Alexa 2007] measures deformation relative to the closest rotation

$$\mathbf{R} = \underset{\mathbf{Q}\in SO(3)}{\arg\min} \|\mathbf{F} - \mathbf{Q}\|_F^2 , \tag{3.19}$$

typically obtained from the polar decomposition of $\mathbf{F}$. The resulting energy is

$$\psi_{\mathrm{ARAP}}(\mathbf{F}) = \mu \ \|\mathbf{F} - \mathbf{R}\|_F^2 . \tag{3.20}$$

This energy favors locally rigid motion and penalizes stretching and shearing relative to the best local rotation.

3.3.5 Fixed Corotated Elasticity

The fixed corotated model uses a rotation-relative distortion term together with a volumetric term [Stomakhin et al. 2012]. One common form is

$$\psi_{\mathrm{FC}}(\mathbf{F}) = \mu \ \|\mathbf{F} - \mathbf{R}\|_F^2 + \frac{\lambda}{2}(J-1)^2, \tag{3.21}$$

where $\mathbf{R}$ is the closest rotation to $\mathbf{F}$.

3.4 The Finite Element Method for Elasticity

The continuum elastic potential becomes a finite-dimensional function once the body is discretized into elements and the deformation is interpolated from nodal degrees of freedom. In a Lagrangian setting the mesh is attached to the material, so the unknowns are the current nodal positions, while the fixed reference positions define the undeformed geometry.

Let $e \in \mathcal{T}$ be an element with n_v local nodes. We denote its nodal positions in the reference and current configurations by $\mathbf{X}_a^e$ and $\mathbf{x}_a^e$, respectively, for $a = 1, \ldots, n_v$. Using shape functions N_a, the deformation map restricted to the element is approximated as

$$\varphi^e(\mathbf{X}) = \sum_{a=1}^{n_v} N_a(\mathbf{X})\, \mathbf{x}_a^e. \tag{3.22}$$

Equivalently, the displacement field is interpolated as

$$\mathbf{u}^e(\mathbf{X}) = \sum_{a=1}^{n_v} N_a(\mathbf{X})\, \mathbf{u}_a^e. \tag{3.23}$$

This gives a piecewise polynomial approximation of the deformation map over the body. It is often convenient to express the interpolation through a master element with coordinates ξ, as illustrated in Figure 3.1. This is known as the isoparametric concept. The mapping from the master element to the reference element is

$$\mathbf{X}^e(\xi) = \sum_{a=1}^{n_v} N_a(\xi)\, \mathbf{X}_a^e, \tag{3.24}$$

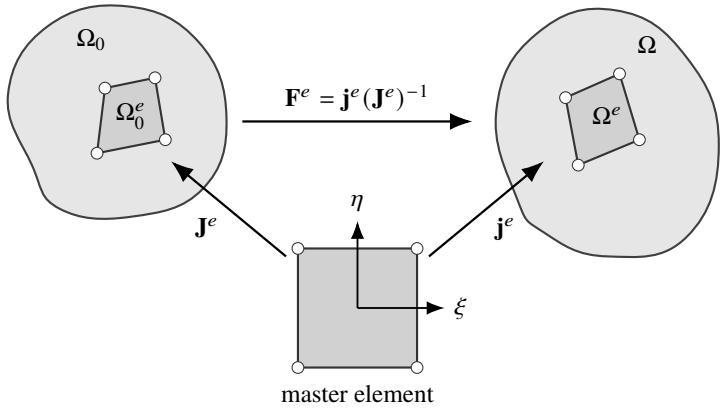


Figure 3.1: *Isoparametric mapping used to compute deformation gradients. The arrows represent the mappings from the master element to the reference and current elements. Their Jacobians,* $\mathbf{J}^e(\xi)$ *and* $\mathbf{j}^e(\xi)$*, map differential vectors in master coordinates to differential vectors in the reference and current configurations. The deformation gradient* $\mathbf{F}^e(\xi) = \mathbf{j}^e(\xi)\,(\mathbf{J}^e(\xi))^{-1}$ *is obtained pointwise in* ξ*.*

and its reference Jacobian is

$$\mathbf{J}^e(\xi) = \frac{\partial \mathbf{X}^e}{\partial \xi}. \tag{3.25}$$

Similarly, the mapping from the master element to the current element is

$$\mathbf{x}^e(\xi) = \sum_{a=1}^{n_v} N_a(\xi)\,\mathbf{x}_a^e. \tag{3.26}$$

Its current Jacobian is

$$\mathbf{j}^e(\xi) = \frac{\partial \mathbf{x}^e}{\partial \xi}. \tag{3.27}$$

Assuming that the reference mapping is locally invertible, the deformation gradient follows from the chain rule as

$$\mathbf{F}^e(\xi) = \frac{\partial \mathbf{x}^e}{\partial \mathbf{X}^e} = \mathbf{j}^e(\xi)\,(\mathbf{J}^e(\xi))^{-1}. \tag{3.28}$$

Thus, $\mathbf{F}^e$ is obtained by composing the differential map from the master element to the current element with the inverse differential map from the master element to the reference element.

In general, the deformation gradient varies within the element. For linear simplex elements, however, it is constant. For a d-dimensional simplex with local nodes indexed by $0, \ldots, d$, the edge matrices are defined as

$$\mathbf{D}_s^e = \begin{bmatrix} \mathbf{x}_1^e - \mathbf{x}_0^e & \cdots & \mathbf{x}_d^e - \mathbf{x}_0^e \end{bmatrix}, \qquad \mathbf{D}_m^e = \begin{bmatrix} \mathbf{X}_1^e - \mathbf{X}_0^e & \cdots & \mathbf{X}_d^e - \mathbf{X}_0^e \end{bmatrix}, \tag{3.29}$$

and the deformation gradient can be written compactly as

$$\mathbf{F}^e = \mathbf{D}_s^e (\mathbf{D}_m^e)^{-1}. \tag{3.30}$$

Once $\mathbf{F}^e$ is available, the elastic energy of the element is obtained by integrating the deformation energy density over the reference element,

$$\phi_{\mathrm{el}}^e(\mathbf{x}^e) = \int_{\Omega_0^e} \psi(\mathbf{F}^e(\mathbf{X}))\,\mathrm{d}V. \tag{3.31}$$

Equivalently, using the master element $\hat{\Omega}$, the same energy can be written as

$$\phi^e_{\text{el}}(\mathbf{x}^e) = \int_{\hat{\Omega}} \psi(\mathbf{F}^e(\xi)) \det \mathbf{J}^e(\xi) \, \mathrm{d}\xi. \tag{3.32}$$

This integral is typically evaluated by numerical quadrature,

$$\phi^e_{\text{el}}(\mathbf{x}^e) \approx \sum_{q=1}^{Q_e} \hat{w}_q \, \det \mathbf{J}^e(\xi_q) \, \psi\big(\mathbf{F}^e(\xi_q)\big), \tag{3.33}$$

where ξ_q and $\hat{w}_q$ are quadrature points and weights on the master element. The global elastic potential is then

$$\phi_{\text{el}}(\mathbf{x}) = \sum_{e \in \mathcal{T}} \phi^e_{\text{el}}(\mathbf{x}^e). \tag{3.34}$$

Here $\mathbf{x}^e$ denotes the subset of global degrees of freedom associated with element e.

3.5 Rayleigh Damping

Not all forces are conservative. Damping forces depend on velocity and therefore do not derive from a potential in the unknown positions alone. A common model that can nonetheless be incorporated into the variational framework through lagging is Rayleigh damping.

Rayleigh damping defines the damping force as a linear combination of mass- and stiffness-proportional terms,

$$\mathbf{f}_d = -(\alpha \mathbf{M} + \beta \mathbf{K}(\mathbf{x}))\, \mathbf{v}, \tag{3.35}$$

where $\alpha, \beta \geq 0$ are the mass and stiffness damping coefficients, and $\mathbf{K}(\mathbf{x}) = \nabla^2 \phi_{\text{el}}(\mathbf{x})$ is the elastic stiffness matrix. Since $\mathbf{K}$ depends on positions, the force cannot be directly integrated into a potential. Gast et al. [2015] resolve this by lagging the stiffness matrix, evaluating it at the positions from the start of the time step,. Using the Backward Euler substitution $\mathbf{v} = (\mathbf{x} - \mathbf{x}^{\text{prev}})/\Delta t$, the corresponding incremental potential term becomes

$$\phi_{\text{damp}}(\mathbf{x}) = \frac{1}{2\,\Delta t} (\mathbf{x} - \mathbf{x}^{\text{prev}})^T (\alpha \mathbf{M} + \beta \mathbf{K}(\mathbf{x}^{\text{prev}}))\, (\mathbf{x} - \mathbf{x}^{\text{prev}}). \tag{3.36}$$

3.6 Contact and Friction Potentials

The coupling of deformable solids to each other and to other bodies is typically realized through frictional contact. In the variational framework, contact is expressed as an interaction potential that depends on the distances between surface primitives.

3.6.1 Barrier Contact Potentials

The *Incremental Potential Contact* (IPC) method [Li et al. 2020] is a fully implicit, unconstrained formulation of contact. It defines admissibility in terms of unsigned distances $d_k(\mathbf{x})$ between non-adjacent surface primitive pairs $k \in C$, where C is the set of all non-incident point-triangle and non-adjacent edge-edge pairs on the surface mesh.

Contact is incorporated to the potential sum in Eq. (2.5) with a sum of barrier potentials (see Section 2.8) over all primitive pairs,

$$\phi_{\text{contact}}(\mathbf{x}) = \kappa \sum_{k \in C} b\Big(d_k(\mathbf{x}), \hat{d}\Big), \tag{3.37}$$

where $\kappa > 0$ is an adaptively chosen barrier stiffness, $\hat{d} > 0$ is a distance threshold controlling geometric accuracy, and $b(d, \hat{d})$ is a C^2-continuous, smoothly clamped log-barrier,

$$b(d, \hat{d}) = \begin{cases} -(d - \hat{d})^2 \ln\left(\dfrac{d}{\hat{d}}\right), & 0 < d < \hat{d}, \\ 0, & d \geq \hat{d}. \end{cases} \tag{3.38}$$

This barrier is exactly zero for $d \geq \hat{d}$, allowing efficient pruning of distant pairs, and tends to infinity as $d \to 0$, generating arbitrarily large repulsive forces to prevent intersection. Only the *active set* $\hat{C}(\mathbf{x}) = \{k \in C : d_k(\mathbf{x}) \leq \hat{d}\}$ contributes non-zero terms, making evaluation efficient despite the quadratic number of potential pairs.

To maintain the intersection-free invariant throughout all Newton iterations, IPC augments the backtracking line search (Section 2.6) with *continuous collision detection* (CCD): before accepting a step $\alpha\Delta\mathbf{x}$, CCD determines the largest feasible step size α_{CCD} such that no primitive pair intersects along the linear trajectory from $\mathbf{x}$ to $\mathbf{x} + \alpha_{\text{CCD}}\Delta\mathbf{x}$.

3.6.2 Variational Friction

IPC also provides a variational treatment of Coulomb friction based on the *Maximal Dissipation Principle* (MDP): frictional forces maximize the rate of dissipation in relative tangential motion subject to the Coulomb constraint. For each active contact pair $k \in \hat{C}$, a tangential sliding basis $\mathbf{T}_k(\mathbf{x}) \in \mathbb{R}^{3\times 2}$ is constructed in the plane orthogonal to the contact normal. The local relative sliding is $\mathbf{u}_k = \mathbf{T}_k(\mathbf{x})^T(\mathbf{x} - \mathbf{x}^{\text{prev}})$.

Two challenges prevent direct inclusion of friction forces in unconstrained optimization: the non-smoothness at the stick-slip transition, and the absence of a potential whose spatial gradient generates friction forces. To address the first, IPC introduces a C^1-smooth approximation of the friction magnitude using a velocity threshold $\epsilon_v > 0$. For the second, IPC lags the tangential sliding basis $\mathbf{T}_k^n$ and contact force magnitude λ_k^n at fixed previous values, so the friction contribution can be written as a potential,

$$\phi_{\text{friction}}(\mathbf{x}) = \sum_{k \in \hat{C}} \mu \lambda_k^n \gamma(\|\mathbf{u}_k\|), \tag{3.39}$$

where μ is the Coulomb friction coefficient and

$$\gamma(y) = \begin{cases} -\dfrac{y^3}{3\hat{y}^2} + \dfrac{y^2}{\hat{y}} + \dfrac{\hat{y}}{3} & \text{if } y \leq \hat{y} \\ y & \text{if } y > \hat{y}, \end{cases} \tag{3.40}$$

for $\hat{y} = \Delta t \epsilon_v$. The negative gradient of this lagged dissipation potential recovers the smoothed Coulomb friction force used in the IPC model.

3.7 Shells and Cloth

Codimensional bodies such as cloth, shells, and rods are an important aspect of multimaterial simulations. They too can be described by energies that enter as additive contributions to the incremental potential.

3.7.1 Shell and Cloth Energies

For cloth and thin shells, the total energy combines a membrane deformation model for in-plane forces with a bending energy for out-of-plane deformations:

$$\phi_{\text{shell}}(\mathbf{x}) = \phi_{\text{membrane}}(\mathbf{x}) + \phi_{\text{bend}}(\mathbf{x}). \tag{3.41}$$

The membrane term ϕ_{membrane} is obtained by integrating a hyperelastic energy density over the surface. The constitutive models from Section 3.3, such as the Stable Neo-Hookean model [Smith et al. 2018], can be applied directly to surface triangles by restricting the deformation gradient $\mathbf{F}$ to the local tangent plane.

For bending, a common model is the *Discrete Shells* model [Grinspun et al. 2003] which defines a discrete flexural energy by measuring the change in dihedral angles along mesh edges,

$$\phi_{\text{bend}}(\mathbf{x}) = k_B \sum_{e \in \mathcal{T}} \frac{\|\bar{\mathbf{e}}\|}{\bar{h}_e} \left(\theta_e(\mathbf{x}) - \bar{\theta}_e\right)^2, \tag{3.42}$$

where θ_e and $\bar{\theta}_e$ are the dihedral angles at edge e in the deformed and rest configurations, respectively, k_B is the bending stiffness, and the ratio $\|\bar{\mathbf{e}}\|/\bar{h}_e$ of the rest edge length to the average height of the incident triangles ensures correct integration of the bending energy over the surface area. By storing rest angles $\bar{\theta}_e$, this model naturally supports curved rest configurations.

An alternative is the quadratic bending model by Bergou et al. [2006], which can be used for flat rest configurations, also defined on each pair of adjacent triangles. For an internal edge e, let $\mathbf{x}^e \in \mathbb{R}^{12}$ stack the positions of the four unique vertices in the two-triangle stencil. The bending energy is written as

$$\phi_b^e(\mathbf{x}^e) = \frac{k_b}{2} (\mathbf{x}^e)^T \mathbf{Q}_e \mathbf{x}^e, \tag{3.43}$$

where k_b is a bending stiffness coefficient and $\mathbf{Q}_e \in \mathbb{R}^{12\times 12}$ is a quadratic form determined by the rest geometry of the stencil. This model is purely quadratic in the vertex positions, requires no trigonometric functions, and results in constant Hessian contributions.

3.7.2 Strain Limiting

Physical cloth strongly resists large in-plane deformations, and many fabrics are nearly inextensible. Using a very stiff membrane model to achieve this leads to *membrane locking*, a numerical artifact where the material appears artificially resistant to bending [Li et al. 2021]. A common workaround is to use a relatively soft membrane energy and impose *strain limits* that prevent excessive stretching.

In the C-IPC approach [Li et al. 2021], the singular values σ_i of the deformation gradient of each triangle are constrained below an upper bound s, enforced through a C^2-continuous

barrier analogous to the contact barrier from Eq. (3.37),

$$\phi_{\mathrm{SL}}(\mathbf{x}) = \kappa_s \sum_{t \in \mathcal{T}} \sum_i V^t \, b\big(s - \sigma_i^t,\ \hat{d}_s\big), \tag{3.44}$$

where $\kappa_s > 0$ is the strain limiting stiffness and $\hat{d}_s > 0$ is the distance threshold for strain limiting.

3.8 Rigid Bodies

Rigid bodies are another key component of multiphysics simulations. Unlike deformable bodies, they do not require a volumetric discretization. For a rigid body, let

$$\underline{\mathbf{x}} = \begin{bmatrix} \mathbf{r} \\ \mathbf{q} \end{bmatrix} \in \mathbb{R}^7, \qquad \underline{\mathbf{v}} = \begin{bmatrix} \mathbf{v} \\ \omega \end{bmatrix} \in \mathbb{R}^6, \tag{3.45}$$

denote its *generalized position* and *generalized velocity*, respectively. Here, $\mathbf{r} \in \mathbb{R}^3$ is the center-of-mass position, $\mathbf{q} = [q_s, q_x, q_y, q_z]^T \in \mathbb{R}^4$ is a unit quaternion representing the body orientation, and $\mathbf{v}, \omega \in \mathbb{R}^3$ are the linear and angular velocities in world coordinates [Bender et al. 2014].

The time derivative of the generalized position is related to the generalized velocity through a kinematic map $\mathbf{G}(\underline{\mathbf{x}})$:

$$\dot{\underline{\mathbf{x}}} = \mathbf{G}(\underline{\mathbf{x}})\, \underline{\mathbf{v}}, \qquad \mathbf{G}(\underline{\mathbf{x}}) = \begin{bmatrix} \mathbb{1}_{3\times3} & \mathbf{0}_{3\times3} \\ \mathbf{0}_{4\times3} & \mathbf{G}_q(\mathbf{q}) \end{bmatrix} \in \mathbb{R}^{7\times6}, \tag{3.46}$$

where $\mathbb{1}_{3\times3} \in \mathbb{R}^{3\times3}$ is the identity matrix and

$$\mathbf{G}_q(\mathbf{q}) = \frac{1}{2} \begin{bmatrix} -q_x & -q_y & -q_z \\ q_s & q_z & -q_y \\ -q_z & q_s & q_x \\ q_y & -q_x & q_s \end{bmatrix} \in \mathbb{R}^{4\times3}. \tag{3.47}$$

Equivalently, the rotational kinematics are $\dot{\mathbf{q}} = \frac{1}{2}[0, \omega]^T \otimes \mathbf{q}$, where $\otimes$ denotes quaternion multiplication. The resulting quaternion derivative is tangent to the unit sphere, and in practice the quaternion is renormalized after each integration step.

The *generalized mass matrix*, defined in generalized velocity space, has a block-diagonal structure

$$\underline{\mathbf{M}} = \begin{bmatrix} m\mathbb{1}_{3\times3} & \mathbf{0}_{3\times3} \\ \mathbf{0}_{3\times3} & \mathbf{I}_\omega \end{bmatrix} \in \mathbb{R}^{6\times6}, \tag{3.48}$$

where m is the scalar mass and

$$\mathbf{I}_\omega = \mathbf{R}(\mathbf{q})\, \mathbf{I}_{\mathrm{body}}\, \mathbf{R}(\mathbf{q})^T \in \mathbb{R}^{3\times3} \tag{3.49}$$

is the inertia tensor in world coordinates. Here, $\mathbf{R}(\mathbf{q}) \in \mathbb{R}^{3\times3}$ is the rotation matrix corresponding to the quaternion $\mathbf{q}$, and $\mathbf{I}_{\mathrm{body}} \in \mathbb{R}^{3\times3}$ is the constant inertia tensor in the body frame. Consequently, $\mathbf{I}_\omega$ changes with the body orientation.

3.8.1 Rigid Body Dynamics

Macklin et al. [2020] define a variational inertial energy for rigid bodies in terms of generalized velocities as

$$\phi_{\text{RB}} = \frac{1}{2}(\underline{\mathbf{v}} - \underline{\hat{\mathbf{v}}})^T \underline{\mathbf{M}}(\underline{\mathbf{v}} - \underline{\hat{\mathbf{v}}}), \tag{3.50}$$

where $\underline{\hat{\mathbf{v}}} = \underline{\mathbf{v}}^t + \Delta t\, \underline{\mathbf{a}}^t$ is the inertial generalized velocity obtained from the previous state and the generalized acceleration $\underline{\mathbf{a}}^t$, comprising linear and angular acceleration.

Generalized forces derived from a position-space potential ϕ_i are obtained through the kinematic map as

$$\underline{\mathbf{f}}_i = -\mathbf{G}(\underline{\mathbf{x}})^T \frac{\partial \phi_i}{\partial \underline{\mathbf{x}}}. \tag{3.51}$$

Freezing the kinematic map $\mathbf{G}$ and the world-space inertia tensor $\mathbf{I}_\omega$ at the beginning of the time step turns Eq. (3.50) into a potential over the updated generalized position that can be included additively in the incremental objective. Gyroscopic forces are integrated explicitly.

Two notable alternatives relax or refine this formulation. *Affine Body Dynamics* [Lan et al. 2022] represents each body by an affine transformation (12 DoF) with a stiff orthogonality penalty instead of enforcing $SO(3)$ exactly. Romanyà-Serrasolsas et al. [2025] avoid the constant $\mathbf{G}$ approximation by parametrizing rotations with $SO(3)$ matrices and computing derivatives in the Lie algebra tangent space, yielding improved convergence for rotational degrees of freedom.

3.8.2 Constrained Rigid Body Systems

Rigid bodies are often coupled through joints to form articulated systems. Following Chen et al. [2022], such articulation constraints can be incorporated into the same unconstrained optimization framework using the penalty and barrier techniques introduced in Section 2.8.

Simple point-to-point distance connections can be modeled with penalty potentials,

$$\phi_{\text{dist}}(\mathbf{x}) = \frac{k}{2}\left(\|\mathbf{p}_1(\mathbf{x}) - \mathbf{p}_2(\mathbf{x})\| - l\right)^2, \tag{3.52}$$

where $\mathbf{p}_1$ and $\mathbf{p}_2$ are attachment points on two rigid bodies, l is the rest distance, and k is a stiffness parameter.

Inequality constraints, such as directional joint limits, can be modeled with barrier potentials,

$$\phi_{\text{limit}}(\mathbf{x}) = \kappa\, b\big(d_{\max} - \|\mathbf{d}_1(\mathbf{x}) - \mathbf{d}_2(\mathbf{x})\|,\ \hat{d}\big), \tag{3.53}$$

where $\mathbf{d}_1$ and $\mathbf{d}_2$ are body-attached unit directions, $d_{\max}$ is the chord-distance equivalent of the angular limit $\theta_{\max}$, $\hat{d}$ is the distance threshold from Eq. (3.37), and b is the same smoothly clamped log-barrier used for contact.

Such elementary energies provide the building blocks for more complex articulated mechanisms by composition.

3.9 Chapter Summary

This chapter introduced the concrete variational models that populate the optimization-based simulation framework described in Chapter 2. Starting from continuum descriptions, we

traced the path from strain energy densities through kinematic quantities and constitutive models to discrete finite element potentials with well-defined first and second derivatives. We then extended the picture to contact, friction, and Rayleigh damping, showing how non-conservative effects can be incorporated through lagged potentials. Codimensional models (shells and cloth) fit naturally into the same additive structure, combining membrane energies derived from the previously introduced hyperelastic models with discrete bending energies. Finally, constrained rigid body systems were shown to admit variational formulations that unify them with deformables in a single coupled solve.

CHAPTER 4

Differentiable Simulation and Sensitivity Analysis

The optimization-based simulation framework of the previous chapters can be extended to problems where the simulation parameters themselves are unknown. In many inverse design, control, and parameter identification tasks, one wishes not only to simulate a system for fixed parameters, but to *adjust* those parameters so that the simulation output matches a desired objective. This chapter introduces a common technique for computing the required sensitivities: the adjoint method, derived from the Implicit Function Theorem. We focus on variational problems, where the forward simulation is itself posed as the minimization of an energy or incremental potential, but the same principle applies more broadly to nonlinear systems of equations. The presentation here is concise; a more exhaustive treatment can be found in the course notes by Coros et al. [2021].

4.1 Inverse Problem Setting

Let $\theta \in \mathbb{R}^M$ be a vector of design parameters and let $\mathbf{x}(\theta) \in \mathbb{R}^N$ denote the converged simulation state obtained by minimizing an energy or incremental potential $E(\mathbf{x}, \theta)$. We define a scalar loss $\mathcal{L}(\mathbf{x}(\theta), \theta)$ that measures how well the simulated outcome satisfies a desired objective, and seek to solve the bi-level optimization problem

$$\theta^\star = \arg\min_{\theta} \; \mathcal{L}\big(\mathbf{x}(\theta), \theta\big), \qquad \text{subject to} \quad \mathbf{x}(\theta) = \arg\min_{\mathbf{x}} E(\mathbf{x}, \theta). \tag{4.1}$$

This nested structure appears in many simulation-driven inverse problems. In the case study presented in Chapter 8, for example, θ collects nodal strain and bending stiffness values of a facial simulation shell, while $\mathbf{x}$ contains the quasistatic equilibrium positions for a set of target facial expressions. The global potential E models shell mechanics and the loss $\mathcal{L}$ penalizes deviation from the target facial expressions.

To solve for θ with gradient-based optimization, we need the total derivative $\frac{\mathrm{d}\mathcal{L}}{\mathrm{d}\theta}$. A naive approach would differentiate through every iteration of the inner Newton solve, storing or recomputing the full computational graph. While possible in principle with automatic

differentiation, this is often impractical for large-scale simulations due to memory and runtime costs. Instead, we exploit the fact that the converged state is defined implicitly by a stationarity condition.

4.2 Implicit Function Theorem

The inner minimization in Eq. (4.1) implies the first-order stationarity condition

$$\mathbf{g}(\mathbf{x}, \theta) \;:=\; \frac{\partial E}{\partial \mathbf{x}} = \mathbf{0}. \tag{4.2}$$

The stationarity condition defines $\mathbf{x}$ implicitly as a function of θ. The Implicit Function Theorem states that, if $\mathbf{g}$ is continuously differentiable and the Jacobian

$$\frac{\partial \mathbf{g}}{\partial \mathbf{x}} = \frac{\partial^2 E}{\partial \mathbf{x}^2} \tag{4.3}$$

is nonsingular at the solution, then $\mathbf{x}(\theta)$ is locally a C^1 function. Since $\mathbf{g}(\mathbf{x}(\theta), \theta) = \mathbf{0}$ as the parameters vary, its total variation with respect to θ must vanish: any direct change in the residual induced by the parameters is balanced by the corresponding change in the equilibrium state. Differentiating Eq. (4.2) with respect to θ therefore gives

$$\frac{\mathrm{d}\mathbf{g}}{\mathrm{d}\theta} = \frac{\partial \mathbf{g}}{\partial \mathbf{x}} \frac{\partial \mathbf{x}}{\partial \theta} + \frac{\partial \mathbf{g}}{\partial \theta} = \mathbf{0}, \tag{4.4}$$

and therefore

$$\frac{\partial \mathbf{x}}{\partial \theta} = -\left(\frac{\partial^2 E}{\partial \mathbf{x}^2}\right)^{-1} \frac{\partial^2 E}{\partial \mathbf{x}\, \partial \theta}. \tag{4.5}$$

Inserting Eq. (4.5) into the chain rule yields

$$\begin{aligned} \frac{\mathrm{d}\mathcal{L}}{\mathrm{d}\theta} &= \frac{\partial \mathcal{L}}{\partial \theta} + \frac{\partial \mathcal{L}}{\partial \mathbf{x}} \frac{\partial \mathbf{x}}{\partial \theta} \\ &= \frac{\partial \mathcal{L}}{\partial \theta} - \frac{\partial \mathcal{L}}{\partial \mathbf{x}} \left(\frac{\partial^2 E}{\partial \mathbf{x}^2}\right)^{-1} \frac{\partial^2 E}{\partial \mathbf{x}\, \partial \theta}. \end{aligned} \tag{4.6}$$

This expression is useful conceptually, but it should not be evaluated by explicitly computing the inverse matrix or the full sensitivity matrix $\frac{\partial \mathbf{x}}{\partial \theta} \in \mathbb{R}^{N \times M}$. The adjoint method avoids this by replacing the inverse multiplication with one linear solve.

4.3 The Adjoint Method

We introduce the adjoint state $\lambda \in \mathbb{R}^N$ as the solution of

$$\left(\frac{\partial^2 E}{\partial \mathbf{x}^2}\right)^T \lambda = -\left(\frac{\partial \mathcal{L}}{\partial \mathbf{x}}\right)^T. \tag{4.7}$$

Substituting this identity into Eq. (4.6) gives the adjoint gradient

$$\frac{\mathrm{d}\mathcal{L}}{\mathrm{d}\theta} = \frac{\partial \mathcal{L}}{\partial \theta} + \lambda^T \frac{\partial^2 E}{\partial \mathbf{x}\, \partial \theta}. \tag{4.8}$$

Thus, instead of solving for the full sensitivity matrix $\frac{\partial \mathbf{x}}{\partial \theta}$, one solves a single adjoint system with the transpose of the tangent matrix and contracts the result with the mixed derivative. The additional cost in comparison to the forward solve is one extra linear solve, the contraction of the mixed derivative.

Lagrange multipliers perspective. The same equations can also be derived from a Lagrange multiplier viewpoint. Treating the stationarity condition as an equality constraint, one introduces the augmented loss

$$\mathcal{L}_{\text{aug}}(\mathbf{x}, \theta, \lambda) = \mathcal{L}(\mathbf{x}, \theta) + \lambda^T \mathbf{g}(\mathbf{x}, \theta). \tag{4.9}$$

At feasible states, $\mathbf{g} = \mathbf{0}$, the augmented loss equals the original loss. Choosing λ such that $\frac{\partial \mathcal{L}_{\text{aug}}}{\partial \mathbf{x}} = \mathbf{0}$ recovers Eq. (4.7), and differentiating $\mathcal{L}_{\text{aug}}$ with respect to θ recovers Eq. (4.8).

Multiple solves. The derivation above assumes a single equilibrium solve, but the same idea applies directly to several forward solves sharing the same parameters. In Chapter 8, for example, the forward problem consists of multiple independent quasistatic equilibria, one for each target facial expression, all depending on the same stiffness parameters θ. Each equilibrium yields its own adjoint state, and the total gradient is obtained by summing the contributions of Eq. (4.8) over all solves.

Dynamics. For time-dependent simulations, the unknown state is a sequence of time-step solutions rather than a single equilibrium. The same implicit differentiation principle applies, but the adjoint states are propagated backward through the time-stepping sequence, accumulating sensitivities from later states to earlier ones. A detailed presentation of such a scheme can be found in the course notes by Coros et al. [2021]. Note that sensitivity analysis of time-dependent problems is not considered in this thesis.

Differentiation frameworks. The adjoint method reduces the sensitivity computation to a small set of derivative objects: $\frac{\partial^2 E}{\partial \mathbf{x}^2}$, $\frac{\partial^2 E}{\partial \mathbf{x} \partial \theta}$, $\frac{\partial \mathcal{L}}{\partial \mathbf{x}}$, and $\frac{\partial \mathcal{L}}{\partial \theta}$. Some of these derivatives are determined by the physical model, while others change whenever the objective, design variables, or parameterization change. This makes inverse simulation difficult to implement robustly by hand: every new energy, loss term, or design parameter may require new derivatives and new assembly code. Differentiation frameworks such as SymX, discussed in Chapter 5, lower this barrier by dynamic code generation of derivatives and automatic assembly.

4.4 Optimization in Parameter Space

With the gradient $\mathrm{d}\mathcal{L}/\mathrm{d}\theta$ available, the outer minimization (4.1) can be approached with standard first-order optimization methods. A key aspect is that each trial parameter vector requires solving a new forward simulation. Thus, the cost of the outer optimization is dominated not by the parameter update itself, but by the number and difficulty of the forward solves it triggers.

For this reason, inverse simulation problems are typically solved with conservative, safeguarded variants of first-order methods. Plain gradient descent is simple and robust, but can require many iterations on ill-conditioned parameter spaces. Adam [Kingma and Ba 2015] can be useful when gradients are noisy or when different parameters have very different scales.

When gradients are expensive but reasonably reliable, L-BFGS [Nocedal and Wright 2006] is often preferable, since it uses gradient differences to construct a better search direction and usually reduces the number of outer iterations.

A second important consideration is the parameterization itself. Physical parameters such as stiffness may span several orders of magnitude and must remain positive. It is therefore often preferable to optimize them in a reparametrized space such as log-exp,

$$\theta = \exp(\bar{\theta}). \tag{4.10}$$

Safeguarded steps. The search direction proposed by the outer optimizer is only a local model of improvement. Large parameter updates can make the forward problem ill-conditioned, violate admissibility constraints, or cause Newton's method to struggle or outright fail. As in the forward Newton solve, the outer optimization therefore benefits from a step cap and a backtracking line search.

Given a proposed direction $\Delta\theta$, a step cap first limits its maximum componentwise or relative change. A backtracking line search then tries

$$\theta_{\text{trial}} = \theta + \alpha\Delta\theta, \tag{4.11}$$

with decreasing values of α until the trial point satisfies the practical requirements of the inverse problem. At minimum, the trial parameters should be admissible, the forward simulation should converge, and the resulting loss should be finite. When the direction is expected to be a descent direction, an Armijo condition can additionally enforce sufficient decrease,

$$\mathcal{L}(\theta_{\text{trial}}) \leq \mathcal{L}(\theta) + c\,\alpha\,\frac{\mathrm{d}\mathcal{L}}{\mathrm{d}\theta}\Delta\theta. \tag{4.12}$$

Rejected trials are discarded and the forward simulation state is restored from the last accepted iterate.

Part III

Scientific Contributions

CHAPTER 5

SymX: Simulation from Symbolic Expressions

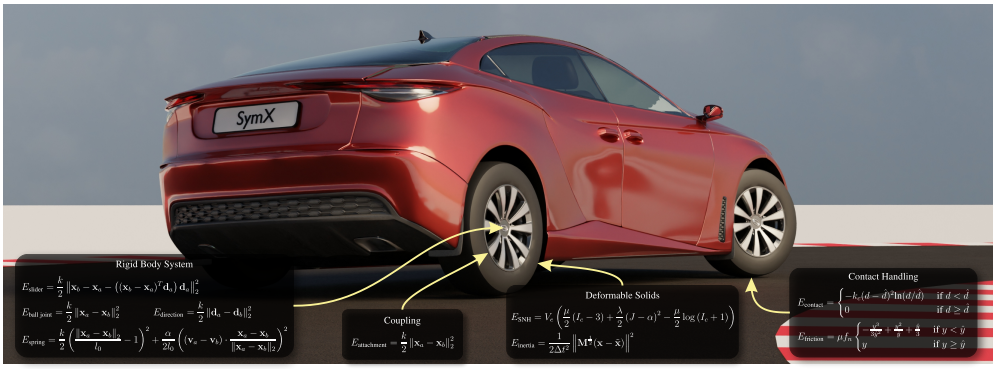


Figure 5.1: *A car drifts through a hairpin corner with strong rigid/deformable coupling. The simulation combines non-linear material, constraint-based rigid body dynamics, attachments, and frictional contact. All components are expressed symbolically in SymX, which automatically carries out the entire solve.*

This chapter presents SymX, the main differentiation and assembly infrastructure contribution of this thesis. It is adapted from the original SymX publication [Fernández-Fernández et al. 2025b], and its implementation is available as open-source software[1].

Chapter 2 introduced the optimization-based formulation used throughout this thesis: physical models are written as local energy contributions, assembled into a global objective, and solved with Newton-type methods using first- and second-order derivatives. SymX uses this formulation as a programming model for representing complex simulation problems symbolically, which can then be turned into executable programs automatically.

Subsequent chapters use or build on the infrastructure described here. STARK (Chapter 6) uses SymX as the assembly backend for a strongly coupled simulator; PPN (Chapter 7) refines

[1] The SymX source code is available at `https://github.com/InteractiveComputerGraphics/symx`.

the Newton solve once assembled Hessians are available; and IFA (Chapter 8) extends the pipeline to inverse and differentiable simulation through adjoint methods. Since the original publication, SymX has also been extended with adjoint-method support, which is used later in the IFA chapter.

5.1 Introduction

In the research area of physically-based simulation a common problem is to efficiently compute the solution of non-linear equations, e.g., to simulate non-linear materials [Smith et al. 2018], to handle collisions with friction [Li et al. 2020], or to resolve non-linear constraints [Bender et al. 2014]. This problem is also highly relevant for simulation methods based on energy minimization which have become increasingly popular [Gast et al. 2015; Narain et al. 2016; Brown et al. 2018; Chen et al. 2022]. Such methods allow the user to combine different material models and constraints in a single simulation by formulating (typically non-linear) potential energy functions for each component. Implicit time integration is often performed by minimizing the sum of the inertia and all other energy potentials, e.g., using Newton's method. As described in Foundations Chapter 2, optimization-based solvers provide important robustness qualities such as the use of line search to ensure progress during the solve. While first-order methods, such as Projective Dynamics [Bouaziz et al. 2014], may be used to solve this class of problem, we focus exclusively on global second-order methods due to their strong convergence [Li et al. 2020].

In this context, first- and second-order derivatives of many, and possibly very complex, energy expressions are required. Simulations with multiple interacting physical systems, such as rigid and deformable bodies, might require tens of different energies when considering, not only the internal mechanical effects, boundary conditions and joints, but also contacts and friction between their discretization primitives, e.g. triangles, edges and vertices. Developing, testing and maintaining efficient simulation code to evaluate these energy expressions and their derivatives, and to assemble the results into global data structures is a laborious and error-prone endeavour.

Available tools which try to solve some of these problems assist e.g., by computing the required derivatives using automatic differentiation or the assembly process. However, there was no tool capable of automating the whole pipeline in an effective manner. In this chapter, we propose an integrated solution to differentiation and assembly in the context of physically-based simulation with the following goals:

- **Automation:** First- and second-order derivatives should be computed and assembled completely automatically.
- **Performance:** The evaluation of the energy expression and its derivatives must be fast in order to make the system relevant beyond very early prototyping or small simulations.
- **Productivity:** It should be easy to add or to change expressions and recompilation times should be short to avoid user idling.
- **Flexibility:** The system should work with user-defined data structures, notably the performance-critical sparse matrix for the global Hessian, while imposing no limitations on the choice of

minimization method or linear system solver. Additionally, it should support dynamic problem topologies and enable the processing of individual element contributions, for example to allow for projection to positive semi-definiteness.

- **Accessibility:** The system should be accessible and uncomplicated to set up, build and distribute in order to further facilitate the exchange of ideas between researchers and the replicability of other's work.

In this chapter we show in a detailed analysis that the existing tools at the time the research took place fail to fulfill at least one of these requirements and present our open source framework, SymX, which addresses these points. By drastically reducing the time spent on differentiation and tedious or repetitive implementation tasks, our proposed system enables researchers to explore ideas very efficiently and to easily compare between different concepts, virtually eliminating iteration delays. While our system can be used as a prototyping tool, it provides enough performance as-is to run relatively large scenes with complex state-of-the-art models as we show in Figure 5.1.

5.2 Related Work

Here, we first review works related to the optimization-based simulation setting introduced in Foundations Chapter 2. Then, we give an overview of the broad landscape of automated approaches to compute derivatives and other systems that make possible to express complex problems in terms of succinct expressions or programs. We refer to Section 5.3.2 for an in-depth discussion about the feasibility of applying specific methods and tools listed herein to our application.

5.2.1 Optimization Time Integrators

Incremental potential formulations [Ortiz and Stainier 1999] are a common approach for dynamic simulation in computational mechanics and computer animation. By formulating each time step as the minimization of a scalar objective rather than as the solution of a nonlinear system of equations, these methods often lead to robust and efficient implementations [Kharevych et al. 2006; Gast et al. 2015]. See Foundations Chapter 2 for further works in this domain. Their practical performance, however, depends strongly on the availability of first- and second-order derivatives of the global objective.

This requirement becomes especially important as the simulated models grow in complexity. Barrier contact potentials [Li et al. 2020], for example, provide a powerful mechanism for robust contact handling, but integrating them into existing multiphysics systems [Holz et al. 2025] can require substantial implementation effort. This creates a need for tools that allow new energy terms, constraints, and contact models to be expressed compactly while still providing the derivative information required by Newton-type solvers.

5.2.2 Differentiation

Automating the task of differentiation via computer programs has a long history, the dissertation of John F. Nolan [1953] being one of the original works in the field. Over the decades that followed, the relevancy of this field has seen huge leaps forward, and it is at the core of

today's most advanced technologies in important fields such as artificial intelligence. Since a full review is outside the scope of this chapter, we point the interested reader to the book by Griewank and Walther [2008].

There are different strategies to differentiation. *Automatic differentiation* (AD) is perhaps the most widely used one due to its capabilities to handle derivatives of complex computer programs with dynamic control flow. In AD, a computation graph of the program to differentiate is built and derivative information is propagated along with the original computation. At a very high level, AD techniques can be divided in two main categories, *backward* and *forward* mode. The former one is more efficient when the program has a large number of degrees of freedom, while the latter can be more efficient otherwise. In recent years, the increased interest in machine learning has brought a lot of attention to backward AD techniques and very powerful tools, such as TensorFlow [Abadi et al. 2016] or PyTorch [Paszke et al. 2017], have been widely adopted. In our setting, however, due the structure of the problem, we need to compute derivatives of local functions which depend on a relatively low number of degrees of freedom, therefore forward mode is usually preferred. We refer the reader to the work by Schmidt et al. [2022] which presents an in-depth discussion on the efficiency of forward and backward AD for such problems. The authors also provide an implementation, TinyAD, that is shown to outperform state-of-the-art tools in their applications. Aside of AD solutions to specific problems, there are general purpose AD tools that can be used to conveniently obtain derivatives in a more general context, e.g. CasADi [Andersson et al. 2019], albeit at the cost of being unable to fully exploit the structure of the problem at hand. We also point to Enzyme [Moses and Churavy 2020], an LLVM-based AD compiler plugin, and Tapenade [Hascoet and Pascual 2013], a source-to-source AD tool, as further examples of software that can generate efficient derivatives from code.

On the other hand, *symbolic differentiation* can be used to generate derivatives from input mathematical expressions. Dynamic loops and branching are usually more restricted in comparison to AD solutions, but the upside is that there is potentially more room for static analysis and optimization of the expressions, assuming that the target function can be described in closed form. Symbolic differentiation used as an external tool to the main application (e.g., using SymPy [Meurer et al. 2017], Mathematica [Wolfram Research 2026] or Maple [Maplesoft 2026]) is a well-known option. This approach has seen some criticism [Schroeder 2019] related to performance and the error-prone, often manual, process to integrate the generated code into the simulator. However, efficiency concerns can be addressed by using Common Sub-expression Elimination (CSE) on the resulting derivative expressions, which can be carried out directly in the aforementioned tools. Recently, the work by Herholz et al. [2022] has proven that integrating symbolic differentiation in the application code, coupled with CSE and on-demand compilation can solve the performance shortcomings while making the process completely autonomous.

5.2.3 Simulation Systems and DSLs

In the context of simulation, Domain Specific Languages (DSLs) aim to simplify description and solution of specific problem classes or systems that process or encompass an entire program. Liszt [DeVito et al. 2011] is a DSL designed to develop mesh-based PDE solvers that allows to define data at discretization nodes, batching subsequent operations for efficient processing. Simit [Kjolstad et al. 2016] and Ebb [Bernstein et al. 2016] are DSLs designed to ease writing high performance simulations by splitting the problem definition between data

structures and simulation code and automatically generating routines taking care of sparse matrix assembly both on the CPU and on the GPU. More recently, Taichi [Hu et al. 2019] and MeshTaichi [Yu et al. 2022] take this further by allowing internal data structures to be changed, allowing the user to easily find which is the most suitable for their application. DeVito et al. [2017] proposed a DSL to solve non-linear least squares problems with first-order methods from a concise objective function definition using symbolic differentiation at intermediate representation level. Further, Thallo [Mara et al. 2021] presents performance improvements by allowing computation and storage reorganization of the code.

Outside of DSLs, SANM [Jia 2021] is a solver that applies the Asymptotic Numerical Method fully automatically to problems defined symbolically. ACORNS [Desai et al. 2022] generates first- and second-order derivatives of target functions defined in the main application codebase at build time. Herholz et al. [2022] propose a code generator to transform symbolically defined sparse operations into compiled high performance applications that avoid expensive sparse data structure bottlenecks. Similarly, Dr.Jit [Jakob et al. 2022] compiles per-scene kernels to accelerate execution times in the context of physically-based differentiable rendering. SymX follows the general philosophy of splitting core definitions from the internal procedures and data structures. To the best of our knowledge, none of the above methods fulfill all the requirements established in Section 5.1: some are too specialized for other applications or require a particular type of solver, and others are not flexible enough in terms of discretization and sparsity or are generally not efficient enough, especially when considering second-order derivatives.

Another class of automated systems are PDE solvers which facilitate the process of using the Finite Element Method (FEM) to solve problems defined in the continuum. Popular examples are FEniCS [Alnæs et al. 2015], Firedrake [Rathgeber et al. 2016], Freefem [Hecht 2012] or Moose [Lindsay et al. 2022]. These type of frameworks usually offer a wide range of capabilities such as the use of different finite element spaces, meshing, choice of solver, distributed computing and more. However, most problems found in computer graphics are (at least partially) discrete in nature (e.g., rigid body dynamics, contacts or friction), rendering this class of frameworks unfit for our task.

5.3 Problem Definition

As introduced in Foundations Chapter 2, the common structure of the problems we are interested in is an objective of the form

$$E(\mathbf{x}) = \sum_{p \in \mathcal{P}} \sum_{e \in \mathcal{T}_p} \phi_p^e(\mathbf{x}^e),$$

where $\mathcal{P}$ is the set of different physical effects, $\mathcal{T}_p$ is the set of discrete contributions associated with effect p, and ϕ_p^e is the corresponding local potential. A forward time step, quasistatic solve, or inner inverse-simulation problem then seeks

$$\mathbf{x}^\star = \arg\min_{\mathbf{x}} E(\mathbf{x}).$$

One can exploit the structure of this problem by implementing each energy potential definition $\phi_p^e(\mathbf{x}^e)$, along with its gradient and Hessian. Only the gathered local values and parameters change from one element, constraint, or contact pair to another. These are then

scattered into (typically concurrent) global data structures for energy, gradient, and Hessian and passed along to the optimizer algorithm.

5.3.1 Example: Deformable Solids

As a concrete example, consider a finite element discretization of a deformable solid with a compressible Neo-Hookean material, following the notation of Section 3.3. For an element e, the local elastic potential is obtained by integrating a strain energy density over the reference element,

$$\phi^e_{\mathrm{NH}}(\mathbf{x}^e) = \int_{\Omega^e_0} \psi_{\mathrm{NH}}(\mathbf{F}^e(\mathbf{X}; \mathbf{x}^e)) \, \mathrm{d}V, \qquad \mathbf{x}^e = \mathbf{P}^e \mathbf{x}. \tag{5.1}$$

Here $\mathbf{F}^e$ is the deformation gradient induced by the element interpolation. For the standard compressible Neo-Hookean model,

$$\psi_{\mathrm{NH}}(\mathbf{F}) = \frac{\mu}{2}(I_1 - 3) - \mu \ln J + \frac{\lambda}{2}(\ln J)^2, \qquad I_1 = \mathrm{tr}(\mathbf{F}^T\mathbf{F}), \qquad J = \det(\mathbf{F}), \tag{5.2}$$

with Lamé parameters μ and λ.

This relatively simple and widely used constitutive model already showcases how expressions can grow complex and preset implementation challenges. The deformation gradient depends on the element shape functions; the deformation depends on nonlinear invariants, a determinant, and a logarithm. Further, and a Newton-type solver requires second-order derivatives of the fully composed local potential.

5.3.2 Existing approaches

Manual implementation

The baseline option is to differentiate the energies by hand and to manually implement and optimize the corresponding evaluation and assembly. Naturally, in the context of commonly used potentials this can be relatively straightforward since the derivatives might be known, but this is not always the case in research. Thorough manual code optimization can yield very high performance results and the approach is flexible, however, manual implementations are typically very time-consuming and error-prone to develop, test and maintain. Moreover, changes or additions to the existing energies are slow, impeding fast prototyping of new solutions.

Automatic differentiation

AD is often the solution of choice for many applications due to its flexibility, ease of integration and large capabilities for automation, which is why it is commonly used for prototyping and testing. AD excels at differentiating complex programs with arbitrary control flow that depend on a large number of variables. Our problem, however, has a very specific structure, as outlined in the previous section, which should be leveraged for efficient generation and evaluation of derivatives. General differentiation frameworks, e.g. CasADi [Andersson et al. 2019], cannot take advantage of the structure of our specific problem, necessitating a formulation of the global energy as an explicit sum of all element contributions. Additionally, the limitations of general-purpose AD tools are further exacerbated when the topology of the problem changes,

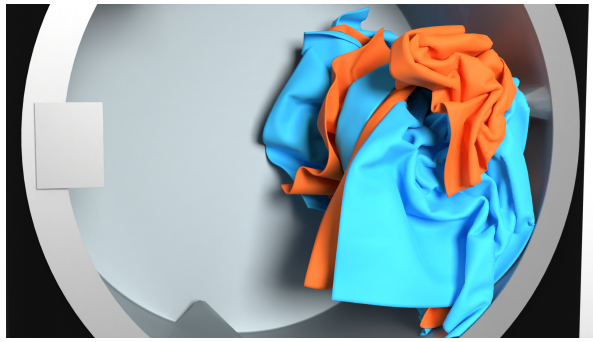


Figure 5.2: *Close-up of the tumble dryer simulation introduced in Foundations Section 2.7. This scene features a total of 46 distinct energies (138 auto generated functions), including rigid body dynamics and constraints, shell mechanics and contact and friction potentials for all the combinations between all discretization primitive pairs. The simulation features 245k degrees of freedom.*

for example due to dynamic contacts or remeshing. The lack of structural awareness of the problem also inhibits the possibility to perform per-element operations, such as projecting element Hessians positive definite.

After evaluating various tools, we have determined that TinyAD [Schmidt et al. 2022] is the best AD candidate for our problem as it is specifically designed to compute the same type of derivatives found in our applications, supports per-element projections to positive semi-definiteness, and, as the authors show in their original paper, it outperforms established AD libraries for such problems. To assess if TinyAD meets our performance requirements, we conduct comprehensive performance comparisons in Section 5.6.

Symbolic off-the-shelf tools

Symbolic mathematical engines such as Mathematica [Wolfram Research 2026], Maple [Maplesoft 2026] or SymPy [Meurer et al. 2017] can be used to compute derivatives and generate corresponding code externally. However, our proposed framework not only performs the differentiation and code generation, but it is also aware of the simulation data structures and as such is able to take care of the evaluation of these functions as well as the assembly of the global gradient and Hessian. This removes the weak link of coupling two codebases: one solver and one external (potentially licensed) repository of energy definitions.

To highlight why this is desirable, consider the tumble dryer simulation shown in Figure 5.2. This simulation requires 46 distinct energy types to model deformable materials, rigid bodies, joints and constraints, as well as contact and friction between all discretization primitives. SymX not only generates and compiles the three required functions per expression to compute the energy, gradient and Hessian (a total of 138 functions), but it is also able to autonomously evaluate them using user-defined accessible data arrays to assemble the global data structures.

Another problem is that some general purpose tools are not built with performance as a priority, e.g. SymPy is written in Python, and relying on them for the derivatives can drastically slow down the entire pipeline. See Section 5.6 for SymPy differentiation benchmarks.

Simulation systems and DSLs

While there is a plethora of relevant systems and Domain Specific Languages (DSL) as outlined in Section 5.2, we did not find a solution that fulfills all of our requirements.

The most relevant approaches in the context of computer graphics that support second-order derivatives are ACORNS [Desai et al. 2022] and the method by Herholz et al. [2022]. ACORNS can generate Hessians that must be then manually integrated in the simulation but it does not support functions by parts (branching) and is outperformed by integrated solutions such as TinyAD [Schmidt et al. 2022]. The method by [Herholz et al. 2022], on the other hand, presents in fact good performance by reducing and compiling all the sparse queries into a single program, but this prohibits changes in the sparsity pattern, necessary for contacts and remeshing.

Considering solutions with no support for differentiation, we find that Simit [Kjolstad et al. 2016] and Ebb [Bernstein et al. 2016] are effective simulation systems that offer convenience and performance. Assuming that the derivatives of all the potentials needed in the simulation are known, these systems offer scripting languages that allow the user to conveniently describe such potentials, as well as other components of the simulation, such as time-stepping schemes, minimizers and linear solvers. While the lack of differentiation capabilities makes them incompatible with our goals, we validate the performance of SymX in a comparison with Simit in Section 5.6.3.

5.4 SymX Framework

SymX is an integrated solution that seeks to fulfill all the requirements defined in Section 5.1 while avoiding the shortcomings of the existing methodologies. Note that the system's concern is to provide global assembled derivatives, therefore it does not impose any requirements to the simulation software and it is independent of the rest of its components, e.g. minimization method, time discretization, collision detection, etc. We give a general overview of the framework in Section 5.4.1. The symbolic engine is introduced in Section 5.4.2. Finally, we discuss the required matrix assembly in Section 5.4.3.

5.4.1 Overview

At a high level, the input to SymX is a collection of symbolic expressions with symbols associated with user-owned data arrays, and it returns the global gradient vector and the global Hessian sparse matrix for the specified sets of degrees of freedom. Figure 5.3 shows how the minimization problem of the deformable solids example in Section 5.3.1 is solved using our framework. For the simulation model on the left, the user has to implement the mathematical expressions (center). Then our framework generates an expression graph and determines the derivatives using symbolic differentiation. For efficient evaluation, SymX generates source code for each energy, compiles and caches it before the simulation starts. During the simulation, at the user's request, the compiled functions are evaluated for each energy and for each element and the global gradient and Hessian are assembled. Evaluation and assembly are automatically parallelized across elements.

We now analyze the code example in Figure 5.4, which defines the energies shown in Figure 5.3. First the array of degrees of freedom are declared (line 10) in order to identify the symbols the system will take derivatives with respect to. Lines 11 and 23 define the energy

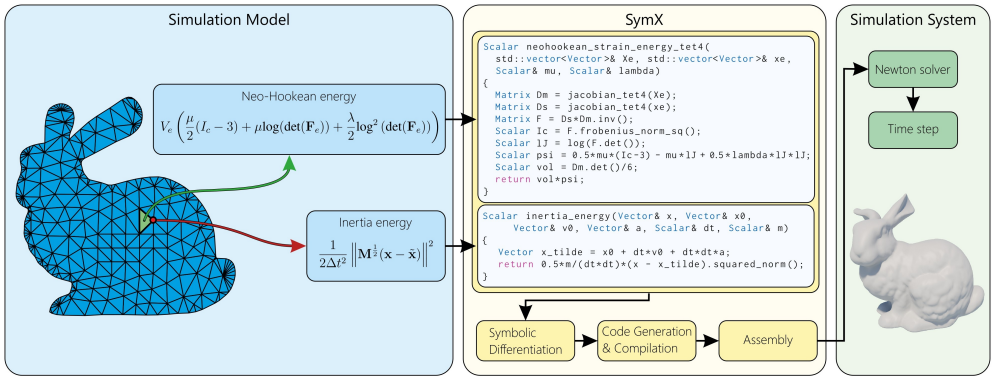


Figure 5.3: *Overview of a simulation step with the SymX framework. Left: The input is a discretized model and energy functions. In our application example we use a tet mesh and the inertia and Neo-Hookean strain energy functions. Center: The user has to implement a symbolic definition of these functions. The framework will then compute the element gradients and Hessians by symbolic differentiation, generate and compile efficient code, and assemble the element contributions to get the global gradient and Hessians. Finally, these terms can be used in a Newton solver to perform a simulation step for the deformable bunny.*

functions for the tets (strain) and nodes (inertia), respectively. Within the body of each energy definition, symbols are created as counterparts of the arrays they represent (lines 15-18 and 27-32) and the symbolic expression to evaluate the energy for an element is set (lines 21 and 35). In line 39, SymX is instructed to compile all the required functions, including derivatives, and to write the shared objects (`.dll` or `.so`) in a specific folder. There are three global evaluation functions available: one to compute the global energy (typically used during line search), another to compute the global energy and gradient (to check for Newton's method convergence) and another, to compute the global energy, gradient and Hessian (to assemble the linearized system of equations in Newton's method). After the initial definitions and compilation, these evaluation functions can be used as many times as necessary. Every global assembly will be executed using updated user data from the mapped arrays, even when they change in size, for example, in the case of mesh refinement.

5.4.2 Symbolic Engine

The core of our framework is the symbolic engine. As input, the engine requires a symbolic mathematical expression for each energy, which is done using SymX's symbolic types `Scalar`, `Vector` and `Matrix`. The framework provides operator overloading and common linear algebra functionalities for these types. Both, the symbolic `Vector` and `Matrix`, are dynamically allocated arrays of scalar expressions.

Instead of directly executing the operation of a symbolic expression, our engine internally generates an expression graph of scalar expressions. In this graph, each node represents either a user-defined symbol, a constant value or an operation applied to the result of its child nodes. SymX supports arithmetic and trigonometric operations as well as square roots and logarithms, and new operations can be added. It is also possible to add custom scalar derivative rules. However, vector and matrix differentiation rules are not yet supported. Conditional branching

```
// Simulation data (uninitialized for brevity)
std::vector<std::array<double, 3>> x, x0, v0, a, x_rest;
std::vector<double> lumped_mass;
double time_step, mu, lambda;
std::vector<std::array<int, 4>> tets;
std::vector<int> nodes;

// Create global energy and define contributions
GlobalEnergy G;
DoF dof = G.add_dof_array(x);
G.add_energy("neo_hookean_tet4", tets,
   [&](Energy& E, Element& tet)
   {
      // Create local symbols from the data arrays
      std::vector<Vector> xe = E.make_dof_vectors(dof, x, tet);
      std::vector<Vector> Xe = E.make_vectors(x_rest, tet);
      Scalar m = E.make_scalar(mu);
      Scalar l = E.make_scalar(lambda);

      // Define energy
      E.set(neohookean_strain_energy_tet4(Xe, xe, m, l));
   });
G.add_energy("inertia", nodes,
   [&](Energy& E, Element& node)
   {
      // Create local symbols from the data arrays
      Vector xn = E.make_dof_vector(dof, x, node);
      Vector x0n = E.make_vector(x0, node);
      Vector v0n = E.make_vector(v0, node);
      Vector an = E.make_vector(a, node);
      Scalar mn = E.make_scalar(lumped_mass, node);
      Scalar dt = E.make_scalar(time_step);

      // Define energy
      E.set(inertia_energy(xn, x0n, v0n, an, dt, mn));
   });

// Compilation
G.compile("path/to/codegen/directory");

// Assemble global data structures
Assembled assembled = G.evaluate_E_grad_hess();
```

Figure 5.4: *SymX code to define, compile and evaluate the inertia and strain energies as well as their gradients and Hessians for the example problem defined in Figure 5.3.*

is a special type of operation that is discussed in Section 5.4.2.

In the following we describe the components of our symbolic engine using the example presented in Section 5.3.1.

Common subexpression elimination

A naively constructed expression graph of all scalar operations for common energies typically contains many reoccurring identical subgraphs. For example, consider a single entry of the deformation gradient of a Lagrangian finite element. The corresponding symbolic expression becomes quite complex for higher-order elements and occurs multiple times in typical strain energy densities. We identify and eliminate structurally identical sub-graphs in order to reduce the amount of generated code and improve performance. To achieve this, we use a hash map local to each energy that stores all expression nodes that were already created in the graph together with an identifier. Whenever a new expression is added, we perform a lookup and replace it with an existing identifier if an identical expression was previously constructed. This deduplication is performed in a bottom-up way when expressions are constructed, therefore it

is sufficient for the lookup of an expression to only compare the expression type and identifiers of its direct children expressions to guarantee uniqueness. In our example, the expression complexity of the Neo-Hookean potential Hessian defined on a linear tetrahedral element (see Figure 5.3) was reduced by 70%, from 7517 to 2284 operations.

To also eliminate algebraically equivalent expressions, Herholz et al. [2022] proposed "algebraic hashing" which assigns hashes to elementary nodes (e.g. variables and literal numbers) and computes hashes of more complex expressions by applying their corresponding operations such as multiplications and additions to the hashes of their subexpressions. However, while improbable, it is possible that the operations performed on the hashes lead to hash collisions of non-equivalent expressions. Therefore, we instead opt for the more robust approach of checking for structural identity.

Symbolic Differentiation

To compute the derivatives of a symbolic expression, our framework recursively traverses its expression graph and applies the chain rule with table lookups for the derivatives of elementary functions. During this traversal, we use a different hash map to cache derivatives of subexpressions that were already computed, making use of the previously performed common subexpression elimination. The same cache is used across the entries of an element gradient and Hessian as their expression trees often have significant overlaps which significantly reduces the time required for differentiation. In the case of the Neo-Hookean energy potential defined on a linear tetrahedron element, differentiation times improve from 1.88 ms to 0.44 ms thanks to caching intermediate derivatives.

Code Generation and Compilation

Once we have the symbolic expressions for a function and its derivatives, we need to evaluate them for all the elements. However, evaluating the expressions by dynamically traversing the expression graph would be prohibitively slow, instead, equivalent C++ code for such functions is generated and compiled, which can be hundreds of times faster. To this end, the expression graph is traversed bottom-up, collecting all operations in the order they need to be calculated, emitting one line of C++ code per operation or graph node. Every generated function has two arguments: a pointer to an input buffer, with the data corresponding to the input symbols (e.g. element vertices, material parameters, etc.), and a pointer to an output buffer (e.g. energy value, gradient and Hessian). Figure 5.5 partially shows the generated function to evaluate the energy value, gradient and Hessian of the Neo-Hookean energy defined in Figure 5.4. It has 26 inputs (12 for `xe`, 12 for `Xe`, 1 for `mu` and 1 for `lambda`), and 157 outputs, (1 for the energy value, 12 for the gradient and 144 for the Hessian). SymX manages the buffers and the mappings between the symbols and the inputs and outputs of the generated functions. Besides the function itself, each generated C++ function file contains metadata with a signature hash id, and the number of inputs and outputs.

Data-Symbol Mapping

A central concept in SymX is the automated handling of the mapping between user's simulation data and the corresponding symbols, which is a crucial benefit of SymX in comparison to external codegen solutions. To achieve this, every symbol in an energy definition is associated with a C++ lambda function that returns an updated view to the data array it represents.

```
void neohookean_tet4_hess(double* in, double* out)
{
  /*
     Add:     450
     Sub:     310
     Mul:     1347
     Inv:     3
     PowN:    11
     Log:     1
     Total:   2122
  */
  double v49 = in[46] * 0.5;
  double v51 = in[24] * in[45];
  double v52 = in[12] + v51;
  ...
  out[144] = v1898;
  out[145] = v931;
  out[146] = v1088;
}
```

Figure 5.5: *Generated C++ function to evaluate the energy value, gradient and Hessian of the Neo-Hookean energy for a tet4 element from the definition in Figure 5.4 lines 11 to 22. Only some representative lines from the top and the bottom of the function are shown.*

This mapping works for both symbol data (e.g., positions, velocities, . . .) and connectivity information (e.g., tetrahedra, triangles, edges, . . .). Before evaluation, SymX requests updated views of all data arrays. The indices represented in each element are used to index the data arrays and to calculate the global indices where the local gradient and Hessian must be assembled at. The only requirement for an array to be compatible with our system is that it must hold its data contiguously in memory so that it can be accessed by beginning, stride and index.

SymX also exposes a low-level API to the symbolically generated and compiled functions for cases where manual evaluation of such functions is preferred. Through this interface, compiled functions can be called directly by the user specifying the data corresponding to each symbol in the expression, bypassing the need for the data-symbol mapping. It is important to note that while this option grants more control, it reintroduces significant complexities, optimization concerns, and safety responsibilities that SymX is designed to handle automatically. For the remainder of the chapter it is assumed that the high-level data-symbol mapping is used and that evaluations and assembly are automated.

Dynamic topology

Contact and adaptive mesh refinement are two factors that can lead to changes in a problem's topology. For instance, points that were previously apart can briefly come into contact and then separate again. Also, new elements may be introduced in regions undergoing large deformations.

SymX supports dynamic topology in a straightforward, general manner: it evaluates and assembles the derivatives for the elements present in the connectivity arrays at the moment of each evaluation request. Hence, updating the objects' tesselation or the list of active contact pairs before evaluating derivatives is all that is needed to correctly handling changing problem topologies.

It is worth noting that SymX itself does not explicitly define concepts such as collision, contact, or even mesh. Symbolic expressions are associated with data arrays, regardless of

whether they represent contact pairs, FEM elements, rigid bodies, or something else entirely.

Branching

SymX supports differentiation and code generation of expressions with arbitrary nested branching using

```
Scalar res = branch(Scalar& c, Scalar& a, Scalar& b);
```

where c is an expression that represents a conditional variable, the expression a is used if $c \geq 0$, and b is used otherwise. `branch` emits an actual `if-else` statement in the C++ generated code and therefore only the correct branch is executed. Differentiation does not affect the branching points since the condition stays the same, the only difference is that code to evaluate the derivatives appear on each side of the branch.

Branching is extensively used in simulation, for example, in the implementation of the mollifiers needed for the IPC edge-edge contacts and the friction potentials [Li et al. 2020]. Additionally, thanks to branching we can implement expressions with functions like `min`, `max`, `abs`.

Conditional evaluations

All the branches spawned by `branch` will generate results that will be assembled. However, a special case of branching is when the value of one branch is zero and therefore derivatives and assembly should be skipped, e.g., when modeling contact barrier potentials. In this case, the energy definition is declared along with an activation condition:

$$\phi_{\text{cond}} = \sum_{e \in \mathcal{T}} \phi_e, \quad \phi_e = \begin{cases} \phi_e^+ & \text{if } c_e > 0 \\ 0 & \text{if } c_e \leq 0, \end{cases} \tag{5.3}$$

where ϕ_e^+ is the local potential of element e and c_e is the activation function. To handle such expressions efficiently, SymX compiles the activation function separately and uses it to gather only the active elements for the evaluation. In this way we can avoid the evaluation and assembly of zero energy contributions.

Fixed value summations

While most potentials are given by a single expression, in numerical simulation it is common to have an inner loop per element over a set of constant data. Such is the case in some FEM simulations where the energy density function is evaluated at multiple integration points. In general, we can formulate this particular case abstractly as

$$\phi_{\text{int}} = \sum_{e \in \mathcal{T}} \sum_{k} \tilde{\phi}(\mathbf{x}_e, \mathbf{p}_k) \tag{5.4}$$

for some energy density function $\tilde{\phi}$, where $\mathbf{p}_k$ are the parameters specific to the inner iteration k.

While it is possible to handle such energies by adding each iteration of the loop to the expression graph, this approach becomes expensive for complex integration rules, even for

moderate iteration counts. To solve this problem, SymX compiles a single function for a symbolic set of inner iteration parameters and calls the function multiple times with updated inputs. This approach scales well to complex models and discretizations making SymX well-suited for high-order FEM simulations as we demonstrate in Section 5.5.2.

Caching compiled functions

Symbolic differentiation, code generation and compilation typically takes less than a couple of seconds for most common expressions, as we later show in Section 5.6.4. However, some expressions such as high-order FEM elements can take significantly longer. To avoid unnecessary work before running simulations, SymX only differentiates and compiles new expressions or modified ones. Compiled functions which correspond to expressions which have not changed are directly loaded. This is achieved by storing a SHA256 hash for each energy, generated from string representations of all the expressions in the graph, and storing it in the compiled objects.

Projection to positive semi-definiteness

As established in Foundations Section 2.5, per-element Hessian projection to SPD is required to ensure descent directions. This is handled in our system by eigendecomposition and negative eigenvalue filtering before assembly. Chapter 7 below discusses advanced Hessian stabilization techniques and presents an adaptive strategy that achieves avoiding most eigendecompositions, named Progressively Projected Newton's Method.

External contributions

Contributions to the global energy and its derivatives can also be added directly, circumventing the need for defining energy expressions. This enhances the usability of SymX, enabling the integration of potentially faster or more robust hand-tuned derivatives when required. Additionally, external contributions make it possible to use energies that require numerical approximations [McAdams et al. 2011; Chao et al. 2010; Barbič 2012] or closed-form derivatives that need intricate procedures to be obtained and cannot be obtained by direct scalar-based differentiation [Lin et al. 2022].

5.4.3 Assembly

Foundations introduced the local-to-global assembly structure abstractly in Section 2.3. This section describes the concrete data structures used by SymX to perform that assembly for multiple DoF sets and changing sparsity patterns.

In this section we describe the assembly for the general case of having multiple sets of degrees of freedom $\mathbf{u}_0, \ldots \mathbf{u}_n$, e.g., one set for deformable volumetric solids, one for cloth models, and one for the rigid body system.

In SymX, all sets are internally concatenated into a global vector $\mathbf{u}$. This establishes a global indexing of the degrees of freedom, which is automatically considered by SymX during

the assembly step. The linear system associated with a Newton iteration then takes the form

$$\begin{pmatrix} \frac{\partial^2 E}{\partial \mathbf{u}_1^2} & \frac{\partial^2 E}{\partial \mathbf{u}_1 \partial \mathbf{u}_2} & \cdots & \frac{\partial^2 E}{\partial \mathbf{u}_1 \partial \mathbf{u}_n} \\ \frac{\partial^2 E}{\partial \mathbf{u}_2 \partial \mathbf{u}_1} & \frac{\partial^2 E}{\partial \mathbf{u}_2^2} & \cdots & \frac{\partial^2 E}{\partial \mathbf{u}_2 \partial \mathbf{u}_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 E}{\partial \mathbf{u}_n \partial \mathbf{u}_1} & \frac{\partial^2 E}{\partial \mathbf{u}_n \partial \mathbf{u}_2} & \cdots & \frac{\partial^2 E}{\partial \mathbf{u}_n^2} \end{pmatrix} \cdot \begin{pmatrix} \Delta \mathbf{u}_1 \\ \Delta \mathbf{u}_2 \\ \vdots \\ \Delta \mathbf{u}_n \end{pmatrix} = - \begin{pmatrix} \frac{\partial E}{\partial \mathbf{u}_1} \\ \frac{\partial E}{\partial \mathbf{u}_2} \\ \vdots \\ \frac{\partial E}{\partial \mathbf{u}_n} \end{pmatrix}, \tag{5.5}$$

where E is the global energy of the simulation. The diagonal blocks in the global Hessian matrix contain the second derivatives of internal energies to a physical system, such as strain energies for deformable objects, while off-diagonal blocks contain the second derivatives of cross-system interactions, such as collisions or attachments.

SymX has default custom parallel data structures to build and return the global gradient and Hessian. The sparse matrix structure in specific, is based on the Blocked Compressed Row Storage (BCRS) format and uses 3×3 matrix blocks for 3D problems. However, our framework can also return the local element gradients and Hessians together with their global indices so that existing simulation systems can use their own data structures.

As discussed, contacts and remeshing, among other things, can change the sparsity pattern of the Hessian matrix between evaluations. However, these changes exhibit strong time coherence, meaning few non-zero elements appear or disappear from one iteration to the next (even if significant changes accumulate over a longer timescale). To efficiently manage these time-coherent, dynamic topology changes, the default BCRS structure in SymX adopts a dual storage strategy. Algorithm 2 provides a high-level overview.

The first is a standard BCRS sparse matrix with all the values and offsets allocated contiguously in memory for high performance. The second is a dynamic list of "buckets", with one bucket per block-row, that is initialized empty at the beginning of each assembly run with the current number of block-rows. To insert a new block, the algorithm checks whether there is a non-zero block in the corresponding position in the BCRS matrix. If so, the block is simply added, avoiding expensive dynamic memory allocations. Otherwise, the block is appended to the corresponding block-row bucket in the second structure. Block insertions, regardless of whether they are added or appended, are performed in parallel using mutexes for thread synchronization.

After the insertion phase, if new blocks have been added to the buckets or any existing blocks have been left as zero in the matrix, the BCRS is rebuilt, which can be efficiently done in parallel. This design keeps the most frequent task, adding blocks to existing non-zero positions, very efficient, with relatively little overhead for the much rarer changes in the non-zero structure.

5.5 Applications

In the following, we show how complex problems in the area of physically-based simulation can be solved using our framework SymX. All of the application examples were implemented with Backward Euler time integration without loss of generality. Note that we do not present an exhaustive list of all that can be accomplished with SymX; but rather, a showcase of use cases for which our system can be effectively employed.

Algorithm 2: Parallel global Hessian $\mathbf{H}$ evaluation and assembly. To avoid data races during parallel execution, the function `AddInPlace` and `AppendToBucketList` implement mutexes.

```
1  H ← 0;                                                  // Previous BCRS matrix zeroed
2  ClearAndResizeBucketList(B);
3  foreach energy E do
4      foreach element e in parallel do
5          d_e ← GatherData(user data, symbol-data maps);
6          H_e ← hess_E(d_e);                              // Call compiled function
7          if project then
8              ProjectToPD(H_e);
9          foreach block-row i in H_e do
10             I ← GlobalIndex(symbol-data maps, i);
11             foreach block-column j in H_e do
12                 J ← GlobalIndex(symbol-data maps, j);
13                 if BlockExists(H, I, J) then
14                     AddInPlace(H, I, J, H_e, i, j);
15                 else
16                     AppendToBucketList(B, I, J, H_e, i, j);
17     if HasSparsityChanged(H, B) then
18         H ← Rebuild(H, B);
```

5.5.1 Non-Linear Material Models

We show how the five constitutive models introduced in Foundations Section 3.3 can be modeled using SymX. We implemented all energy density functions in Figure 5.6, which took just 46 lines of code. These models are relatively complex and usually would require involved processing in the form of differentiation with respect to the deformation gradient and careful application of the chain rule. In SymX however, we can directly use the energy density expression for any given element type and let the framework work out the rest.

Figure 5.7 presents a comparison of these materials in a simulation of a stretched cube with a Young's modulus $E = 1 \times 10^4$ Pa and Poisson ratio $\nu = 0.3$, showcasing the distinctive deformation behavior of such models. We use lagged rotations (constant per time step), a long-standing common practice in computer graphics [Müller and Gross 2004; Kugelstadt et al. 2018], to implement the As-Rigid-As-Possible (ARAP) [Sorkine and Alexa 2007] and fixed co-rotational [Stomakhin et al. 2012] materials in this example. Note that such lagging introduces additional dissipation depending on the time step size [Sanan 2014, Ch. 2.5.1]. We also add a volume conservation term to the ARAP material [Lin et al. 2022] (see also [Stomakhin et al. 2012]).

```
Scalar constitutive_models_energy_density(Energy& energy,
    Matrix& F, Matrix& R,  /* R is assumed constant (lagged) */
    Scalar& lambda, Scalar& mu,
    ConstitutiveModels model)
{
    // Eq. 14 from [Smith et al. 2018]
    if (model == ConstitutiveModels::StableNeoHookean) {
        Scalar mu_ = (4.0/3.0)*mu;
        Scalar lambda_ = lambda + (5.0/6.0)*mu;

        Matrix C = F.transpose()*F;
        Scalar detF = F.det();
        Scalar Ic = C.trace();
        Scalar alpha = 1.0 + mu_/lambda_ - mu_/(4.0*lambda_);
        return 0.5*mu_*(Ic - 3.0) +
            0.5*lambda_*(detF - alpha).powN(2.0) -
            0.5*mu_*log(Ic + 1.0);
    }

    // Eq. 5 from [Smith et al. 2018]
    else if (model == ConstitutiveModels::NeoHookean) {
        Matrix C = F.transpose()*F;
        Scalar Ic = C.trace();
        Scalar logdetF = log(F.det());
        return 0.5*mu*(Ic - 3.0) - mu*logdetF +
            0.5*lambda*logdetF.powN(2.0);
    }

    // Eq. 49 from [Smith et al. 2018, Stomakhin et al. 2012]
    else if (model == ConstitutiveModels::FixedCorot) {
        Matrix I = energy.make_identity_matrix(3);
        Scalar detF = F.det();
        return mu*(F - R).frobenius_norm_sq() +
            0.5*lambda*(detF - 1.0).powN(2.0);
    }

    // Eq. 14 and 36 from [Lin et al. 2022]
    else if (model == ConstitutiveModels::ARAP) {
        Matrix C = F.transpose()*F;
        Scalar Ic = C.trace();
        Scalar detF = F.det();
        return 0.5*mu*(Ic - 2.0*(F.transpose()*R).trace() + 3.0) +
            0.5*lambda*(detF - 1.0).powN(2.0);
    }

    // Eq. 50 from [Smith et al. 2018]
    else if (model == ConstitutiveModels::SaintVenant) {
        Matrix I = energy.make_identity_matrix(3);
        Matrix E = 0.5*(F.transpose()*F - I);
        return mu*E.frobenius_norm_sq() +
            0.5*lambda*E.trace().powN(2);
    }
}
```

Figure 5.6: *Implementation in SymX of the five constitutive models used in the simulation shown in Figure 5.7.*

5.5.2 High-order Lagrangian Finite Elements

The finite element discretization framework was introduced in Chapter 3. Here we show how SymX handles the composition of energy density and element type and the numerical integration needed for high-order elements. The total deformation energy of an element is then given by

$$\phi^e_{\text{FEM}} = \sum_{k=1}^{n_{\text{ip}}} w_k \, \det(\mathbf{J}^e_0) \, \psi(\mathbf{F}(\xi_k, \mathbf{X}^e, \mathbf{x}^e)) \,, \tag{5.6}$$

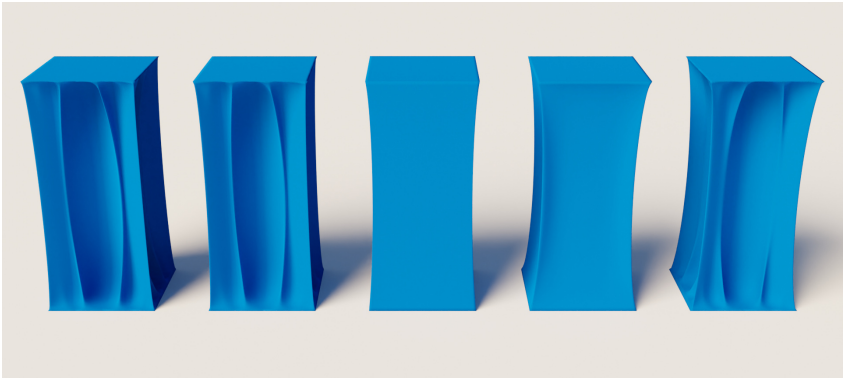


Figure 5.7: *Comparison of different material models in a simulation of a stretched deformable cube. From left to right: ARAP with a volume conserving term, fixed co-rotational, St. Venant-Kirchhoff, Neo-Hookean and Stable Neo-Hookean material. Note that we use* lagged *(constant per time step) rotation matrices for the ARAP and fixed co-rotational energies.*

```
Scalar fem_integration(Energy& E,
std::vector<Vector>& Xe, std::vector<Vector>& xe,
std::vector<std::array<double, 4>>& integration_points,
std::function<Matrix(std::vector<Vector>&,Vector&)> jac,
std::function<Scalar(Matrix& F)> psi)
{
   Scalar sum = E.add_for_each(integration_points,
      [&](Vector& ip)
      {
         Scalar w = ip[0];
         Vector xi = Vector({ip[1], ip[2], ip[3]});
         Matrix Dm = jac(Xe, xi);
         Matrix Ds = jac(xe, xi);
         Matrix F = Ds*Dm.inv();
         return psi(F)*w*Dm.det();
      }
   );
   return sum;
}
```

Figure 5.8: *Implementation of a generic FEM integrator for elasticity in SymX.* `jac` *and* `psi` *are generic element Jacobian and potential energy density functions.*

where ψ is the strain energy density function, n_{ip} is the number of integration points, and w_i represents the quadrature weight. The integration point ξ_k is defined in the coordinate system of the reference element, and $\mathbf{J}_0^e = \mathbf{J}_0^e(\xi_k)$ is the Jacobian of the mapping from the reference element to the physical element in the undeformed configuration (see, e.g., Wriggers [2008]). Figure 5.8 shows the generic implementation of Eq. (5.6) in SymX.

Since the quadrature points and weights are typically constant per element type, we can employ the fixed summation feature of SymX (see Section 5.4.2). As a result, only the evaluation of ψ at a generic integration point needs to be differentiated and compiled. Note that such high level and generic SymX definitions do not degrade simulation performance since the code that describes the expressions is only executed once to generate the optimized

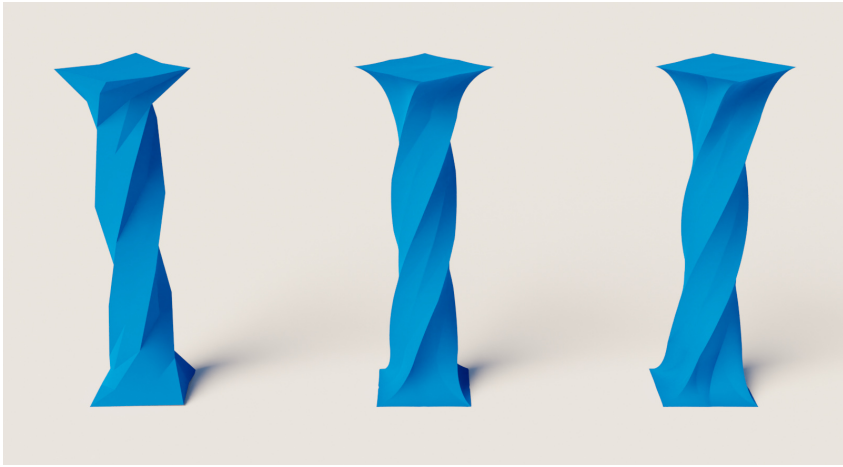


Figure 5.9: *Comparison of linear (left), quadratic (center) and cubic (right) finite elements in a simulation of a stretched and twisted cube.*



Figure 5.10: *Our pipeline seamlessly handles changes in discretization, number of degrees of freedom and sparsity pattern in a cloth simulation with adaptive mesh refinement.*

code that is actually evaluated at runtime. Figure 5.9 shows a comparison of linear, quadratic, and cubic finite elements for the Stable Neo-Hookean material [Smith et al. 2018].

5.5.3 Adaptive Cloth Simulation

We implement a cloth simulation using a non-linear material in combination with a quadratic bending model, strain limiting and Rayleigh damping, in which we used an adaptive mesh

refinement strategy to demonstrate that SymX can handle changes in discretization topology (see Figure 5.10).

We use the Neo-Hookean strain energy for the cloth (using a 2D FEM integrator) and the quadratic bending energy proposed by Bergou et al. [2006], already introduced in Foundations Section 3.7,

$$\phi_b^e(\mathbf{x}^e) = \frac{k_b}{2}(\mathbf{x}^e)^T \mathbf{Q}_e \mathbf{x}^e,$$

where k_b is a stiffness coefficient, $\mathbf{x}^e \in \mathbb{R}^{12}$ are the four unique mesh vertices of two adjacent triangles sharing a common internal edge e, and $\mathbf{Q}_e \in \mathbb{R}^{12\times 12}$ is the internal edge quadratic form, which is constant during the simulation. Implementing this local potential in our system requires the precomputation of the constant matrices $\mathbf{Q}_e$ and just one line of code for the energy:

```
Scalar cloth_bending(Vector& x_e, Matrix& Q_e,
    Scalar& k_b)
{
   return 0.5 * k_b * x_e.transpose() * Q_e * x_e;
}
```

We employ a strain limiting model inspired by the one proposed by Li et al. [2021], where the two eigenvalues of the Green-Lagrange strain tensor $\mathbf{E} = \frac{1}{2}\left(\mathbf{F}^T\mathbf{F} - \mathbf{I}\right)$ are used to measure the strain of a triangle. We use a simple cubic penalty with user-defined stiffness k_{sl} to enforce the constraint using a C^2 local potential:

$$\phi_{\text{sl}}(\mathbf{E}) = \sum_i^2 \begin{cases} k_{sl} A_e (\sigma_i(\mathbf{E}) - \sigma_l)^3 & \text{if} \quad \sigma_i(\mathbf{E}) > \sigma_l \\ 0 & \text{if} \quad \sigma_i(\mathbf{E}) \leq \sigma_l, \end{cases} \tag{5.7}$$

where A_e is the undeformed area of the triangular element, σ_i is the ith eigenvalue of $\mathbf{E}$ and σ_l is the user-defined stretch limiting threshold. The implementation in our system is:

```
Scalar cloth_strain_limiting(Matrix& F, Scalar& area,
   Scalar& sl, Scalar& k)
{
   Vector s = singular_value_2x2(F);
   Vector c = s - sl;
   Scalar e0 = branch(c[0] > 0, area*k*c[0].powN(3), 0);
   Scalar e1 = branch(c[1] > 0, area*k*c[1].powN(3), 0);
   return e0 + e1;
}
```

The singular value decomposition of a 2×2 matrix can be computed using the direct method presented by Blinn [1996].

Finally, for the adaptive mesh refinement we use a quadtree subdivision scheme that splits cells based on the divergence of the normals of the mesh vertices within the quadtree node. Although our refinement algorithm is rather simple, it suffices to show that SymX is capable of handling changes in the number of elements and degrees of freedom.

5.5.4 Contact and Friction

Contact handling with friction is an important part in the simulation of deformable solids and rigid bodies and it is often a great source of complexity of the simulation model and

the simulation software. The IPC [Li et al. 2020] contact and friction models used here are described in Foundations Section 3.6. In this section we show how the contact barrier and the friction potentials can be implemented in our framework.

First, we recall the contact barrier potential as

$$\phi_{\text{contact}}(d) = -k_c(d - \hat{d})^2 \ln(d/\hat{d}), \tag{5.8}$$

where k_c is the barrier stiffness, d the unsigned distance to the contact surface and $\hat{d}$ the maximum influence distance of the collision barrier force. The corresponding code in SymX is

```
Scalar contact(Scalar& k_c, Scalar& d, Scalar& dh)
{
    return -k_c*(d - dh).powN(2)*ln(d/dh);
}
```

Second, we recall the local friction potential from the IPC friction model

$$\phi_{\text{friction}}(y) = \mu f_n \begin{cases} -\frac{y^3}{3\hat{y}^2} + \frac{y^2}{\hat{y}} + \frac{\hat{y}}{3} & \text{if } y \le \hat{y} \\ y & \text{if } y > \hat{y}, \end{cases} \tag{5.9}$$

where $y = \|\mathbf{T}\Delta\mathbf{u}\|_2$ is the slide magnitude with the contact projection matrix $\mathbf{T} \in \mathbb{R}^{2\times 3}$ and the relative displacement $\Delta\mathbf{u}$. The slide/stick threshold $\hat{y}$ is $\Delta t \epsilon_v$, f_n is the contact pressure and μ is the Coulomb's friction coefficient. This energy is implemented in SymX as

```
Scalar friction(Vector& du, Matrix& T, Scalar& mu, Scalar& fn, Scalar& yh)
{
    Vector dut = T*du;
    Scalar y = dut.stable_norm(EPS);
    Scalar f = branch(y > yh, y, -y*y*y/(3*yh*yh) + y*y/yh + yh/3);
    return mu*fn*f;
}
```

Note that we use a stable norm function ($\epsilon = 10^{-14}$m/s) to avoid norm singularities at near-zero. This issue, which also cannot be circumvented using other symbolic tools like SymPy [Meurer et al. 2017], can be avoided when the derivatives are determined by hand due to mathematical simplification. Using stable norm works well in practice. Further discussion about this limitation can be found in Section 5.7.

In these examples we update the primitive contact pairs (point-point, point-edge, point-triangle and edge-edge) using collision detection before each energy evaluation. We employ an octree acceleration structure based on the method presented in Chapter 9 to efficiently find which primitives are in contact. The lists of pairs are rebuilt based on the current state of the simulation and can work with potential mesh refinement.

We use two experiments with extreme contact configurations (see Figure 5.11, 5.12) to show that the system enables robust evaluation of complex contact energies. Point contacts between a deformable mesh and the environment are shown in the armadillo simulation while triangle mesh self-collisions are shown in the twisted cloth simulation which features energies based on triangle-point and mollified edge-edge distance kernels as described by Li et al. [2020].

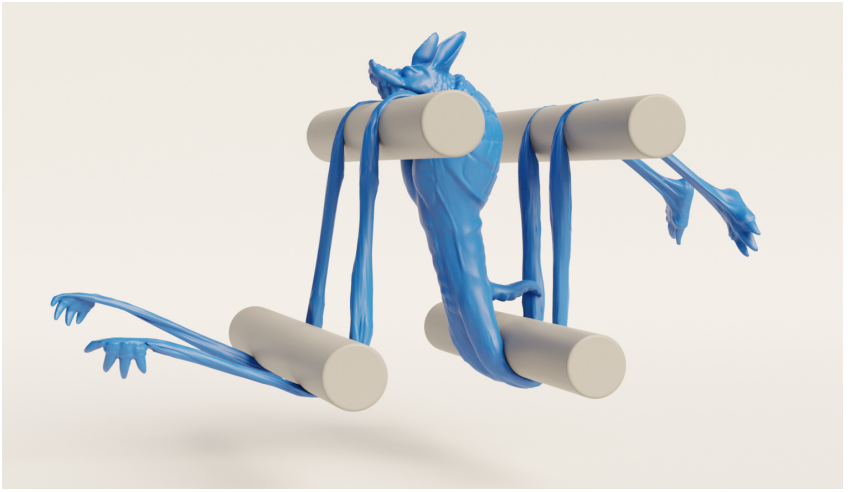


Figure 5.11: *Robust contact handling in a simulation of an armadillo which is extremely deformed by animated cylinders.*

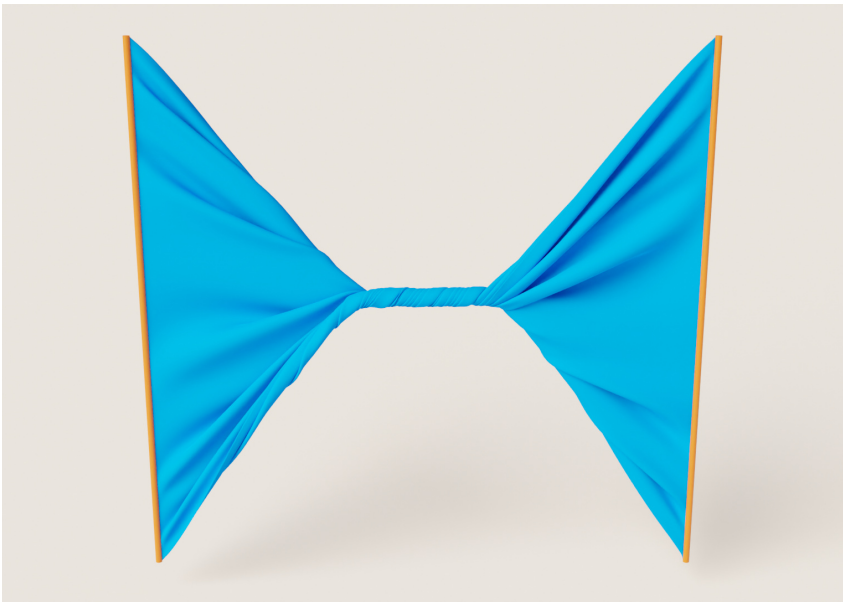


Figure 5.12: *A cloth is twisted which leads to a configuration with thousands self-collisions.*

5.5.5 Coupling Multiple Systems

An important feature of our proposed system is the ability to handle multiple sets of degrees of freedom, a requirement when simulating coupling between different physical systems. Rigid body models are discussed in Foundations Section 3.8, and the global block linear system that arises from multiple DoF sets is described in Section 5.4.3.

Drifting car We simulate a car model (see Figure 5.1) by coupling a rigid body system with joints and deformable volumetric solids for the tires. For the rigid body dynamics and the corresponding inertia terms we use the formulation presented by Macklin et al. [2020]. Alternatively, SymX also supports implementing the potentials for the rigid body formulation introduced by Ferguson et al. [2021] or for Affine Bodies [Lan et al. 2022]. Note that SymX supports the non-linear DoF mappings that typically arise in rigid body simulations (e.g. quaternion manipulations), as these can be incorporated into the energy definitions themselves.

We implement constraint potentials using the penalty method

$$\phi_C = \frac{1}{2} k_C C^2, \tag{5.10}$$

where k_C is the penalty stiffness and C the constraint function. For two connector points a and b with global positions $\mathbf{x}_a$, $\mathbf{x}_b$, velocities $\mathbf{v}_a$, $\mathbf{v}_b$ and two normalized direction vectors $\mathbf{d}_a$, $\mathbf{d}_b$, we define ball joints, direction lock constraints, and slider joints as

$$C_{bj}^2(\mathbf{x}_a, \mathbf{x}_b) = \|\mathbf{x}_a - \mathbf{x}_b\|_2^2 \tag{5.11}$$

$$C_{dl}^2(\mathbf{d}_a, \mathbf{d}_b) = \|\mathbf{d}_a - \mathbf{d}_b\|_2^2 \tag{5.12}$$

$$C_{sj}^2(\mathbf{x}_a, \mathbf{x}_b, \mathbf{d}_a) = \left\| \mathbf{x}_b - \mathbf{x}_a - \left((\mathbf{x}_b - \mathbf{x}_a)^T \mathbf{d}_a \right) \mathbf{d}_a \right\|_2^2 . \tag{5.13}$$

Hinge joints are simply modeled by two ball joints. Additionally, the dampers of the car are implemented using the local potential of a damped spring

$$\begin{aligned} \phi_{\mathrm{ds}}(\mathbf{x}_a, \mathbf{x}_b, \mathbf{v}_a, \mathbf{v}_b) = & \frac{k_{sp}}{2} \left(\frac{\|\mathbf{x}_a - \mathbf{x}_b\|_2}{l_0} - 1 \right)^2 + \\ & \frac{\alpha_{dp}}{2 l_0} \left((\mathbf{v}_a - \mathbf{v}_b) \cdot \frac{\mathbf{x}_a - \mathbf{x}_b}{\|\mathbf{x}_a - \mathbf{x}_b\|_2} \right)^2 , \end{aligned} \tag{5.14}$$

where k_{sp} is the stiffness of the spring, l_0 its the rest length, and α_{dp} the damping coefficient.

Each wheel of the car has its own suspension system composed of multiple energies. A slider in combination with a damped spring models the damper of the car and attaches a rigid body to the chassis which is then linked by a hinge joint to the wheel rim to enable spinning. We use an additional hinge joint for each front wheel to steer the car. Unwanted relative rotations around the slider axes are eliminated by direction lock constraints, which are also used to steer the car. The tires are modeled by linear tet elements using the Stable Neo-Hookean material by Smith et al. [2018] and connected to the rims by attaching contacting mesh vertices with constraints analogous to ball joints. Finally, contact and friction between the tires and the floor and obstacles are handled using the formulation introduced in Section 5.5.4. To simplify collisions, only point-plane contacts between tires and floor are considered in this experiment.

Tumble dryer In the second experiment we simulate a tumble dryer with eight pieces of cloth inside. The drum is attached to the machine's mainframe, which is fixed, by a hinge joint while torque is applied along the rotation axis. The cloth, contact and friction models are kept as described above, the latter two are extended for coupling between cloth and rigid bodies. This is the most complex simulation we present in this chapter in regards to number of distinct

energies with a total of 46, most of them being contact and friction potentials between the primitive geometries of the rigid bodies, cloth and their cross interactions. More multi-system experiments can be found in the accompanying supplemental video.

5.6 Benchmarks

In the first part of this section, we present benchmarks to compare SymX, SymPy [Meurer et al. 2017], TinyAD [Schmidt et al. 2022], and an optimized manual implementation of the Stable Neo-Hookean energy [Smith et al. 2018]. A comparison with Simit for evaluating and assembling the Neo-Hookean energy on linear tetrahedral meshes follows. In the end, differentiation, compilation and evaluation timings and other measurements are presented for all the simulations shown in the previous section. Element projections to positive semi-definiteness were disabled for all experiments, as performing them would distort the assembly runtime results by adding a very significant computational cost to all methods. To prevent Newton's method from getting stuck due to indefiniteness, time steps that were too difficult (e.g. due to too many Newton iterations or a line search not descending) were restarted and half of the time step size was used instead. After a few successful time steps, the time step size was increased again. While for a given simulation this increases the total number of Newton iterations and therefore executions of the global assembly, the average runtime for the derivatives evaluation and assembly, which are the metrics we are actually interested in comparing, are largely unmodified. This correction is only triggered in scenes featuring collisions. All simulations and benchmarks were run on a workstation equipped an AMD Ryzen Threadripper PRO 5975WX with 32 cores, 3.60 GHz and 256 GB of RAM. We used version 12.2.0 of the gcc compiler.

5.6.1 Single Element Benchmark

The first benchmark is the repeated evaluation of the Stable Neo-Hookean energy [Smith et al. 2018], its gradient and its Hessian for a single linear, quadratic and cubic tetrahedral element, respectively. Note that this is a synthetic experiment aimed to assess the performance of evaluating the derivatives in isolation, between different approaches representing different effort requirements. The benchmark results are shown in Table 5.1.

In regards to evaluation times, as expected, the hand-optimized solution is the fastest in all cases, with a gap that grows as the polynomial order increases. Both symbolic differentiation approaches, SymPy and SymX, perform in the same order of magnitude than manual and within 27% of each other, which is also expected as they use the same fundamental principles for differentiation. On the other hand, TinyAD is one order of magnitude slower than the other methods in all cases. While all other approaches result in more compact final expressions due to manual or automatic reductions and simplifications, evaluating derivatives with TinyAD requires traversing the operation graph and applying the chain rule at each node, carrying the gradient and Hessian along. This in turn leads to potentially more redundant operations and less room for compiler optimizations. While in TinyAD the full derivative information is known at all intermediate operations, the other methods optimize for the final derivatives alone.

SymPy's differentiation times for the linear, quadratic and cubic functions were 20.01 s, 24.19 min, 5.93 h while SymX took 1.75 ms, 4.19 ms and 14.8 ms, respectively. For reference,

Table 5.1: *Average evaluation time $t_{\text{eval.}}$ of the stable Neo-Hookean energy, its gradient and Hessian for a single linear, quadratic and cubic tetrahedral finite element. Relative time with respect to SymX in parenthesis.*

Method	linear $t_{\text{eval.}}\,[\mu s]$	quadratic $t_{\text{eval.}}\,[\mu s]$	cubic $t_{\text{eval.}}\,[\mu s]$
Manual	0.16 (0.88×)	2.83 (0.48×)	26.17 (0.41×)
SymPy	0.23 (1.27×)	6.39 (1.08×)	49.32 (0.77×)
SymX	0.18 (1.00×)	5.91 (1.00×)	64.13 (1.00×)
TinyAD	7.21 (40.06×)	365.13 (61.78×)	5300.39 (82.65×)

compilation times were 0.432 s, 2.78 s and 25.1 s, respectively, which completely dominates the pre-simulation phase.

These results highlight that SymPy, while being a powerful general-purpose tool, it was never intended for handling this type of complex expressions with such high-performance demands. Consequently, a user experimenting with complex materials or high-order integrators will face lengthy processing times. In any case, the code generated by SymPy is in fact relatively close to SymX's output in terms of evaluation performance, which validates SymX differentiation capabilities. We experimented with SymPy's `simplify` in an attempt to further reduce the final expressions complexity in addition to the already applied common subexpression elimination. However, it timed out after eight hours already for the Stable Neo-Hookean linear tet potential.

Finally, although proprietary mathematical engines (e.g., Mathematica, Matlab or Maple) might potentially produce derivatives faster than SymPy, they conflict with our accessibility and distribution goals outlined in Section 5.1, since they would introduce external licenses to operate the pipeline.

We also would like to accompany the benchmarks with the qualitative experience the different solutions provided when designing the experiment. While just a single user experience, it is worth reporting that an expert took roughly a work-day to differentiate, implement, test and optimize the hand-written solution. Using SymPy, however, it took just about an hour for an experienced user to reach the solution if we exclude the time it took for SymPy to compute the derivatives themselves. Note that while generic symbolic tools have comprehensive differentiation and code generation modules, the exact functionality needed for this specific task was not readily available and some scripting was required. The code footprint of the SymPy solution was significantly larger than for the other three approaches due to the code being divided between scripts and main application. Finally, both TinyAD and SymX presented the most streamlined processes, allowing for a trained user to reach the solution in about ten minutes, including the time to differentiate and to compile the all the expressions. This is due to both tools being fully-automated and specifically designed for this type of task. Notably, typos and other bugs were much less problematic as the code itself is very short and corrections to the root expressions have an immediate impact, unlike the two previous methods.

5.6.2 Simulation Benchmark

In this experiment we run a benchmark in a more realistic setting where we compare the total runtime of the derivatives evaluation and assembly during a simulation. We use the simulation setup shown in Figure 5.9 with linear, quadratic and cubic tetrahedral elements sharing the

Table 5.2: *Simulation benchmark of the stretched cube example with 137K degrees of freedom using linear, quadratic and cubic tetrahedral finite elements. Timings are averaged per Newton iteration and include the evaluation of the energy and its derivatives as well as the assembly. The linear system solve is not included in the timings.*

Method	linear t_{total} [ms]	quadratic t_{total} [ms]	cubic t_{total} [ms]
Manual	12.5 (0.84×)	17.9 (0.74×)	28.8 (0.54×)
SymPy	14.3 (0.97×)	22.2 (0.91×)	46.6 (0.88×)
SymX	14.8 (1.00×)	24.3 (1.00×)	52.9 (1.00×)
TinyAD	698.9 (47.22×)	1269.7 (52.25×)	19125.2 (361.53×)

Table 5.3: *Number of degrees of freedom and average evaluation and assembly timings per Newton iteration of the application examples from Section 5.5.*

Figure	Scene	# DoF	t_{total} [ms]
5.7	ARAP	307623	38.79
5.7	Fixed Corot.	307623	35.40
5.7	StVK	307623	37.85
5.7	Neo-Hookean	307623	35.83
5.7	Stable NH	307623	35.10
5.9	Beam linear	189	0.09
5.9	Beam linear	1728	0.24
5.9	Beam linear	11253	1.19
5.9	Beam quadratic	975	0.24
5.9	Beam quadratic	11253	1.75
5.9	Beam quadratic	80703	14.42
5.9	Beam cubic	2793	0.92
5.9	Beam cubic	35328	10.93
5.9	Beam cubic	262353	96.46
5.10	Adaptive cloth	6339 – 24903	4.19
5.11	Armadillo	195510	29.56
5.12	Twisted Cloth	482403	95.64
5.1	Car	17220	2.89
5.2	Dryer	244836	73.33

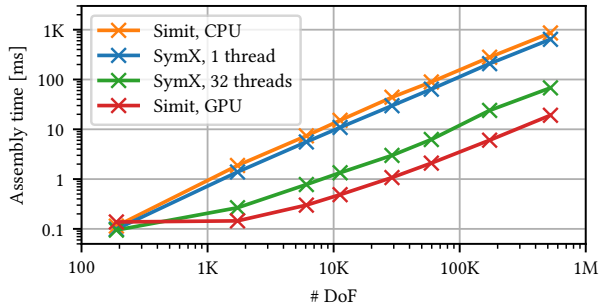


Figure 5.13: *Total evaluation and assembly time of the Hessian for the dynamic simulation of a cube with Neo-Hookean material and linear tetrahedra. Simit does not parallelize assembly on the CPU.*

same 137K degrees of freedom, and present the timings in Table 5.2. TinyAD and SymX use their respective built-in data structures and assembly. Manual and SymPy use SymX's global data structures by being declared into a simulation as external contributions.

The manually optimized solution is again the fastest in all cases and TinyAD is again the slowest, with SymX being up to 361× faster for cubic elements. SymX and SymPy are again roughly matched in performance, and both close the gap to the manual solution due to the more realistic execution conditions. The implementation effort of using TinyAD and SymX is again similar, both the lowest by a significant margin, as the global solutions are generated directly from the mathematical expressions and global data structures are provided. Both SymPy and manual evaluations allowed for further optimizations, which made the assembly faster than using the generic evaluation and assembly procedure in SymX, and resulted in the SymPy solution being ultimately roughly 10% faster, not by own merits, but by virtue of a hand tuned assembly. SymX on the other hand must account for arbitrary element types, and different function inputs and outputs, information which is only made available at runtime.

In any case, it is important to emphasize that the difference in total simulation runtime between methods is much less pronounced than what Table 5.2 might suggest, as evaluation of derivatives and assembly is just part of the total execution time. The largest portion of the total simulation runtime is usually spent in the linear system solve, which often plays a significant role as performance equalizer.

SymX employs a 3×3 Block Diagonal Preconditioned Conjugate Gradient linear system solver with a forcing sequence tolerance [Nocedal and Wright 2006]. In our experiments, the average runtimes for the linear solves are 29.8 ms, 52.9 ms and 256.4 ms for the linear, quadratic and cubic SymX's simulations, respectively. Both the manual and SymPy simulations used SymX's linear solver, while TinyAD uses its own solver. Note that these times are for reference and that analyzing the role of linear system solvers in optimization time integrators is outside the scope of this chapter as different choices can have a large impact in overall simulation runtime. For example, choosing between direct or iterative solver is a decision that might be conditioned by simulation size or expected numerical stiffness and which might drastically change the total simulation runtime independently of the derivative computation.

In addition to these benchmarks, we also show timings of the application examples from Section 5.5 in Table 5.3.

5.6.3 Comparison with Simit

In this section we evaluate and compare the performance of SymX with Simit [Kjolstad et al. 2016]. While Simit does not offer differentiation capabilities, it does provide code generation and automates the assembly process over sets of abstract elements. For the experiment, we use the previously introduced stretched cube experiment with linear tetrahedral elements and the classic Neo-Hookean material model. We measure the average time required for the derivatives evaluation and the assembly of the global Hessian for varying mesh resolutions. The results are shown in Figure 5.13.

Note that while SymX supports parallel evaluation and assembly (Section 5.4.3), Simit only provides sequential evaluation and assembly on the CPU. Comparing Simit and SymX in single thread execution, the evaluation and global assembly in SymX is between 15% and 48% faster depending on the resolution (34.7% on average over all resolutions). These competitive results for SymX validate its evaluation and assembly procedures in relation to established tools. With multi-threading enabled, we measure speedups between 9.5× and 12.7× in comparison to Simit if we exclude simulations with fewer than 6075 degrees of freedom. The original figure also includes runtimes corresponding to Simit's GPU assembly for this experiment. Using a NVIDIA GeForce RTX 3090 Ti, Simit reports global assembly runtimes between 2.6× and 4.0× faster than SymX's multithreaded execution, again excluding simulations with fewer than 6075 degrees of freedom.

It is important to note that DSLs come with challenges beyond the potentially large number of derivatives needed to be sourced (e.g. for the dryer scene of Figure 5.2): changing topologies, minimizer, linear solver and line search, are some of the components that need to be done either in the DSL scripting language itself or communicating with the host language and device.

5.6.4 Compilation Times

Finally, we also present the timings and memory requirements associated with differentiation and compilation for all the experiments in Table 5.4. Here, T_{diff} denotes the total time needed for differentiating and generating the code for all the expressions in the simulation and T_{comp} the time for gcc to compile the generated C++ code. Concerning the memory requirements, the peak memory column indicates the maximum storage needed during differentiation and the binary size is the sum of all the binaries generated for that simulation, which is shown per scene for illustration purposes. Timings for differentiation and compilation strongly correlate to expression complexity, which explains why the beam simulation with the cubic FEM discretization takes the longest to differentiate and compile. Our memory requirements during differentiation are very low, specially in comparison to relevant methods such as the one by Herholz et al. [2022] which compiles the whole problem, needing tens of gigabytes to differentiate moderately sized simulations. Additionally, the disk space required to store our binaries is very modest, in contrast to the gigabytes needed by Desai et al. [2022].

5.7 Limitations and Future Work

Expression simplification Currently, our system cannot mathematically simplify expressions. Although mathematical simplification has limited performance implications, as shown by Herholz et al. [2022], it would certainly help further reducing the gap to manually opti-

Table 5.4: *Measurements regarding SymX's initialization. T_{diff} indicate total differentiation and code generation runtime, and T_{comp} compilation times. Compilation was carried out from scratch, that is, no cached function was loaded from disk. Note that all energies are initialized in parallel. Furthermore, the table shows the peak memory consumption needed during differentiation and the total size of the output binaries.*

Figure	Scene	T_{diff} [ms]	T_{comp} [ms]	Memory [kB]	Binary [kB]
5.7	ARAP	1.5	675.0	61	182
5.7	Fixed Corot.	1.3	669.7	57	178
5.7	StVK	1.9	853.7	115	180
5.7	Neo-Hookean	1.4	752.0	72	182
5.7	Stable NH	1.7	737.8	68	182
5.9	Beam linear	1.6	722.9	68	182
5.9	Beam quadratic	12.1	3273.3	309	260
5.9	Beam cubic	40.3	26309.4	1090	498
5.10	Adaptive cloth	1.6	768.4	118	332
5.11	Armadillo	4.4	1395.6	483	775
5.12	Twisted cloth	3	1255	1229	873
5.1	Car	6.7	1655.2	545	911
5.2	Dryer	10.3	3676	5889	4505

mized code. In a similar spirit, directly supporting vectors and matrices in the expression graph instead of eagerly reducing all quantities to scalar operations might aid the search for more compact expressions. Finally, while also not a fundamental limitation, SymX currently only works with fixed element sizes. Handling element connectivity with dynamic sizes is left for future work.

Analytic projections Although our system can project element matrices to positive semi-definiteness, some approaches can exploit the properties of the eigenstructure of the material models themselves [Kim et al. 2019; Smith et al. 2019, 2018] to perform a more efficient projection. We believe that the techniques we present in this chapter could be enhanced with analytic eigenstructures to conveniently and efficiently differentiate material models defined in terms of scalar invariants.

Tensor identities SymX currently employs a scalar-based engine for the representation and differentiation of expressions, and therefore it does not implement tensor identities or tensor differentiation rules in its current form. Consequently, it cannot independently find derivatives for which such identities are required. However, it is possible to incorporate such energies in SymX as external contributions, for instance using the closed form expressions of the ARAP derivatives presented by Lin et al. [2022]. This is a limitation of the scalar nature of the current engine, a constraint shared with other scalar-based engines, not of the overall concept presented. This limitation could be addressed in future work by extending the symbolic engine to handle tensor identities.

Singularities Arguably the most significant limitation of symbolic and automatic differentiation is that some expressions, while analytically differentiable, may contain partial expressions that are non-differentiable. Therefore, evaluating the result near a non-differentiable point in the intermediate expression may cause the intermediate result to become numerically unstable. Typically these kinds of issues occur when the scalar expression contains norms, square roots or more generally fractional powers, as we have already seen in the symbolic definition of the friction energy, Section 5.5.4. Users may be taught to be wary of these issues in the presence of such expressions and apply workarounds like smooth approximations provided by the framework, but ultimately this is not foolproof. A mechanism for automatic reformulation of the expression to avoid such problems would be an improvement. In the interim, the system could be augmented to optionally detect such potential problems and notify the user, so that they may try a different formulation or use the stable operators provided.

5.8 Impact and Adoption

Since its publication, SymX has been extended and adopted in ways that validate the core design decisions of the original work. The most significant technical extension was the addition of adjoint differentiation for inverse simulation. As described in Chapter 4, the adjoint method enables differentiating through an equilibrium solve without forming explicit state sensitivities. Integrating it into SymX extended the original forward-simulation pipeline into a fully differentiable one. Local symbolic expressions now provide not only energies, gradients, and Hessians with respect to the simulation state, but also the mixed derivatives needed to optimize over material parameters, actuation variables, geometric design, and mode.

SymX has been used as the core mathematical infrastructure in several published works:

- *STARK: A Unified Framework for Strongly Coupled Simulation of Rigid and Deformable Bodies with Frictional Contact* [Fernández-Fernández et al. 2024], discussed in Chapter 6. STARK uses SymX as differentiation and assembly infrastructure inside a larger strongly coupled simulator for rigid bodies, deformables, joints, motors, contact, and friction.

- *Micropolar Elasticity in Physically-Based Animation* [Löschner et al. 2023], where SymX supports the implementation of richer continuum models with additional rotational degrees of freedom.

- *Strongly Coupled Simulation of Magnetic Rigid Bodies* [Westhofen et al. 2024], which uses the same energy-based modeling and derivative-generation workflow for magnetic interactions between rigid bodies.

- *Curved Three-Director Cosserat Shells with Strong Coupling* [Löschner et al. 2024], which relies on generated derivatives for strongly coupled shell energies with director fields.

- *Interactive Facial Animation with Real-Time Shell and Contact Simulation* [Fernández-Fernández et al. 2025a], discussed in Chapter 8. In that chapter, SymX is used in the inverse quasistatic fitting stage to obtain the derivatives required for adjoint-based optimization of non-uniform shell stiffness fields.

- *Progressively Projected Newton's Method* [Fernández-Fernández et al. 2026], discussed in Chapter 7. PPN builds on simulation systems where local energy derivatives are assembled

into global Newton systems, and therefore directly benefits from the type of derivative and assembly infrastructure provided by SymX.

These works illustrate the main practical outcome of SymX: once the cost of implementing and maintaining derivatives is reduced, more effort can be spent on modeling, coupling, solver design, and validation.

Further, the following are published works that follow or build upon the concepts presented in SymX:

- *A mesh-based simulation framework using automatic code generation* [Herholz et al. 2024].
- *Iskra: A System for Inverse Geometry Processing* [Mahmoud et al. 2026].
- *Locality-Aware Automatic Differentiation on the GPU for Mesh-Based Computations* [Dodik et al. 2026].

CHAPTER 6

STARK: Strongly Coupled Rigid and Deformable Simulation

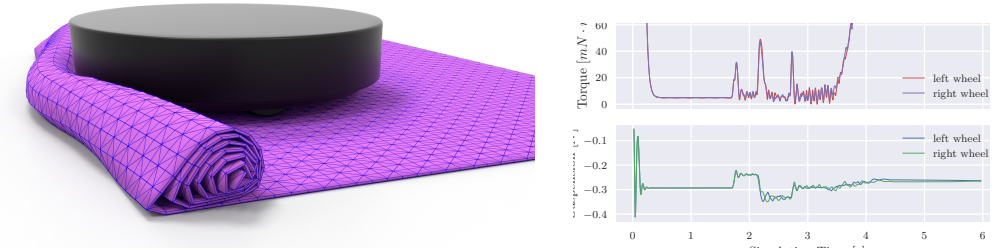


Figure 6.1: *Simulation of a vacuum cleaning robot driving straight into a deformable towel on the floor. On the left, the robot drives over the loose end of the towel and deforming it while remaining stuck. On the right, the corresponding motor torque and suspension stress is shown.*

This chapter presents STARK as the integrated simulation-system contribution of the thesis, adapted from the original STARK publication [Fernández-Fernández et al. 2024]. The implementation is open source[1].

The Foundations chapters introduced optimization time integration, local potentials, Newton-type solvers, rigid body models, deformable models, contact, friction, and local-to-global assembly (Chapters 2–3). The previous chapter, SymX (Chapter 5), provided the derivative and assembly infrastructure that makes local symbolic energies executable. STARK brings these pieces together into a single strongly coupled simulator that handles rigid bodies, deformables, joints, motors, IPC contact, and friction in one variational solve.

Later chapters build on this machinery: PPN (Chapter 7) refines Newton/Hessian stabilization for the solver layer, and IFA (Chapter 8) leverages the coupling and differentiability for inverse, contact-aware graphics applications.

[1] The STARK source code is available at `https://github.com/InteractiveComputerGraphics/stark`.

6.1 Introduction

Simulation has become an increasingly important part of robot learning, design, and testing. Examples range from corner case identification for self-driving cars, data generation for robot reinforcement learning, or shorter development and testing cycles for consumer robots. One aspect to this end is accurate simulation of robot-object interaction for which a number of popular physics engines can be used: ODE [Smith 2026], Bullet [Coumans and Bai 2026], MuJoCo [Todorov et al. 2012], Dart [Lee et al. 2018], Chrono [Tasora et al. 2016], and PhysX [NVIDIA Corporation 2026b].

Here, we are interested in the simulation of deformable objects as such objects are an integral part of everyday life. Robotic bin picking systems in intra-logistics, for example, must be able to handle a variety of flexible, fabric or limp objects and robot vacuum cleaners have to deal with household objects from plants, clothing to carpets. Precise models for contact, friction and material response, together with adequate solvers for systems of deformables *and* rigid bodies are essential to simulate such interactions accurately.

Often, simulation of different mechanical systems (e.g. cloth, rigid bodies, deformable objects, etc.) is handled in two stages: the different systems are first considered in isolation, sometimes even with different simulation frameworks, followed by the inter-system interactions, such as the interaction between the wheel and the towel in Figure 6.1. Such *weak coupling* can lead to inaccurate results, or even robustness issues, as errors are not minimized in a global sense and might be simply transferred between systems during the coupling iterations. This problem is magnified in the presence of complex and stiff interactions between the different systems, such as the soft cloth being squeezed between the robot and the floor in Figure 6.1.

In contrast, *strongly coupled* simulations handle all the systems in a single stage, including the interactions between them. This often results in more stable results in challenging scenarios, but usually comes at a higher computational cost. Second-order methods, which use second-order derivative information, are often preferred in this context as they provide better guarantees than first-order methods, which only use gradient information, such as Position Based Dynamics [Bender et al. 2013] or Projective Dynamics [Bouaziz et al. 2014]. Especially when simulating stiff materials or complex contact configurations, the convergence of first-order methods can grind to a halt [Lan et al. 2023].

Implementation complexity however is typically significantly higher in second-order methods since global data structures, representing all the involved physical systems and interactions, must be built to solve large linear systems of equations. This complexity is further exacerbated in simulations of multiple interacting systems as an increasing number of different types of primitive pairs for collision and friction must be accounted for.

We use a unified global potential energy formulation to model all phenomena in the simulation, from internal mechanical stresses to joints and collisions, for which a minimizer represents a configuration where the global balance of forces is satisfied. Such an approach is well-established and can be found in related works such as [Gast et al. 2015; Li et al. 2020; Chen et al. 2022]. Beyond the benefits it provides in terms of robustness, this formulation has the additional advantage that it concentrates the definition of the problem into a single global expression. STARK uses this to its advantage and employs SymX to automate the generation of first- and second-order derivatives of all energy potentials in the simulation, which otherwise would be a rather time-intensive task to do manually. Very importantly, such differentiation facilities are also made available to STARK users by its API, facilitating expanding on the

default capabilities.

The Foundations chapter introduced the variational and numerical ingredients underlying this formulation: incremental potentials, local-to-global assembly (Section 2.3), Newton-type solvers (Section 2.4), rigid and deformable models, and contact/friction potentials (Chapter 3). This chapter focuses on their integration into a broader simulator and on providing real-world validation experiment for robot-cloth interactions.

In detail, our contributions are:

1. A complete simulation pipeline named *STARK* with support for rigid body systems with joints and non-linear deformable materials (volumetric and cloths/shells), with frictional contacts between all of them. STARK uses state-of-the-art second-order optimization time integration techniques, which coupled with the IPC model for contacts, ensures penetration-freeness and great stability even for very challenging simulations. It is shipped with non-linear constitutive models for strain, integrated with FEM, and strain limiting, as well as bending for cloths and shells. The proposed system employs a clean C++ API for programmatically modeling robotic systems and environments and it is designed so that the system can be easily expanded with new materials, joints or constraints.

2. We present simulations involving non-linear materials undergoing large deformations coupled with rigid body systems connected with joints and powered by motors. Most notably, we validate STARK's capabilities to reproduce the outcome of different scenarios carried out in a laboratory setting. In particular, we study the interactions between a mobile vacuum cleaning robot and a towel in different configurations and types of floor. In a second experiment we simulate a parallel jaw gripper grasping a deformable plastic cup, and verify that the simulation follows the outcome predicted by static friction analysis.

The chapter is structured as follows: we discuss related work in Section 6.2, highlight the main components used in our framework in Section 6.3, present our validation experiments in Section 6.4 and conclude the chapter in Section 6.5.

6.2 Related Work

This discussion reflects the state of the art at the time the original research was conducted [Fernández-Fernández et al. 2024].

ODE [Smith 2026], Bullet [Coumans and Bai 2026], MuJoCo [Todorov et al. 2012], Dart [Lee et al. 2018], Chrono [Tasora et al. 2016], and PhysX [NVIDIA Corporation 2026b] are among the most commonly used physics engines for simulating entire robotic applications. They are also used in robotics simulators like Gazebo [Koenig and Howard 2004], Webots [Cyberbotics Ltd. 2026], CoppeliaSim (formerly V-Rep) [Coppelia Robotics, Ltd. 2026], and Isaac Sim [NVIDIA Corporation 2026a] that provide further visualization, sensor simulations, and interfaces to robotic software frameworks. Liu and Negrut [2021] present a discussion on the importance of physics-based simulators in robotics. Körber et al. [2021] and Erez et al. [2015] provide experimental comparisons of these physics engines and simulators in the context of rigid body simulation.

Simulators with focus on real-time performance (all of the above except Chrono) do not provide the required features to run the complex simulations we aim for. The lack of proper

continuum mechanics based material modeling or the very limited self-collision resolution guarantees for deformable objects make it impossible to handle a simulation like the one shown in Figure 6.1. While it is possible to couple deformable and rigid bodies simulators [Bai and Liu 2014], such coupling approaches require expert knowledge not only of the mechanics of the problem but also of the simulators themselves, potentially involving significant internal modifications to achieve true strong coupling.

PhysX is integrated in a larger proprietary GPU-based physics simulation environment for reinforcement learning research, called IsaacGym, and support for linear continuous materials with FEM is added [Makoviychuk et al. 2021]. Huang et al. [2022] demonstrate that the FEM model provided in IsaacGym performs well on a variety of grasping simulations validated with real-world experiments, although it does not provide very accurate results with shell-like objects. Further, it was demonstrated that the IPC model for frictional contact outperforms the contact model provided by IsaacGym [Kim et al. 2022] in a collection of grasping simulations validated with real-world experiments.

On the other hand, Chrono [Tasora et al. 2016] supports simulation of deformable solids via linear and non-linear FEM for volumetric objects and shells, rigid body systems and frictional contact, which makes it the natural choice for our target simulations from the aforementioned available solutions. However, we faced challenges with Chrono as it did not provide stable solutions in the presence of tight contacts and penetrations occurred. This was observed already in simulations less complex than the ones shown in this chapter.

Finally, systems based on the same simulation techniques as STARK have been discussed in the literature, e.g. [Chen et al. 2022]. However, actual open-source code has not been made available and evaluations of such systems in reproducing complex real-world scenes have not been presented.

In this landscape of simulation solutions, when available options are insufficient, researchers are left with the option of developing their own code or, most often, modifying other researchers' reference implementations. This typically leads to new or rehashed implementations tailored to specific types of problems with little regard for further development. For instance, the authors of the IPC method have provided reference implementations of their contact model in different contexts including volumetric deformables [Li et al. 2020], co-dimensional objects [Li et al. 2021], and rigid bodies [Ferguson et al. 2021], as well as releasing a toolkit [Ferguson et al. 2026] with a set of common functionalities to assist with the implementation of their method. These reference implementations have already been adopted by researchers to solve specific robotics problems. For example, Kim et al. [2022] propose IPC-GraspSim, which extends the original IPC reference implementation to model the specific problem of parallel-jaw grasping. While the conclusion of their studies is positive, as these new contact models do indeed outperform existing solutions, the community is now left with two different reference implementations: the original simulator that can handle contacts accurately, but does not support grippers, and the new simulator that is effectively a fork specialized for a single type of problem. This highlights the issue that, although new methods seem to work better than existing simulation tools, there is no unified platform that can offer easy access to these advanced methods for further development in robotics.

STARK is designed to fulfill this role, providing a comprehensive platform for state-of-the-art techniques in simulation in the context of optimization time integration. Crucially, STARK is built on top of well-studied and tested simulation techniques and is itself, as a system, validated with real-world experiments. Additionally, it is designed to facilitate adding new components, such as controllers, and new materials, allowing researchers to test their

models with ease in a rich-featured environment.

6.3 Concept

The Foundations chapter (Chapters 2–3) introduced the mathematical substrate and modeling used here in detail. This section presents a high level overview for the specific design decisions made in STARK.

6.3.1 Optimization Time Integration

STARK advances the coupled state by minimizing a global objective that aggregates inertial, elastic, joint, motor, contact, and friction contributions. Let us first establish the relation between a local potential energy ϕ_i and its corresponding generalized force $\underline{\mathbf{f}}_i$ as

$$\underline{\mathbf{f}}_i = -\mathbf{G}^T \frac{\partial \phi_i}{\partial \underline{\mathbf{x}}}, \tag{6.1}$$

where $\mathbf{G}$ is the kinematic map ($\dot{\underline{\mathbf{x}}} = \mathbf{G}\underline{\mathbf{v}}$) [Bender et al. 2014]. For particle systems and mesh discretizations, $\mathbf{G}$ is the identity matrix and the generalized quantities reduce to the usual nodal positions and velocities. For rigid bodies, $\underline{\mathbf{x}}$ is the generalized position vector (concatenation of translations and a quaternion for orientation) and $\underline{\mathbf{v}}$ is the generalized velocity vector (linear and angular velocities).

A set of values for the degrees of freedom $\mathbf{u}$ (typically positions or velocities) that fulfills the balance of forces is therefore a minimizer of the global potential energy

$$\sum_i \mathbf{f}_i(\mathbf{u}) = \mathbf{0} \longleftrightarrow \mathbf{u} = \arg\min_{\mathbf{u}} \sum_i \phi_i(\mathbf{u}), \tag{6.2}$$

which can be found using Newton's method (Section 2.4)

$$\frac{\partial^2 E(\mathbf{u}^k)}{\partial \mathbf{u}^2}(\mathbf{u}^{k+1} - \mathbf{u}^k) = -\frac{\partial E(\mathbf{u}^k)}{\partial \mathbf{u}}, \tag{6.3}$$

for a sequence $\mathbf{u}^{0\dots m}$ until the residual is below a threshold $\left\| \frac{\partial E(\mathbf{u}^m)}{\partial \mathbf{u}} \right\|_\infty < \epsilon$. For the experiments in this chapter, we set $\epsilon = 1 \times 10^{-4}$ N.

The goal of STARK is to reliably reach accurate balance of forces in tough non-linear problems with potentially bad conditioning due to stiff non-linear materials or strong contacts. We use Newton's method due to its strong convergence guarantees and super-linear convergence close to the solution. Support for projections to positive-definiteness of the local Hessians is provided (see Section 2.5 and Chapter 7).

6.3.2 Rigid Bodies and Implicit Constraints

We use the inertial energy for rigid body dynamics as defined by Macklin et al. [2020] and introduced in Section 3.8:

$$\phi_{\text{RB}} = \frac{1}{2}(\underline{\mathbf{v}}^{n+1} - \hat{\underline{\mathbf{v}}})^T \underline{\mathbf{M}}^n (\underline{\mathbf{v}}^{n+1} - \hat{\underline{\mathbf{v}}}), \tag{6.4}$$

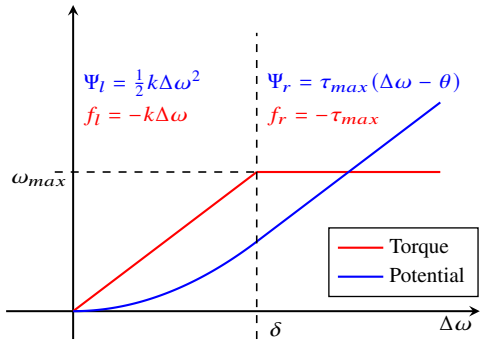


Figure 6.2: *Torque profile of the implicit motor controller provided in STARK. The input is $\Delta\omega$ which is the difference between the target and the current angular velocity of the bodies connected by the motor in the direction of the motor axis. There are two regimes separated by $\Delta\omega = \delta$: on the left, the application of the torque is progressive and proportional to how far from the target velocity the motor is and on the right, the maximum torque is applied. The rate of application $k = \frac{\tau_{max}}{2\delta}$ and the offset $\theta = \frac{\delta}{2}$ are given by δ, which controls the sensitivity of the torque application.*

where $\underline{\mathbf{M}}$ is the generalized mass matrix which contains the mass and the inertia tensors of the rigid bodies. The superscripts indicate the time step, $\hat{\underline{\mathbf{v}}}$ is the inertial generalized velocity of the rigid bodies $\hat{\underline{\mathbf{v}}} = \underline{\mathbf{v}}^n + \Delta t\underline{\mathbf{a}}^n$, where $\underline{\mathbf{a}}^n$ is the generalized acceleration which includes linear and angular accelerations. Rayleigh damping is also provided.

In order to allow for the composition of rigid body systems, STARK implements a collection of constraints, such as ball joints, hinges, sliders, dampers and motors, all of which are handled implicitly within the optimization time integration in order to unconditionally ensure stability. Further, an implicit controller is also provided which enables smooth application of forces with limits. We employ this controller in the motors of the vacuum cleaning robot to smoothly maintain the robot's operational velocity while ensuring the motors respect the maximum stall torque given by the specifications, and in the gripper hand to ensure that it operates at a target close-in velocity but that cannot deliver more than a certain grip pressure. Figure 6.2 shows the power profile of the implicit motor.

6.3.3 Deformable Solids with FEM

STARK uses non-linear hyperelastic constitutive models, e.g., Neo-Hookean, to model embedded (shells and cloths) and volumetric deformable objects. These material models can be included in the global formulation by using the Finite Element Method and integrating the strain energy density over the deformable objects (see Sections 3.2–3.3 in Foundations). Bending is modeled with the method of Bergou et al. [2006]. Using continuous models provides more controllability and higher fidelity than mass-spring systems or networks of chained rigid bodies and, together with the tight balance of forces that the Newton solver can achieve efficiently, ensures that deformations are convergent under space and time refinement. Additionally, inertial- and strain-based Rayleigh damping is incorporated in order to improve the solution for materials that are more dissipative, such as cloth.

	Carpet								Polished floor							
Towel config	1	2	3	4	5	6	7	8	1	2	3	4	5	6	7	8
Lab outcome	O	O	O	S	S	S	S	S	O	O	P*	P	P*	P	O*	S
STARK outcome	O	O	O	S	S	S	S	S*	O	O	P	P	P	P	O	S

Table 6.1: *Outcomes of the 16 setups of the robot-towel experiments, categorized as (P) robot pushing the towel, (O) driving over the towel or (S) getting stuck. Each lab experiment was run 5 times and the most frequent outcome is taken. Simulations are deterministic. In setup Polished-3, marked with *, the robot made it over the towel twice. In Polished-5 and -7 it got stuck twice. In setup Carpet-8 the simulation successfully replicates the real-world outcome (robot gets stuck due to the same underlying interactions) but there are discrepancies in the intermediate results as the towel does not bend.*

6.3.4 IPC Contact and Friction

STARK supports implicit frictional contact for rigid bodies and deformables, volumetric and embedded, including collisions between them, and self-collisions, which no other open-source simulation currently offers. We use the IPC contact and friction models [Li et al. 2020], introduced in Foundations Section 3.6, which meet the high accuracy and robustness requirements of STARK. The high level idea of IPC is that repulsive forces are applied whenever any pair of surface primitives (triangles, edges, or points) comes closer than a very small user-defined distance $\hat{d}$. We use an octree-based procedure based on the work presented in Chapter 9 to find contact candidate pairs efficiently. STARK supports logarithmic based contact forces (as in the original IPC paper) as well as cubic based forces, which are less non-linear and offer better convergence in our experiments. A crucial characteristic of the model is that the stiffness of the contact force response is adapted throughout the simulation to guarantee that enough force can be built up to resolve all the contacts in a given time step, ensuring, in turn, penetration-freeness. Intersection tests are carried out in the line search of Newton's method (Section 2.6) to guarantee that no invalid configuration is ever considered.

In regards to friction, IPC proposes a smoothed unified potential based on Coulomb's friction that transitions between stick and slide within a small velocity threshold ϵ_v. While perfect sticking cannot be captured in this solution, the error induced by the smoothing can be severely reduced by choosing a small ϵ_v, at the cost of higher runtime.

IPC was originally developed in the context of computer graphics simulations but has been already employed with success in the context of robotics [Kim et al. 2022]. STARK employs IPC not only because of its proven accuracy but also due to the strong guarantees it provides and the flexibility to be used in environments with both deformable and rigid objects.

6.4 Experiments

The experiments below validate STARK as an integrated strongly coupled simulator, where rigid bodies, deformables, joints, motors, contact, and friction interact through a single variational solve, rather than testing individual physical models in isolation. The towel experiment is therefore central to the thesis: it exercises the full coupling pipeline and demonstrates that the assembled components produce physically faithful results. In this section, we first validate our simulator by reproducing real-world laboratory experiments of a mobile vacuum cleaner robot interacting with a towel on the floor in different configurations. Then, to demonstrate the

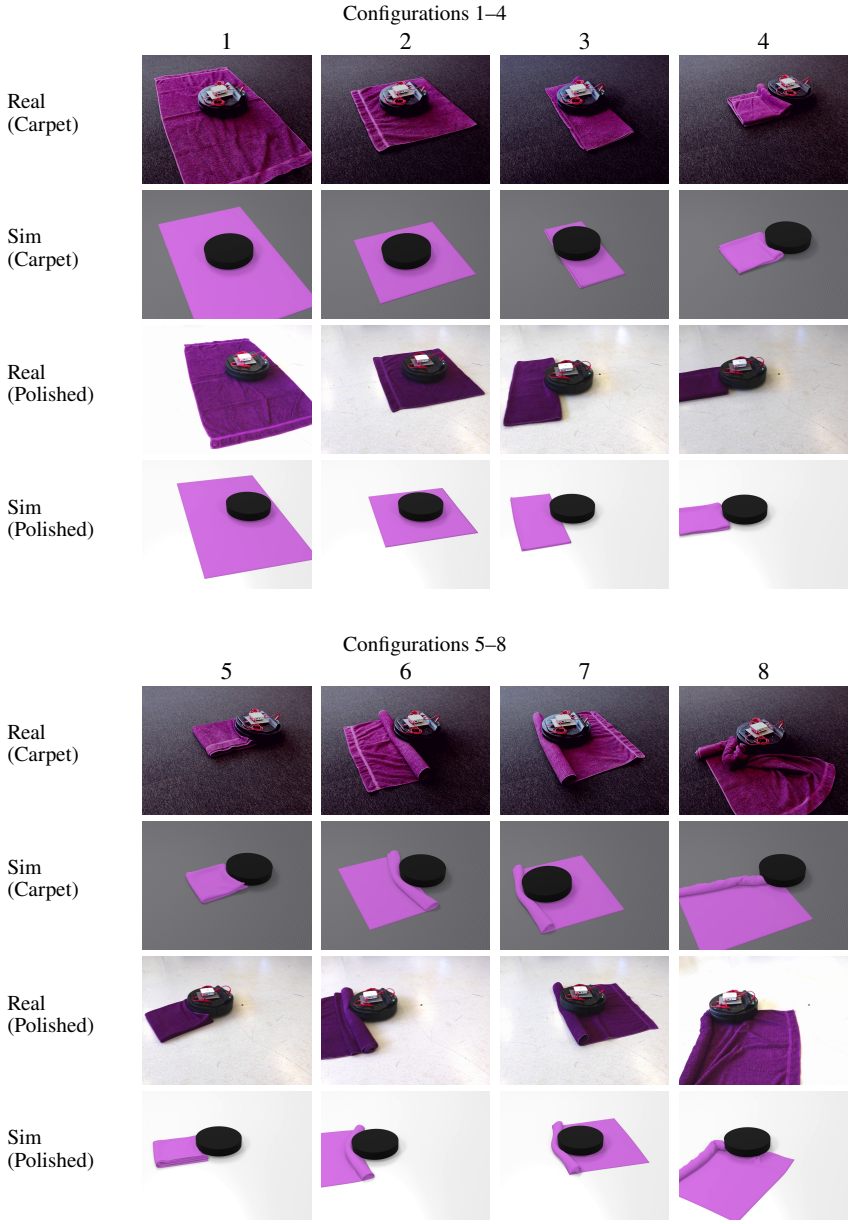


Figure 6.3: *Real and simulated outcomes of the 16 runs in the robot-towel interaction experiment. The columns correspond to the configurations in Table 6.1. Simulation results are visualized with Blender's Eevee renderer.*

general purpose nature of STARK, we show a simulation of a Franka robotic gripper grasping a deformable plastic cup.

6.4.1 Validation of Robot-Cloth Interactions

We validate our simulator in a robot-cloth interaction simulation, a challenging yet relevant use-case driven by the vision of domestic robots that can better deal with everyday household environments.

We run 16 different variations of a vacuum robot driving towards a towel on the floor, in eight towel configurations and two different floor types: carpet (high friction) and polished (low friction). See Figure 6.3 for each variation in simulation and the lab experiment. The goal is to evaluate whether the simulation can reproduce the outcome of each experiment, categorized as the robot pushing the towel (P), driving over the towel (O) or getting stuck (S). The results for all the cases are shown in Table 6.1.

In both, real-world experiments and simulations, we notice how the type of floor is a crucial factor that influences the outcome. While the robot cannot push the towel on the carpet floor in any instance due to the higher friction, it is able to push it in four occasions on the polished floor (cases 3 to 7). The robot can drive over the towel on the polished floor when the towel is more extended (cases 1 and 2) due to larger contact area with the floor and lower stacked thickness. Deformations also play a central role as they influence the robot's pose, lifting it from the front (cases 4 to 8 for both floor types) or providing uneven and changing resistance (case 8 for the polished floor). This has a direct effect on the wheel positions and how much contact pressure there is, which relates to the available traction via friction. This, combined with the motor's power delivery, determines the robot's reaction and ultimately the outcome and how much energy is required during the interaction. This complex chain of interactions highlights the motivation to use strong coupling with tight residuals. Otherwise errors due to low tolerances quickly accumulate, severely reducing confidence in the simulation results. Despite the complexity of the models employed and their intricate interactions, STARK can reproduce the outcome of the experiments in 100% of the cases, for a fixed parametrization of the floors, towel and robot across all simulations.

Here we list the most relevant parameters for the simulations (see the experiment code in the repository for more details). We model the robot as specified in the open-source Kobuki repository [Yujin Robot 2013]. The body is approximated by a cylinder with radius 17.5 cm and height 7 cm although a better fit mesh is used for collision. The powered wheels have a radius of 3.5 cm, while the radius of the non-powered front wheels is 1.35 cm. The total mass of the robot is 2.951 kg and the clearance to the floor is 1.35 cm.

The robot is put together using joints provided by STARK. The free-spinning front and back wheels are attached to the body using hinge joints. A motor is used to attach each powered wheel to a suspension arm, which is in turn attached to the body by a damped spring. The motors deliver a maximum torque of 66.6 mN m (as in spec) and a maximum angular velocity of 7.43 rad s^{-1} to match the peak robot velocity of 0.26 m s^{-1}.

The strain of the towel is modeled using FEM with the Neo-Hookean constitutive model and linear Lagrangian triangular elements. We use $E = 1.54 \times 10^4$ Pa for the Young's modulus and $\nu = 0.3$ for Poisson's ratio. Additionally as cloth is almost inextensible, we employ edge-length based strain limiting to give increased resistance to overstretches beyond 10%. We use a bending stiffness ratio of $k_b = 5 \times 10^{-6}$. Without damping, the simulation would result in an almost perfectly elastic piece of soft rubber, far from the real-world cloth behavior



Figure 6.4: *Simulation of a Franka Emika Hand grasping a plastic cup with weights. Results visualized with Blender's Cycles path tracer.*

which is very energy dissipative. We use Rayleigh damping with an inertial damping factor of 2 and a 0.1 ratio relative to the stiffness for both the strain and bending damping. The total mass of the towel, which is lumped to the nodes, is 0.484 kg and its dimensions are 1.75 m by 0.75 m. These parameters are set based on hands-on experimentation with the real towel and are transferable to other simulations using such towel material.

The thickness of the towel is 3.25 mm, which we conveniently use for the contact distance of $\hat{d}$. We use a friction coefficient of $\mu_t = 0.4$ for the towel self-contacts, $\mu_k = 1.0$ for the robot-towel and -floor contacts and $\mu_c = 1.0$, $\mu_p = 0.2$ for the towel-floor contacts, carpet and polished, respectively. The friction parameters are used as the degrees of freedom of this experimentation setup and fit to match the simulation outcomes to the real experiments. All parameters are kept constant for all the experiment instances. The total runtime for the 16 simulations is roughly 6 hours on a AMD Ryzen Threadripper PRO 5975WX CPU with 32 cores @ 3.60 GHz.

6.4.2 Grasping of a Plastic Cup

We also qualitatively showcase a grasping simulation to demonstrate that STARK can be used in other challenging scenarios, such as grasping of deformable objects (see Figure 6.4). Although not compared to real-world experiments, the simulation follows the expected results dictated by static friction analysis.

We model the parallel-jaw gripper Franka Emika Hand according to the manufacturer specifications [Franka Emika 2022]. We employ STARK's implicit joint controller to model the gripper with a target closing velocity of $100\,\mathrm{mm\,s^{-1}}$ and a maximum grip force of 5 N. The contact distance for this simulation is $\hat{d} = 0.5$ mm and the friction between the gripper and the cup is $\mu = 0.5$. The plastic cup, approximately 10 cm tall, uses the same shell/cloth models as the towel from the previous experiment but significantly stiffer in order to faithfully represent the behavior of a thin PET plastic sheet.

First, the jaws close to grab the plastic cup (Figure 6.4 left), which is then deformed. Once the gripper and the deformed cup come to an equilibrium, steel balls of 14 g each are poured into the cup. Static grasping breaks when 20 balls are inside the cup (Figure 6.4 center), as the vertical pull reaches 2.8 N, surpassing the theoretical 2.5 N static friction threshold given by the 5 N contact pressure and the $\mu = 0.5$ Coulomb's friction. Due to the large deformations of the plastic shell and the reinforcement at the cup's lips, the cup is held for longer as it latches onto the jaws (Figure 6.4 right). It takes 60 minutes to compute the 16 seconds of simulation on the aforementioned hardware, mostly due to the simulation length and amount of collisions between spheres. The simulation of the grasp itself, which happens in the first 2 seconds of the simulation, takes 5 minutes. This type of simulations, which are relevant to a wide range of robotic grasping tasks such as bin picking, are notoriously difficult since they require accurate and robust methods for both materials and contacts, and good coupling between the deformable and rigid body systems.

6.5 Conclusions

In this chapter we presented STARK, a framework to simulate strongly coupled mechanical systems, including rigid and deformables, with frictional contact. The goal of our system is to incorporate recent advances in simulation to provide not only a larger set of supported phenomena than the most used frameworks currently available, but also a higher degree of modeling accuracy and robustness. We demonstrate the capabilities of STARK in simulations featuring non-linear materials undergoing large deformations, rigid body systems with implicit joints and motors, and complex contact configurations with friction. STARK simulations are able to match the outcome of all 16 simulations carried out in a lab setting where a vacuum cleaning robot interacts with a towel on the floor. Available simulation solutions lacked the required features to model such simulations or did not have the required accuracy, especially when handling frictional contacts. STARK is therefore posed to fill a gap in the current landscape of available simulation software and empower researchers with easy access to state-of-the-art simulation solutions.

Nevertheless, there are paths for improvements within STARK. First, introducing additional mechanical effects, like plasticity for instance, is an interesting direction. Second, although the present optimizer implementation has shown its effectiveness in the provided examples, refining its performance would be beneficial as it is the current bottleneck of the simulation execution. Finally, adopting standard scene description files or a higher level API would further bolster its adoption.

Within the thesis, STARK serves as the integrated system counterpart to the SymX infrastructure chapter (Chapter 5). It demonstrates how the mathematical and modelling Foundations, along with the SymX engine, can support a full strongly coupled simulator capable of reproducing very complex real-world materials and interactions.

6.6 Impact and Adoption

Since its publication, STARK has continued to serve as a simulation platform for strongly coupled rigid and deformable systems. The framework has been extended with additional models, a more flexible C++ and Python interface, and performance improvements from later work, including the Hessian-stabilization techniques discussed in Chapter 7. These

developments preserve the original role of STARK: to provide a reusable implementation of optimization-based simulation with rigid bodies, deformable objects, constraints, contact, and friction in a single coupled solve.

Beyond the examples presented in this chapter, STARK has also been used as the simulation backend for following subsequent research projects:

- *Micropolar Elasticity in Physically-Based Animation* [Löschner et al. 2023], which introduces richer continuum models with additional rotational degrees of freedom.
- *Strongly Coupled Simulation of Magnetic Rigid Bodies* [Westhofen et al. 2024], which extends the energy-based modeling workflow to magnetic interactions between rigid bodies.
- *Curved Three-Director Cosserat Shells with Strong Coupling* [Löschner et al. 2024], which formulates strongly coupled shell energies with director fields.
- *Progressively Projected Newton's Method* [Fernández-Fernández et al. 2026], discussed in Chapter 7, which improves the Hessian-stabilization strategy used in Newton-based simulation systems.

In these works, the physical models differ substantially, but the simulations are all carried out within the same STARK framework.

CHAPTER 7

Progressively Projected Newton's Method

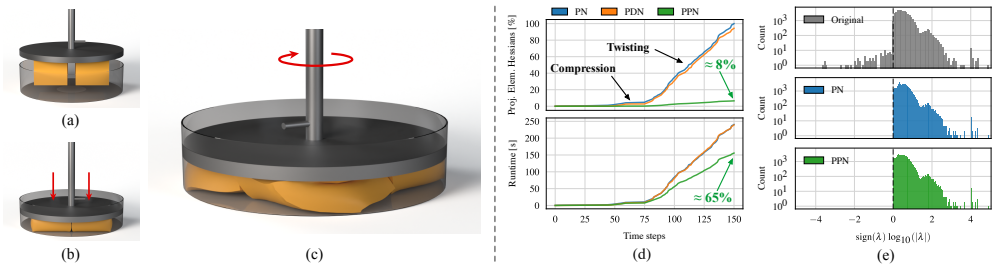


Figure 7.1: ***Press.*** *(a) Four elastic orange boxes drop into a rigid cylinder; (b) a press descends; (c) the press then rotates, forcing the boxes to roll under pressure due to friction. (d)* Top*: cumulative count of projected element Hessians, highlighting that PPN projects only 8 % of all elements;* bottom*: cumulative runtime highlighting that PPN completes the simulation in 65 % of the time required by PN and PDN. (e) Global eigenvalue histogram at time step 125, iteration 4: both PN and PPN handle negative eigenvalues as expected, though not equally; while PN projects all the element Hessians at this iteration, PPN only projects 17.4 % of them.*

This chapter presents Progressively Projected Newton's Method (PPN), a Newton-type method for nonlinear optimization-based simulation that improves Hessian stabilization by applying projection selectively. The chapter is adapted from the original PPN publication [Fernández-Fernández et al. 2026]. The implementation is open source and available in the SymX repository[1].

In Chapter 2, we introduced Newton-type minimization, Hessian definiteness, line search, and the positive-definite projection of local Hessians (Section 2.5). SymX (Chapter 5) and STARK (Chapter 6) provide the practical context in which local Hessians are generated, filtered, assembled, and used for optimization. Here we improve that pipeline by projecting

[1] https://github.com/InteractiveComputerGraphics/symx

only the local Hessians needed to recover a descent direction, reducing both the number of unnecessary projections and the distortion imposed on the global system.

7.1 Introduction

Robust and efficient simulation of dynamic deformable and rigid bodies is a cornerstone of computer graphics, visual effects, and interactive design. Many modern simulators perform time stepping by reformulating implicit integration schemes as successive nonlinear optimization problems, each solved with a variant of Newton's method. Foundations Chapter 2 covered this optimization-based framework in detail: the incremental potential formulation, Newton's method, line search, and the stabilization of indefinite Hessians through local positive-definite projection (Section 2.5). Chapters 5 and 6 provide the practical simulation infrastructure and application context in which local Hessians are symbolically generated, assembled, and stabilized for deformable and robotics-oriented simulation.

When the assembled Hessian of the potential energy is *Symmetric Positive Definite* (SPD), Newton steps are guaranteed to be descent directions, and efficient linear solvers specialized for SPD matrices can be applied, both highly sought-after properties. In practice, however, nonlinear materials, large time steps, and other factors frequently render the assembled Hessian indefinite, causing the line search to fail. As a consequence, practically every nonlinear second-order solver must implement a mechanism to remove indefiniteness from the system. This stabilization process therefore becomes a fundamental component of this type of solver, with a strong influence on robustness and performance.

A widely adopted remedy is *Projected Newton* (PN): an eigendecomposition is performed for every element Hessian and its negative eigenvalues are clamped (or mirrored) before assembly. Although PN produces an SPD system, it over-distorts the global Hessian and thus slows down convergence by unnecessarily discarding element-local negative curvature information. Moreover, projecting all elements imposes the cost of one eigendecomposition per element per Newton iteration, which can be expensive. The root cause of these drawbacks is the per-element *isolated* view on indefiniteness, which is overly limiting when analyzing the actual indefiniteness of the global system. While analytic eigenanalysis can alleviate performance concerns, it can be challenging to derive the required expressions when modeling complex effects or materials not yet established in the literature. This approach is also incompatible with simulation frameworks that use automatic or symbolic differentiation, such as SymX (Chapter 5), which rely on numerical eigendecompositions for flexibility.

We propose *Progressively Projected Newton* (PPN), a direct replacement for PN that avoids most element Hessian projections while ensuring descent directions during the Newton search. The key improvement of PPN over existing solutions is to identify sources of indefiniteness *after* the additive effect of assembly, rather than from an isolated, per-element perspective. Each Newton iteration begins with the unaltered Hessian, exactly as in pure Newton's method. If the linear solver exits early due to indefiniteness, PPN projects only those elements whose local residual exceeds an adaptive tolerance, updates the global matrix incrementally, and retries the solve. The tolerance is tightened until a descent direction is reached, then relaxed for the next iteration. PPN thus trades a few inexpensive solver attempts for the elimination of most eigendecompositions and a global Hessian closer to the unmodified one, which often leads to fewer Newton iterations.

PPN behaves exactly like Newton's method if no projections are required, and degenerates

to full PN only in very rare cases. By projecting only when and where necessary, PPN provides faster convergence than existing methods in dynamic simulations, while maintaining the robust guarantees of PN. In typical simulations, PPN omits more than 90% of element projections, cutting its cost by one order of magnitude, and reducing Newton iterations by up to 50% compared to PN. In our implementation, PPN achieves speedups of up to 2.5× over PN and up to 1.5× over the best alternative.

Crucially, PPN is formulated for dynamics, explicitly exploiting the regularization inherent in time integration, most notably the mass matrix. This allows it to avoid full-spectrum strategies by recognizing that sources of indefiniteness are highly localized in time and space, effectively rendering spectral verification unnecessary for the vast majority of elements.

In summary, the contributions of this chapter are:

- Progressively Projected Newton, a novel Newton-type solver that selectively projects element Hessians, drastically cutting eigendecomposition costs and avoiding unnecessary distortions to the global Hessian.
- A residual-driven heuristic that ranks elements by their likelihood of contributing negative curvature, which reuses information already calculated in the original Newton's method.
- A comprehensive evaluation on contact-free and contact-rich deformable bodies, shells, and rigid body systems.

7.2 Related Work

This discussion reflects the state of the art at the time the original PPN research was conducted [Fernández-Fernández et al. 2026]. We first connect PPN to optimization-based time integration as our main area of application. We then discuss eigenvalue filtering and briefly introduce automatic frameworks that use machine-generated derivatives as a prominent use case for progressive projections.

7.2.1 Optimization-based Time Integration

Optimization-based time integrators reformulate implicit schemes such as backward Euler as Incremental Potential minimization problems [Radovitzky and Ortiz 1999]. Many works in computer graphics adopted this approach [Kharevych et al. 2006; Martin et al. 2011; Liu et al. 2013], see Foundations Chapter 2 for a more in-detail review. In particular, second-order methods in this space require reliable treatment of indefinite Hessians [Gast et al. 2015; Li et al. 2020] in order to achieve robust optimization. Our proposed Newton-type solver is a drop-in replacement that can improve efficiency in these applications while retaining robustness.

7.2.2 Eigenvalue Filtering

To obtain descent directions during optimization, existing Newton-type solvers typically rely on the global Hessian being SPD. For most physical models, this is not always the case in practice. To address this issue, Teran et al. [2005] proposed per-element Hessian projection to the cone of SPD matrices by clamping their negative eigenvalues prior to assembly. This technique, commonly referred to as Projected Newton [Shtengel et al. 2017], avoids infeasible global

eigendecomposition and facilitates use of linear solvers specific to SPD matrices. The success of PN motivated a large body of work on efficient per-energy analytic eigenanalysis to avoid expensive numerical eigendecompositions [Smith et al. 2018, 2019; Kim 2020; Kim et al. 2019; Wang et al. 2023; Lin et al. 2022; Shi and Kim 2023; Wu and Kim 2023; Huang et al. 2024]. However, these analytic projections require careful manual modifications of the second-order derivative implementations. Further approaches for SPD projection include regularization using diagonal matrices [Fu and Liu 2016] or multiples of the mass matrix [Longva et al. 2026], but these are shown to perform worse than PN in contact-rich scenarios [Longva et al. 2026].

In the quasistatic setting with strong volume conservation and large initial deformations, eigenvalue *mirroring* [Chen et al. 2024c] and variants of *blending* [Cheng et al. 2025; Chen et al. 2024b], as opposed to clamping, have been shown to improve convergence. Unfortunately, as shown in Section 7.5, these results do not directly transfer to dynamic problems.

A comprehensive study by Longva et al. [2026] demonstrated that unconditional projection slows asymptotic convergence and breaks affine invariance. Their proposed *Project-on-Demand Newton* (PDN) method performs element projections only when the assembled matrix is detected to be indefinite, which typically occurs far from the solution, recovering Newton-like convergence as the iteration sequence progresses.

The aforementioned strategies share the limitation of acting on *all* elements and *in isolation*, ignoring the additive effect of neighboring contributions, and resulting in "over-projection". Our progressive strategy not only projects on-demand, but also selectively, significantly reducing both the amount of element projections and the distortion imposed on the global Hessian, while still guaranteeing descent directions.

7.2.3 Automatic Frameworks

Several frameworks in geometry processing and simulation automate components of second-order optimization. These systems rely on machine-generated derivatives and automated evaluation pipelines to tackle complex problems from concise symbolic or differentiable expressions [Schmidt et al. 2022; Herholz et al. 2024; Fernández-Fernández et al. 2025b]. In the context of this thesis, SymX (Chapter 5) provides exactly this layer: local energy contributions are described symbolically, differentiated automatically, and assembled into global energy, gradient, and Hessian evaluations. This makes methods such as PPN particularly natural, since the solver can operate directly on the same local Hessian blocks that are generated and assembled by the simulation infrastructure.

This automation has practical value beyond implementation convenience. It enables rapid and safer composition of solvers and models, and several of the works discussed above were developed using TinyAD [Schmidt et al. 2022] or SymX [Fernández-Fernández et al. 2025b]. However, positive-definite projection remains expensive in this setting. Because analytic or semi-analytic projection code is difficult to generate automatically, these frameworks typically rely on numerical eigendecompositions, which represent a significant cost in our measurements (Section 7.5). By avoiding most local projections, PPN reduces this bottleneck and narrows the performance gap between automatically generated simulation code and more hand-tuned implementations.

7.3 Projected Newton and Over-Projection

Newton's method, descent directions, and the positive-definite projection of local Hessians were introduced in Foundations Section 2.4 and Section 2.5. We briefly recall the key elements needed to motivate our method.

The simulation problem is posed as minimizing the total incremental potential $E(\mathbf{x})$, whose gradient is $\mathbf{g} = \nabla E$ and Hessian $\mathbf{H} = \nabla^2 E$. At each Newton iteration we solve $\mathbf{H}\,\Delta\mathbf{x} = -\mathbf{g}$. Both $\mathbf{g}$ and $\mathbf{H}$ are assembled from local element contributions. With the local selector $\mathbf{P}^e$, this can be written as $\mathbf{H} = \sum_e (\mathbf{P}^e)^T \mathbf{H}^e \mathbf{P}^e$ and $\mathbf{g} = \sum_e (\mathbf{P}^e)^T \mathbf{g}^e$. A non-zero Newton step $\Delta\mathbf{x}$ is guaranteed to be a descent direction if $\mathbf{H}$ is SPD [Nocedal and Wright 2006].

In practice, however, $\mathbf{H}$ is often indefinite (see, e.g., [Kim and Eberle 2022]). Projected Newton (PN) remedies this by clamping the negative eigenvalues of each element Hessian before assembly. Consider the eigendecomposition of the Hessian of element e,

$$\mathbf{H}^e = \mathbf{Q}^e\,\mathbf{D}^e\left(\mathbf{Q}^e\right)^T, \tag{7.1}$$

where the columns of $\mathbf{Q}^e$ are the eigenvectors of $\mathbf{H}^e$ and $\mathbf{D}^e$ is a diagonal matrix of the corresponding eigenvalues. Applying clamping, the projected element Hessian is $\widehat{\mathbf{H}}^e = \mathbf{Q}^e\,\widehat{\mathbf{D}}^e\left(\mathbf{Q}^e\right)^T$, with $\widehat{D}^e_{ii} = \max(D^e_{ii}, \varepsilon)$ for $\varepsilon > 0$. Since a sum of SPD matrices is SPD, the assembled global matrix is guaranteed to be SPD.

However, projecting all element Hessians is often unnecessary. Consider element matrices A and B, and the global matrix P:

$$A = \begin{pmatrix} -1 & 0 \\ 0 & 2 \end{pmatrix}, \quad B = \begin{pmatrix} 2 & 0 \\ 0 & 2 \end{pmatrix}, \quad P = A + B = \begin{pmatrix} 1 & 0 \\ 0 & 4 \end{pmatrix}.$$

A is indefinite with eigenvalues $\{-1, 2\}$, B is SPD with eigenvalues $\{2, 2\}$, yet the assembled P is SPD with eigenvalues $\{1, 4\}$. Projecting A would alter the global matrix unnecessarily and likely deteriorate convergence of Newton's method, as shown by Longva et al. [2026]. Project-on-Demand Newton (PDN) addresses this by projecting all element Hessians only when the global matrix is detected to be indefinite, generally improving convergence over PN. However, PDN still projects *every* element Hessian when indefiniteness is encountered, which leads to the over-projection that PPN aims to reduce.

7.4 Progressively Projected Newton's Method

In this section we introduce our method, PPN, starting with a motivating example. Consider a third element matrix added to the previous ones:

$$C = \begin{pmatrix} -10 & 0 \\ 0 & 1 \end{pmatrix}, \quad Q = A + B + C = \begin{pmatrix} -9 & 0 \\ 0 & 5 \end{pmatrix}.$$

In this case, Q is indefinite with eigenvalues $\{-9, 5\}$. However, only projecting C suffices to obtain an SPD approximation, demonstrating that full projection, even when conditionally triggered as PDN does, is still unnecessary. This *element-selective projection* is the core idea of PPN: the global Hessian $\mathbf{H}$ is built from two disjoint sets of element Hessians, the projected set $\mathcal{H}_p$ and the unprojected set $\mathcal{H}_u$. The goal then is to keep $|\mathcal{H}_p|$ minimal while ensuring that the assembled Hessian yields a descent direction at all times during Newton iterations.

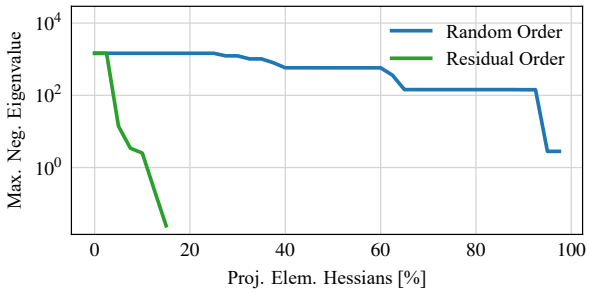


Figure 7.2: *Largest global negative eigenvalue as element Hessians are progressively projected for the iteration shown in Figure 7.1 (d) and (e). In blue, randomized element order. In green, elements ordered by the local assembled residual from largest to smallest.*

In practice, however, identifying which element Hessians to project is not as straightforward as in the example above. Assembly scatters element contributions, making global eigenvalue recovery intractable, and building $\mathcal{H}_p$ based on exact global eigenvalues would be prohibitively expensive. Instead, we propose a cheap post-assembly proxy for the *likelihood* that an element is the source of global indefiniteness. We pair this prediction with a robust global indefiniteness check to adapt $\mathcal{H}_p$ as needed. Both the predictor and the indefiniteness check derive efficiently from information already available during the Newton minimization process.

Projection heuristic We motivate a residual-based heuristic to rank elements by their likelihood of contributing to global indefiniteness. Let $\mathbf{x}^\star$ be a strict local minimizer with $\mathbf{g}(\mathbf{x}^\star) = \mathbf{0}$ and $E(\mathbf{x}^\star+\epsilon) > E(\mathbf{x}^\star)$ for small ϵ. At a local minimizer, $\mathbf{H}(\mathbf{x}^\star)$ is SPD (Section 2.5), and by continuity it remains (semi)definite in a neighborhood of $\mathbf{x}^\star$. Hence, as $\mathbf{x}\to\mathbf{x}^\star$, $\mathbf{g}\to\mathbf{0}$ and indefiniteness in $\mathbf{H}$ diminishes. However, convergence does not occur uniformly across the domain. We observed that elements in regions with lower assembled residual are less likely to induce indefiniteness in the global system, even if left unprojected. Figure 7.2 empirically supports this hypothesis: while a random ordering requires the projection of almost all elements to recover global positive definiteness, projecting in order of larger assembled residual first isolates the indefinite contributions rapidly. Elements in low-residual regions are more likely to either already have all-positive eigenvalues or their minor indefinite spectra cannot overturn stronger SPD contributions, such as the influence of the mass matrix. As demonstrated in Section 7.5, this correlation is robust and consistent across a diverse set of scenarios, material models, and discretization parameters.

Motivated by this, we prioritize projecting element Hessians in regions with larger assembled residual and expand as necessary. Specifically, we introduce a projection tolerance δ and add $\mathbf{H}^e$ to $\mathcal{H}_p$ when the element-local residual of e is larger than δ, that is, $\|\mathbf{S}^e\mathbf{g}\|_\infty > \delta$, where $\mathbf{S}^e$ is a residual-based selection operator that extracts the assembled gradient entries affected by element e (distinct from the local DoF selector $\mathbf{P}^e$ used in Foundations Section 2.3). The ∞-norm captures the largest component of the local residual, acting as a selector of all elements incident to nodes with large residuals, aligning with the concept of domain partitioning. The gradient, or residual, relates directly to stationarity where the system is known to be SPD, while other progress metrics, such as the Newton step or iteration decrements, depend on the optimization procedure. The tolerance δ is adapted over the course of the Newton iterations

by a tightening factor $\alpha \in (0, 1)$ and a release factor $\beta \geq 1$. The former is applied when indefiniteness is detected in the global Hessian, and the latter after a successful step is taken. As a result, we obtain an effective partition using already calculated values at no extra cost. This parametrization exhibits low sensitivity, yielding consistent performance gains across broad ranges of values. Values of $\alpha = 0.5$ and $\beta = 2$ are chosen as safe defaults by ablation tests in Section 7.5. To guarantee termination and bound the worst-case runtime, we fall back to PN and project all elements if δ drops below a small threshold $\varepsilon_\delta = 10^{-12}$.

Algorithm 3: Progressively Projected Newton (PPN)

```
1  δ ← ∞
2  while not converged do
3      Assemble unprojected H and g
4      (Δx, ind) ← SolveSPD(H, −g)
5      while ind do
6          if δ = ∞ then
7              δ ← α ‖g‖_∞
8          foreach e do
9              if (‖S^e g‖_∞ > δ) or (δ ≤ ε_δ) then
10                 ProjectToPD(H, e)                ▷ In-place update
11         (Δx, ind) ← SolveSPD(H, −g)
12         if ind then
13             δ ← α δ                              ▷ Tighten projection
14     δ ← β δ                                      ▷ Relax projection
15     γ ← LineSearch(Δx)
16     x ← x + γ Δx
```

Algorithm Our method is outlined in Algorithm 3. The standard Newton's method logic with an SPD linear system solver is unmodified except for the inner PPN projection logic (lines 5 to 13). Note that the linear solver must report whether indefiniteness was encountered in addition to the solution $\Delta\mathbf{x}$. The first time projection is needed, δ is initialized in relation to the largest absolute value of the current residual (lines 6 and 7). If the system still cannot be solved, δ is reduced (lines 12 and 13). Once the linear solve is successful, δ is increased for the next Newton iteration (line 14). Eigenvalue clamping [Teran et al. 2005] is used for the projection of element Hessians.

7.4.1 Implementation

PPN integrates naturally into existing PN pipelines, requiring only two operations to be done efficiently: incremental global Hessian updates, and an SPD linear solver that exits early upon indefiniteness.

Incremental Hessian updates When element e moves from $\mathcal{H}_u$ to $\mathcal{H}_p$ the update can be done by assembling the difference

$$\Delta\mathbf{H}^e = \widehat{\mathbf{H}}^e - \mathbf{H}^e \tag{7.2}$$

into the global matrix in place, which can reuse existing assembly routines. Importantly, this operation does not change the sparsity of $\mathbf{H}$, which should make updates faster than the original assembly.

Linear Solvers Our implementation considers Preconditioned Conjugate Gradient (PCG) and Cholesky factorization solves (LLT), which can both exit early on indefiniteness for significantly lower cost than the total cost of the linear solve.

The numerical factorization of LLT exits on the first negative pivot encountered. Since the expensive symbolic analysis can be reused as long as the sparsity pattern does not change, failing is amortized with the eventual successful solve.

In the case of PCG, we monitor its intermediate value $\mathbf{d}^T\mathbf{H}\mathbf{d}$ for each CG search direction $\mathbf{d}$. If a direction of negative curvature is encountered, indefiniteness is confirmed and the linear solver is stopped. Thus, when equipped with (P)CG, PPN draws parallels with the "Newton-CG" method: instead of using the last valid intermediate solution of CG as the Newton step (which might be very inaccurate), we restart the CG solve with more projections applied. Even if we do not eliminate all indefiniteness from the global Hessian, as long as CG does not encounter a negative search direction, the resulting intermediate solution is guaranteed to be a descent direction as in Newton-CG [Nocedal and Wright 2006]. In our experiments, warm-starting subsequent PCG calls with the last descent direction yielded worse results than simply starting every solve with the zero vector, hence we use the latter approach.

Multithreading While our method adds a few additional steps, it does not introduce thread load imbalance and remains friendly to multithreading: the unprojected Hessian is built (all threads, unrelated to PPN), the linear solver is run (all threads, also unrelated to PPN), and, if indefiniteness is detected, a selection of elements is projected in place (all threads). Section 7.5 demonstrates our method's performance on a high-core-count workstation CPU.

7.5 Results

In this section, we present a comprehensive suite of experiments to compare PPN with PN and PDN across a variety of simulations. All experiments are implemented in the same STARK simulation ecosystem introduced in Chapter 6, which provides the contact handling, rigid body dynamics, and linear solver infrastructure used throughout. We begin by describing the hardware, software, and models used, followed by an ablation study on PPN's parameters.

7.5.1 Experimental Setup

Hardware and software. All experiments are conducted on a workstation equipped with a 3.60 GHz AMD Ryzen Threadripper PRO 5975WX processor (32 cores, 64 threads) and 256 GB of RAM. We use STARK's built-in 3×3 Blocked Diagonal PCG solver and Intel MKL 2025 for Cholesky factorization. Eigen 3.4 handles all other linear algebra operations, including eigendecompositions, which we measure to be on average 1.53× faster than MKL's ones for matrices of size 15×15 and smaller. We use gcc 12.2 for compilation.

Time stepping and tolerances. We use the backward Euler scheme for time stepping and, unless otherwise stated, a time step size of $\Delta t = 1/30$ s. As stopping tolerance for Newton's

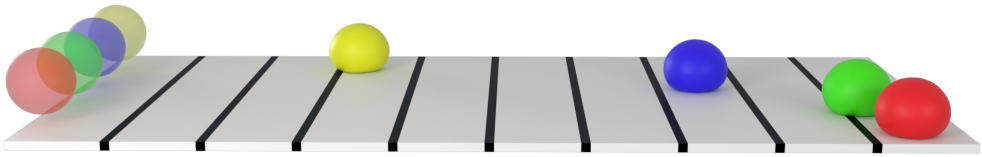


Figure 7.3: ***Rolling sphere.*** *A deformable sphere with initial horizontal velocity rolls on a flat surface. Different Newton step velocity tolerances are used:* $10^{-1}\,\mathrm{m\,s^{-1}}$ *(yellow),* $10^{-2}\,\mathrm{m\,s^{-1}}$ *(blue),* $10^{-3}\,\mathrm{m\,s^{-1}}$ *(green),* $10^{-4}\,\mathrm{m\,s^{-1}}$ *(red).*

method we check if the velocity step infinity-norm $\Delta t^{-1}\|\Delta\mathbf{x}\|_\infty$ falls below $10^{-3}\,\mathrm{m\,s^{-1}}$. The choice of tolerance greatly influences the number of Newton iterations. The experiment shown in Figure 7.3 justifies our choice: a tolerance of $10^{-2}\,\mathrm{m\,s^{-1}}$ or coarser causes outcome-altering energy losses, while tolerances of $10^{-3}\,\mathrm{m\,s^{-1}}$ and $10^{-4}\,\mathrm{m\,s^{-1}}$ lie much closer. For consistent comparisons and to avoid bias from inexactness, we verify convergence across solvers with a final fully projected Hessian solve (as PN would) via LLT factorization. Because such a validation solve is not typically performed in production, we exclude its cost from all timing measurements. PCG uses a relative residual tolerance of $\|\mathbf{r}\|\|\mathbf{r}^0\|^{-1} = 10^{-4}$. Element eigenvalues are clamped to $\varepsilon = 10^{-8}$. For PDN, we adopt the countdown of 4 suggested in the original paper, which also yielded the best results in our "Press" benchmark.

Boundary conditions and materials. Dirichlet constraints are enforced using penalty potentials. All elastic solids employ the Neo-Hookean material in 2D and the Stable Neo-Hookean [Smith et al. 2018] model in 3D. Frictional contact uses the IPC [Li et al. 2020] potentials with backtracking line search for sufficient descent and intersection-based filtering. We use the rigid body inertial potential introduced by Macklin et al. [2020]. A list of material parameters, mesh sizes, and time step settings is provided in Table 7.1.

7.5.2 Experiments

Ablation We systematically analyze the sensitivity of PPN to the tightening (α) and release (β) rates. Figures 7.4 and 7.5 map the performance landscape for the "Press" (Figure 7.1) and "Armadillo Slingshot" (Figure 7.11) scenes, covering both iterative (PCG) and direct (LLT) linear solvers. These scenes provide complementary stress-tests: "Press" is dominated by compression and friction-induced indefiniteness, whereas "Armadillo Slingshot" characterizes a regime of extreme tensile strains without contact. In the aggressive limit ($\alpha = 0.9, \beta \to \infty$), the method successfully minimizes projection counts ($< 7\%$ and $< 2\%$, respectively), but total runtime degrades due to the overhead of recovering from failed indefinite linear solves. Conversely, conservative settings ($\alpha = 0.01, \beta = 1.0$), result in excessive projection and increased Newton iterations, which is also slow. We adopt a power-of-two scaling ($\alpha = 0.5$ and $\beta = 2.0$) as our standard parametrization, owing to its optimal or near-optimal performance. The method exhibits low sensitivity around this optimum, allowing us to apply these parameters across all scenes without individual fine-tuning.

Next, we compare PPN with PN and PDN, using both PCG and LLT, in Figure 7.6. We observe that PDN largely resorts to PN, revealing that only a few steps encountered zero global indefiniteness. Even under these stress conditions, the adaptive nature of PPN avoids over 92% of element projections and reduces Newton iterations by 14%. Runtime gains with

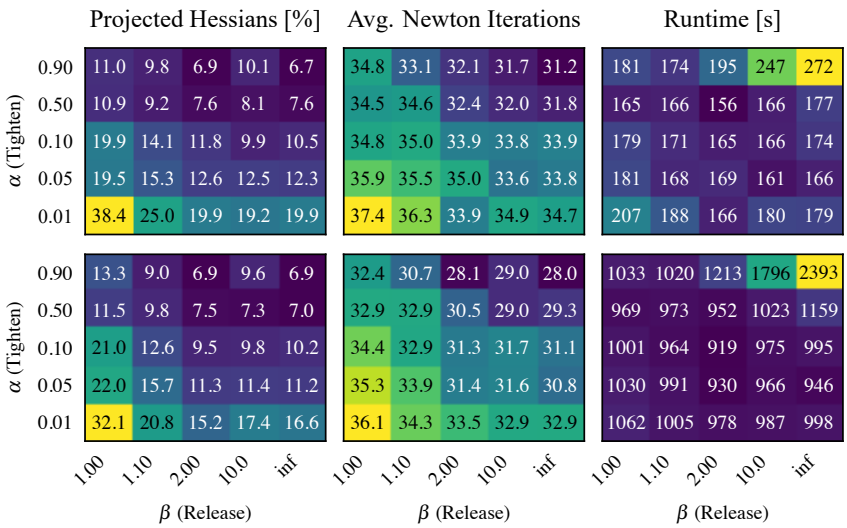


Figure 7.4: *Ablation test for the parametrization of PPN in the Press scene using the iterative PCG linear solver (top) and the direct LLT solver (bottom). Color scale is independent per table.*

LLT are modest (5.2%), as the direct solver dominates total cost, but with PCG, we observe a speedup of 1.5× compared to PN and PDN. See Figure 7.1 (d) for a visualization of these reductions over the time steps, and Figure 7.1 (e) for the effect on negative eigenvalues of the three methods. All following experiments use PCG as the linear solver.

Eigenvalue mirroring [Chen et al. 2024c] was originally introduced specifically for quasistatic problems where strong indefiniteness is not counteracted by e.g. the mass matrix. For completeness, we apply mirroring to the "Press" experiment for all three solvers, however, results are poor in this elastodynamics contact simulation: Mirroring consistently performs worse than clamping, needing an average of 51, 52, and 45 Newton iterations for PN, PDN, and PPN respectively, an increase of about 40% across all solvers. Based on this result, we apply clamping for all further dynamic experiments if not otherwise specified.

Resolution, Time Step Size and Tolerance Figure 7.7 compares all solvers on the "Press" scene across different resolutions (2k, 15k, 108k degrees of freedom), time step sizes (100, 10, 1 ms) and tolerances (10^{-2}, 10^{-3}, 10^{-4} m s^{-1}). PPN solves all instances by projecting only a fraction of the elements (between 30% and 5%), correlating positively with finer resolutions, smaller time steps, and tighter tolerances: Finer resolutions localize sources of indefiniteness more effectively, smaller time steps magnify the regularizing effect of the mass matrix, and stricter tolerances extend the Newton iteration sequence with steps where most of the domain has locally converged. PPN also achieves consistently fewer Newton iterations and lower runtimes, with the exception of the very large time step of $\Delta t = 100$ ms, where both PDN and PPN struggle. Nevertheless, PPN performs strongly, completing the entire benchmark using 27.3% fewer Newton iterations than PN and offering a speedup of 1.93×. In contrast, PDN takes 7.7% fewer Newton iterations and is just 7% faster when compared to PN.

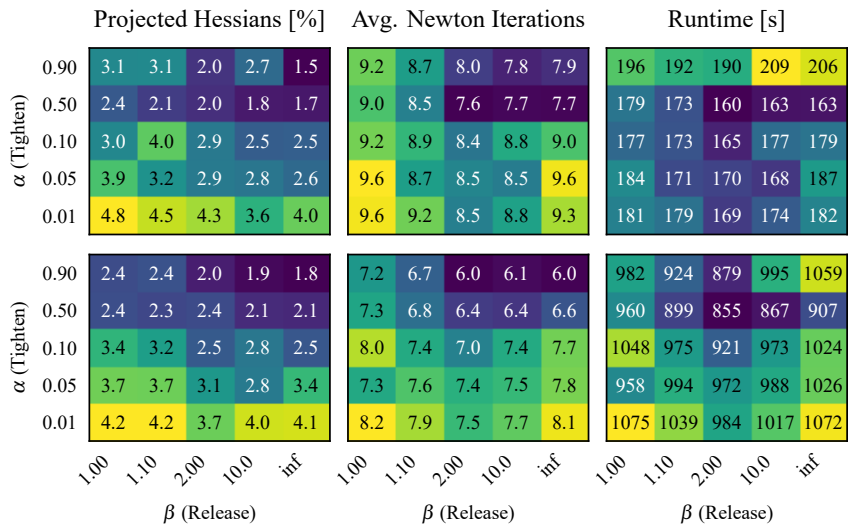


Figure 7.5: *Ablation test for the parametrization of PPN in the Armadillo Slingshot scene using the iterative PCG linear solver (top) and the direct LLT solver (bottom). Color scale is independent per table.*

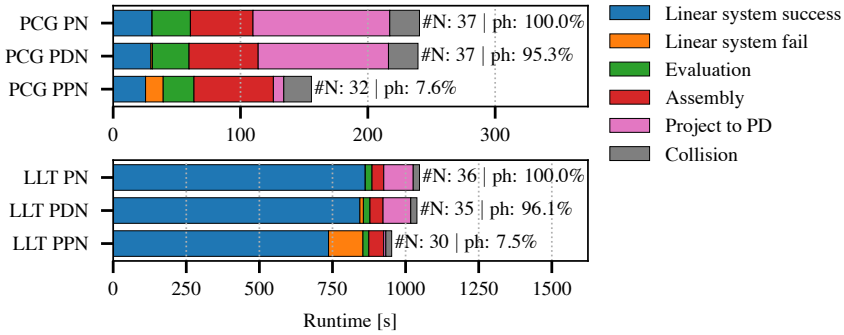


Figure 7.6: *Runtime breakdown for the chosen parametrization of PPN in the "Press" scene using PCG (top) and LLT (bottom) linear solvers. The average number of Newton iterations (*#N*) and the percentage of projected element Hessians (*ph*) are shown at the right of each bar.*

Quasistatic Simulation We compare the three solvers using eigenvalue clamping and mirroring on a quasistatic problem involving a large initial deformation for various resolutions and Poisson ratios (Figure 7.8). We use a Newton step stopping criteria of 0.1% of the domain's size. In line with Chen et al. [2024c], we reproduce the positive outcomes of eigenvalue mirroring for such scenarios while clamping produces artifacts. As suggested by the previous experiment, PDN and PPN face challenges in this inertia-free setting, indicating that unconditional projection might be preferred in this setting.

Large Ratios We compare all three solvers in simulations featuring large stiffness ($E = 10^6$–10^{10} Pa) and density ($\rho = 10^1$–10^4 kg m^{-3}) ratios in Figure 7.9. Here, PPN consistently yields both the fewest Newton iterations and the fastest runtimes. While on average PPN projects

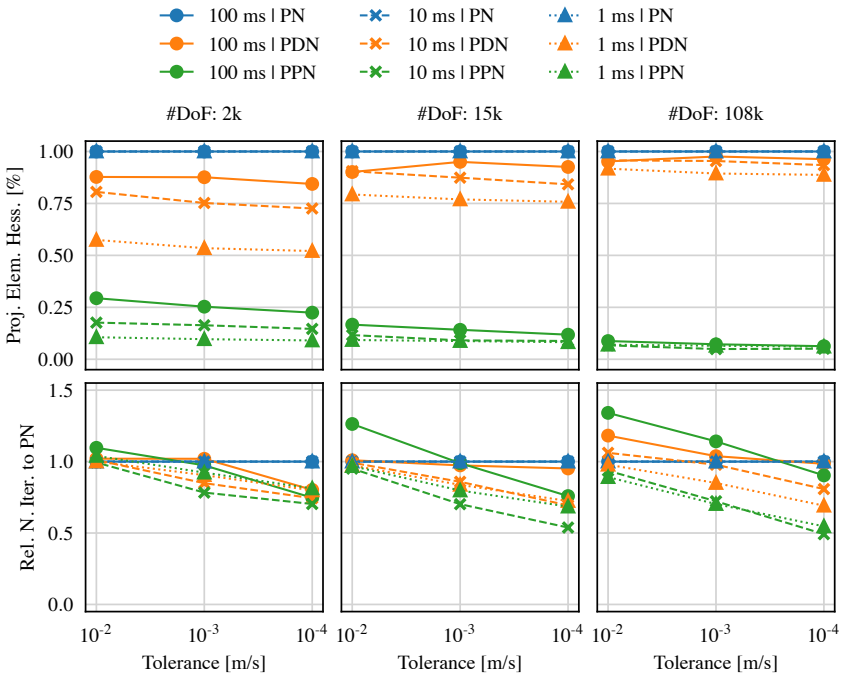


Figure 7.7: *Number of projected Hessians (top) and Newton iterations (bottom) in relation to PN for all solvers in different parametrizations of the "Press" scene: mesh resolution (columns), time step size (marker shape), solver type (color) and tolerance (x-axis).*

6.2% of the element Hessians, PDN projects 88.4%. On average, PPN requires only 35.6% and 54.5% of the Newton iterations of PN and PDN, respectively, corresponding to speedups of 2.02× and 1.42×. These findings suggest that the residual-based heuristic in PPN remains robust even when adjacent elements exhibit curvature variations spanning several orders of magnitude.

Finally, we quantify geometrical reproducibility across the three methods using a simulation characterized by large strains and extreme density ratios ($\rho = 10^1$–$10^4\,\mathrm{kg\,m^{-3}}$), as shown in Figure 7.10. The mean area-weighted surface distance over the full time sequence is just 0.34 mm (0.057% of the domain size), with a maximum deviation of 1.04%. This confirms that the choice of projection strategy does not compromise the fidelity of the outcome, yet PPN delivers speedups of 1.39× and 1.35× over PN and PDN, respectively.

Contact-free The three solvers are compared in a contact-free simulation of an elastic armadillo in Figure 7.11. Even in this simpler setting, PDN ends up projecting more than 50% of all the element Hessians, while PPN only needs to project less than 3%. PPN reduces the Newton iterations by 53% and 20% with respect to PN and PDN, demonstrating that PPN's effectiveness is not exclusive to scenarios with complex frictional contact. Corresponding speedups are 2.5× and 1.34×.

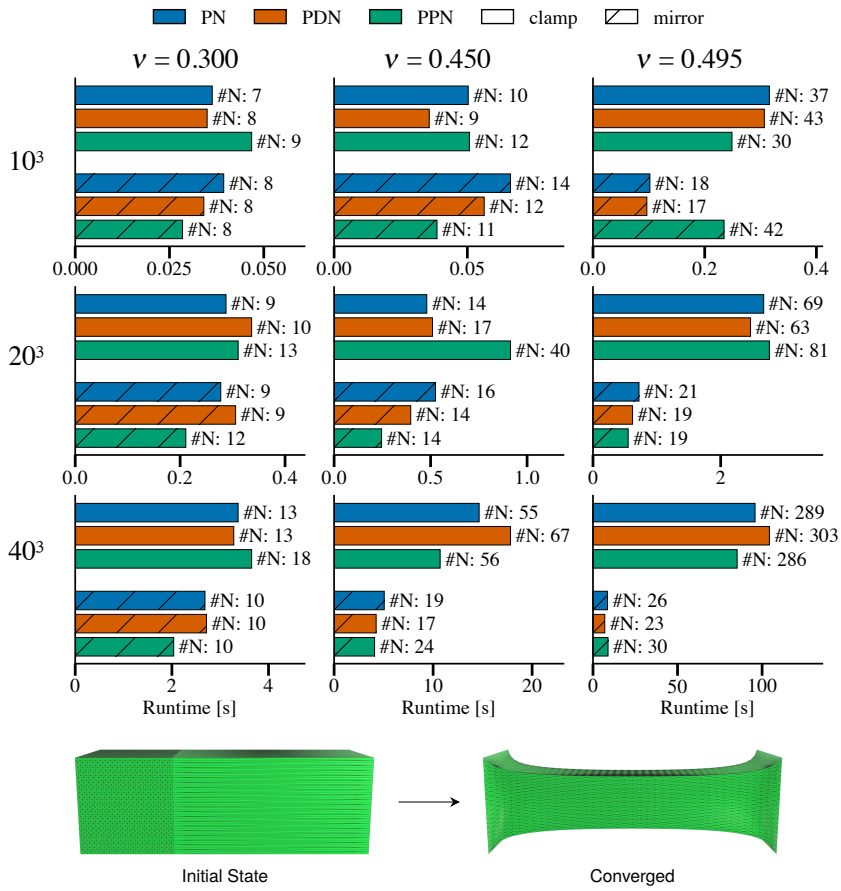


Figure 7.8: ***Quasistatic extrusion***. *An elastic box is stretched by twice its size in a quasistatic setting using different resolutions (rows) and Poisson's ratios (columns). Solvers use different colors, and eigenvalue filtering different shading: solid for clamping and hatched for mirroring. The number of Newton iterations is shown at the right of each bar.*

Codimensional We compare the three solvers in a contact-rich cloth simulation using three resolutions with vertex counts of 64^2, 128^2 and 256^2 in Figure 7.12. In this challenging scenario, PPN provides significant improvements for the coarsest mesh: a reduction of 45.8% and 30.7% Newton iterations, and 55.5% and 41.7% of runtime in relation to PN and PDN, respectively. For the finest discretization, PPN does not reduce iterations but still achieves more than 17% performance improvement over the alternatives.

Impact-rich We test two scenes featuring high-energy impacts in Figure 7.13 and 7.14. The former simulates elastic objects and the latter rigid bodies. To preserve vividness, these simulations use a time step of 1/300 s, as larger time steps resulted in visibly damped dynamics. Although PPN still greatly reduces the number of projections, its advantage over PDN in terms of Newton iterations is more modest. Nevertheless, runtime was reduced by 26.2% and 12.6% in relation to PN and PDN for the elastic scene, and by 86.5% and 10.4% for the rigid body

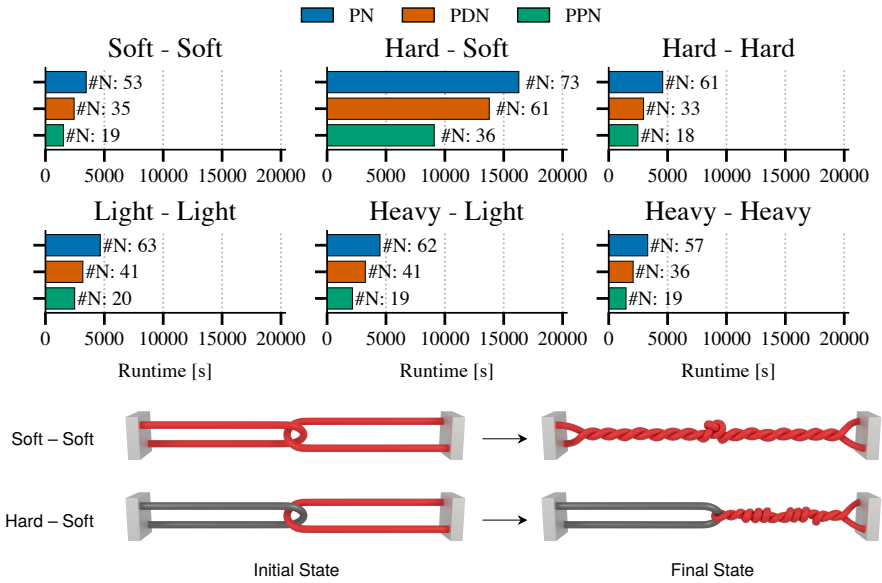


Figure 7.9: ***U-Turn***. *Two elastic cylinders are interlaced and twisted. Above, we compare solvers under varying Young's moduli (top) and densities (bottom). Below, the initial and final states are shown. All simulations except the "Hard-Soft" produce the same deformation.*

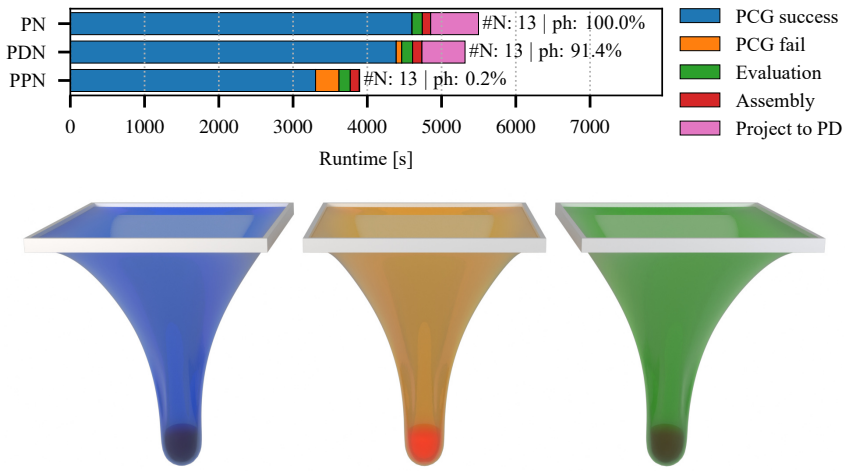


Figure 7.10: ***Jelly Catch***. *A heavy elastic ball falls on a thin sheet of very light elastic material clamped on its perimeter, resulting in very large strain in the material. Top: Comparison between the solvers. Bottom: Time step of maximum strain for PN, PDN and PPN, respectively from left to right.*

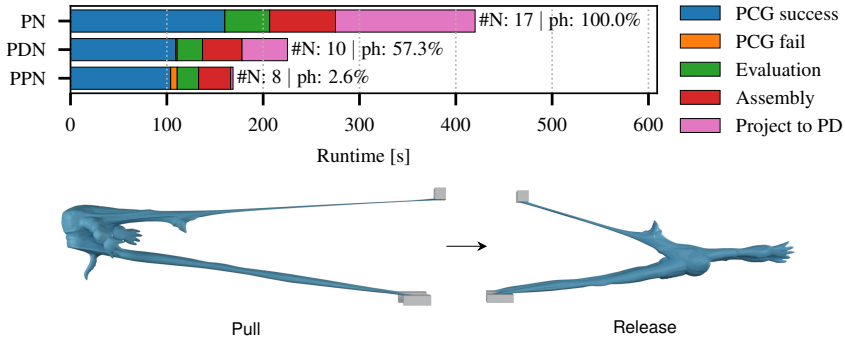


Figure 7.11: ***Armadillo slingshot****. An elastic armadillo is pulled and then released. Comparison between solvers on top, largest deformation states below.*

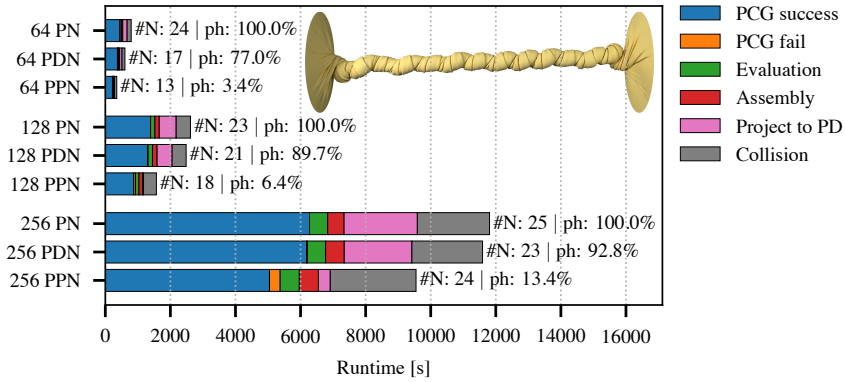


Figure 7.12: ***Twisting cloth****. A cloth cylinder is twisted by rotating its ends in opposite directions using three mesh resolutions. Final configuration of the finest resolution shown inside the plot.*

one. Notably, PN struggles significantly in the rigid body scene, requiring more than five times as many iterations and representing an outlier in our tests.

7.6 Limitations and Future Work

While PPN reliably reduces element projections and Newton iterations in any simulator, the net speedup can vary depending on solver choices (e.g., direct vs. iterative) and execution platforms (e.g., CPU vs. GPU). Codebases that incorporate analytic projections may see lower gains than those relying on numerical eigendecompositions, and applications using a fixed number of Newton iterations may only benefit from the reduced projections.

Our results also indicate that current on-demand projection strategies, including PDN and PPN, perform poorly in scenarios lacking strong mass matrix contributions (or alternative forms of regularization), as in very large time steps or in the quasistatic limit. However, the time steps where this becomes a problem are rare in practice due to the associated numerical damping and loss of detail. In any case, for realistic choices of time step sizes, including relatively large ones such as $1/30$ s PPN consistently excels.

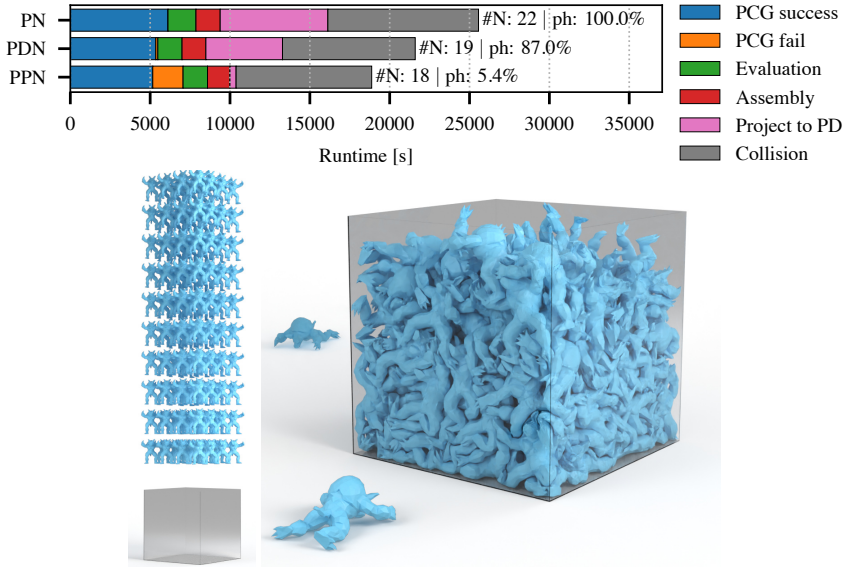


Figure 7.13: ***Armadillo drop***. *160 elastic armadillos are dropped into a rigid box. Comparison between solvers on top. Initial and final state below.*

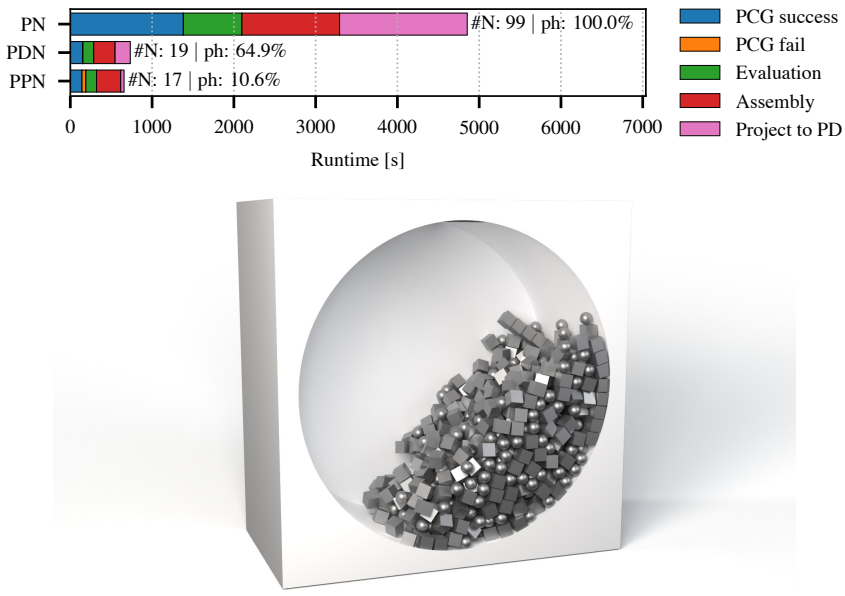


Figure 7.14: ***Tumbler***. *More than 1000 rigid bodies collide inside a spinning tumbler. Comparison between solvers on top. Collision runtime is omitted for clarity, as it dominates total cost. An intermediate state is shown below.*

Table 7.1: *Scene parameters. All simulations use* 0.5 mm *IPC contact distance. Cloth is* 1 mm *thick. Densities (ρ) are given in* kg m^{-3}*, except for the cloth example, where surface density is given in* kg m^{-2}.

Scene	n_{dof}	Δt [s]	**Dim.** [m]	**Duration** [s]	E [Pa], ν	ρ
Rolling sphere	3 K	1/30	0.15	12	10^3, 0.49	1000
Press	47 K	1/30	0.30	5	10^5, 0.40	1000
Quasistatic extrusion	variable	∞	0.50	–	10^8, 0.49	–
U-Turn	87 K	1/30	1.85	12	variable, 0.49	variable
Jelly Catch	570 K	1/30	0.50	10	variable, 0.49	variable
Armadillo slingshot	71 K	1/30	1.00	15	10^5, 0.40	1000
Twisting cloth	variable	1/30	0.50	20	10^5, 0.30	0.20
Armadillo drop	566 K	1/300	0.35	7.5	10^4, 0.45	1000
Tumbler	7 K	1/300	0.50	15	rigid	1000

7.7 Conclusion

We introduced Progressively Projected Newton's Method, a direct replacement for Projected Newton that guarantees descent directions while reducing element projections by an order of magnitude and improving convergence. PPN begins each Newton iteration with the unmodified Hessian and only projects elements incrementally when the linear solver detects indefiniteness, guided by a residual-driven tolerance that adapts across iterations.

Extensive experiments on dynamic simulations of deformable solids, shells, frictional contact, and rigid bodies demonstrate that PPN consistently performs 90% fewer projections, decreases the number of Newton iterations by up to 50% compared to PN, and achieves total simulation speedups of up to 2.5× relative to PN and up to 1.5× over PDN. Hence, PPN stands out as the most efficient method among the three for robustly handling indefiniteness in second-order optimization time integration.

Within the thesis, PPN refines the Newton and Hessian-stabilization layer introduced in Foundations Chapter 2. It complements SymX (Chapter 5) and STARK (Chapter 6) by improving how assembled second-order systems are solved.

CHAPTER 8

Interactive Facial Animation via Differentiable Shell Simulation

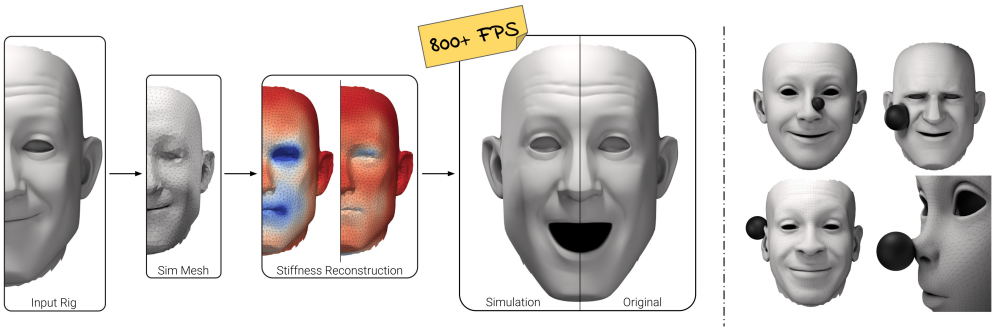


Figure 8.1: *Main stages of the Interactive Facial Animation pipeline and example contact interactions on four different characters. From left to right: The input is a face rig containing a surface mesh of the face and a set of facial expressions. A shell simulation mesh is obtained by decimating the input surface to a user-defined target number of triangles. Quasistatic inverse simulation is used to find suitable non-uniform bending and strain stiffness so that the forward shell simulation can reproduce the original expressions within a strict small tolerance. Using this stiffness, the upsampled forward shell simulation is able to closely reproduce the original rig output while enabling dynamic effects and user-driven contact interactions.*

This chapter presents an application contribution of this thesis in the field of facial animation and interaction for VR applications, adapted from the original Interactive Facial Animation (IFA) publication [Fernández-Fernández et al. 2025a]. The goal is to augment existing facial rigs with contact-aware physical simulation without replacing the original rig as the source of identity, expression, and visual fidelity. In this setting, simulation is not used to rebuild a face from anatomical assumptions, but to add interaction behavior to a rig that may already be production-ready.

For inverse simulation we use differentiable simulation and adjoint-based optimization, introduced in Chapter 4, to reconstruct non-uniform shell stiffness fields from rig expressions.

The implementation uses SymX (Chapter 5) for derivative generation in the adjoint method and STARK (Chapter 6) for quasistatic fitting solves. The final forward stage then uses the fitted model in a specialized real-time shell simulation pipeline for interactive facial animation.

8.1 Introduction

The transition toward highly realistic and interactive virtual experiences is evident in communication, entertainment, training, and other applications. Telepresence, in particular, has seen significant advances in the visual fidelity of virtual faces [Lombardi et al. 2018; Ma et al. 2021; Li et al. 2024a]. The next major milestone in immersion lies in enabling realistic user interactions through contact with these digital faces, whether for virtual character engagement in a video game or touching one's own face during a call. Without properly addressing these physical interactions, such experiences remain confined to *"look-but-don't-touch"*, thus limiting the sense of true engagement. However, existing methods that integrate contact physics with high-fidelity rigs often face one or more of the following limitations: (a) insufficient performance for strict real-time requirements, (b) limited reproducibility mechanisms that fail to maintain the rig's resolution and expressiveness, or (c) dependence on artist intervention or per-identity anatomical priors (e.g., skull or mandible systems).

Much of these experiences run on affordable consumer hardware. Moreover, face simulation is just one component of a broader real-time system that includes rendering, networking, audio, and more. With this resource-constrained environment in mind, our method aims to provide a contact simulation solution for facial animation that can operate at a rate of one millisecond per physics simulation update, ensuring compatibility with high refresh rates such as those often found in VR. Beyond runtime efficiency, setup time is also critical; while users may tolerate a few minutes the first time they set up a character, multi-hour waiting periods should be avoided.

In addition to performance considerations, the preservation of the resolution and expressiveness of high-quality rigs during simulation remains a priority. Many of the existing simulation solutions address contact by replacing the original rig geometry with simulation-ready models, e.g. using anatomically aware human head templates. Such replacements generally do not feature rigorous mechanisms for fidelity preservation and may also perform poorly with non-human characters [Yang et al. 2024].

Our method, Interactive Facial Animation, is a novel real-time facial animation pipeline that augments existing face rigs with shell mechanics and contact, preserves their expressiveness, and requires no anatomical priors. The pipeline is fully automatic and does not require artist intervention, enabling us to scale the method and to seamlessly integrate it into existing pipelines. While our approach is "hands-free", it is still possible to adjust the behavior of the physical simulation, when desired, by tailoring parameters such as stiffness, density, or damping, to specific identity or scenes. However, we demonstrate that a single parameter set works well across multiple characters and resolutions.

Our design decisions are guided by strict real-time performance requirements. The only inputs to our method are a face mesh with the desired render resolution, a set of facial expressions representative of the rig's expressive range, and simulation parameters. In this chapter, we use classical blendshape rigs to conveniently provide both the mesh and expressions in an industry-standard format.

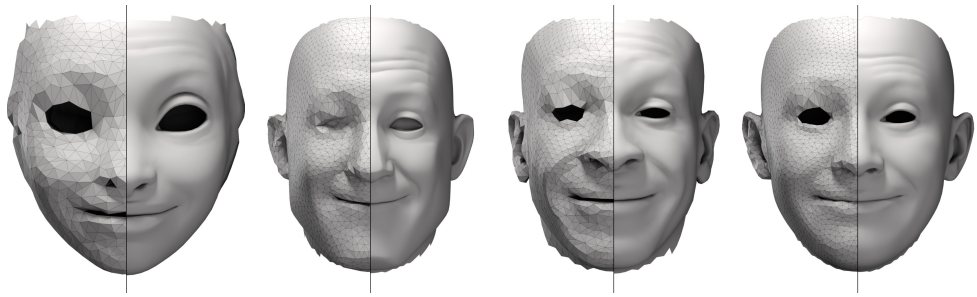


Figure 8.2: *Split screen of the simulation shell mesh (left) and original input mesh (right) for four different identities and resolutions. From left to right, the number of simulation triangles and character names are: 2k (Aura), 6k (Bowen), 4k (Jupiter) and 8k (Proteus).*

At the core of our method is a surface-only shell simulation that tracks the input rig through actuation forces while providing passive elasticity and contact handling. This two-dimensional discretization offers a compact representation of the face geometry, enabling efficient simulation of strain and bending directly on the skin surface. Furthermore, in order to maintain a bounded runtime, we operate on a decimated version of the input face mesh with a user-specified number of triangles (see Figure 8.2). For a given target expression and potential objects in contact, the simulation calculates an equilibrium for the actuation, passive elasticity, and contact on the coarse mesh. We then warp the full-resolution mesh from neutral pose with the resulting displacements and add back fine details lost during decimation. Consequently, the output of our pipeline retains the rig's original fidelity while enabling contact-aware effects.

A key challenge is ensuring that the simulation can reproduce the entire range of input facial expressions since only certain combinations of actuation and passive shell stiffness can achieve these targets while responding realistically to contact. To address this, we formulate an inverse simulation problem with an upper limit on actuation forces and a strict reproducibility tolerance for the simulation to obtain the input facial expressions. Solving for non-uniform distributions of strain and bending stiffness yields a material parametrization of the shell that can be safely used in the forward simulation pipeline, ensuring reproducibility. Note that while the forward simulation incorporates dynamics, the inverse problem is solved in the quasistatic regime, which reduces the computational requirements to obtain a material parametrization. Finally, having a known actuation limit allows us to cap external contact forces accordingly, preventing unnatural large deformations during user interactions.

The contributions of this chapter are:

- A novel inverse quasistatic simulation procedure to recover non-uniform surface stiffness distributions from facial expression geometry, imposing strong reproducibility guarantees.
- A novel forward simulation pipeline that incorporates dynamics, shell mechanics, and contact into face rigs, with update times of around one millisecond on the CPU.
- A two-scale rig-plus-simulation coupling that preserves the behavior and fine-detail fidelity of the original rig while adding contact-aware deformation.

To the best of our knowledge, our method is the first to simultaneously fulfill these requirements regarding reproducibility, automation and performance.

Within the thesis, IFA combines the simulation and differentiation machinery in a concrete facial animation problem. The chapter combines inverse material reconstruction, shell simulation, contact handling, and real-time rig augmentation in one pipeline.

8.2 Related Work

This related-work discussion follows the context of the original IFA publication [Fernández-Fernández et al. 2025a]. It complements the broader background on shell mechanics, contact, differentiable simulation, and optimization introduced in Foundations (Chapters 2–4).

Facial animation is a long-established field spanning several decades. It encompasses numerous topics, including capture, retargeting, and animation, among others. In what follows, we first summarize physically-based face simulation approaches in the context of facial animation. We then discuss existing works that specifically address contact with external objects.

8.2.1 Face Simulation

Early work on face animation dates back to the 1970s [Parke 1972]. Physically-based face simulation followed in subsequent decades [Platt and Badler 1981; Waters 1987; Lee et al. 1995], leading to the seminal work by Sifakis et al. [2005], who introduced an identity-specific anatomical human head model that uses a nonlinear constitutive model with FEM and muscle-based actuation, which is fit from MRI data. Simulating digital faces with expression ranges depicted by blendshape rigs [Lewis et al. 2014] has become an active field of research, with the common goal to improve the realism via simulation by incorporating physically-based effects. Several artist-friendly pipelines have been presented [Cong et al. 2016; Kozlov et al. 2017; Choi et al. 2022] to give artists fine control to the facial expressions while still being fully simulated. Surface-only multi-scale approaches are proposed by Bickel et al. [2008, 2007] to simulate detailed face geometries from a rest pose and marker displacements using a coarse shell model for global deformation and an additional fine-detail solver for wrinkles. A surface-only force-based model is proposed by Barrielle et al. [2016] to add dynamics to blendshape rigs. Ichim et al. [2016] present a volumetric FEM model with an interior bone structure that is strain-actuated and that can incorporate dynamics and contact, later improved by adding explicit muscles, muscle sliding and other effects [Ichim et al. 2017]. Several works also focused on material property estimation from capture [Kozlov et al. 2019; Kadleček and Kavan 2019], with focus on realistic deformations on different regions of the face. Chandran and Zoss [2024] proposed an anatomical implicit face model that can reconstruct a dense anatomical substructure. Different methods have been introduced for real-time simulation of the face [Barrielle and Stoiber 2019; Kim et al. 2020]. A reduced actuation model [Romeo and Schvartzman 2020], a clustered muscle scheme [Romeo and Schvartzman 2020] and neural approaches [Srinivasan et al. 2021; Yang et al. 2022] have been proposed to address the shortcomings of representing actuation on dense meshes. Neural networks also have been used for expression and style disentanglement [Yang et al. 2023] and for generalized physical faces that can be quickly fit to unseen identities [Yang et al. 2024]. Neural physics approximations have been used in the context of low computation environments to achieve real-time simulation of anatomical face models [Wagner et al. 2022, 2023, 2025].

Contact-aware hand-face tracking has been addressed by incorporating collision resolution to captured data before training [Shimada et al. 2023; Wu et al. 2024]. Super-resolution

Table 8.1: *Qualitative comparison of existing contact-aware methods with respect to three key requirements: real-time capability on affordable consumer hardware, ability to reproduce the high-resolution input rig, and absence of anatomical priors or training on human-specific data.*

	Real-Time	Reproducibility	Prior-Free
Ours	Yes	Yes	Yes
Sifakis et al. [2005]	No	No	No
Ichim et al. [2017]	No	No	No
Shimada et al. [2023]	No	No	No
Wu et al. [2024]	No	No	No
Wagner et al. [2025]	Yes	No	No
Park et al. [2024]	No	No	No
Zhu and Joslin [2024]	Yes	No	No

strategies have also been explored to drive a high resolution anatomical model of a face by simulating a coarser counterpart [Park et al. 2024] at near real-time rates. Zhu and Joslin [2024] proposed a surface-only head template, featuring skull and deformable active and passive patches, that uses PBD to solve for actuation, inertia and contact and that can run in real time on consumer hardware.

8.2.2 Facial Contact Interaction

We now frame our proposed solution in the context of existing methods that showcase contact interactions. It is important to emphasize that employing a physically-based face model (anatomical or otherwise) is not a guarantee by itself to produce realistic behavior under external contacts or collisions. For example, some of the aforementioned methods use physically-based constraints to deform a mesh in order to match a series of targets; however, this goal presents a very different proposition from that of contact. Specifically, a face can be unrealistically too soft or too stiff upon contact, but still be able to correctly deform into a target pose given some actuation mechanisms. In lieu of this, we consider methods that handle contact to be those which show actual contact with external objects in the reference material. We provide a high level comparison in Table 8.1.

When tasked with producing a sequence of contact interactions with an input rig, no method guarantees preservation of the input fidelity in full resolution since they do not feature strict reproducibility mechanisms. Methods that use head templates do not provide procedures to recover geometric detail lost in the transfer from input rig to simulation rig [Sifakis et al. 2005; Ichim et al. 2017; Wagner et al. 2025; Zhu and Joslin 2024]. Methods that rely on external tracking pipelines are constrained by these pipeline expression outputs, including resolution [Shimada et al. 2023; Wu et al. 2024]. All methods require anatomical priors either because they fit a head template (simulation methods) or because they train on human data (tracking methods) and therefore they can only support such anatomies. Finally, only two of the methods can run in real time on affordable consumer hardware [Wagner et al. 2025; Zhu and Joslin 2024]. We then establish that, to the best of our knowledge, there is no solution available in the literature that satisfies our requirements of performance, reproducibility and not requiring anatomical priors.

8.3 Forward Simulation Pipeline

The forward formulation follows the optimization-based simulation viewpoint introduced in Chapter 2, specialized here to the shell model and runtime constraints of IFA. The shell and contact modeling choices also build on the variational models discussed in Chapter 3, but the implementation deliberately favors a constant-Hessian model and simple contact corrections to meet the real-time budget of interactive facial animation.

IFA employs a coarse shell simulation that uses actuation forces to track the input rig, incorporates passive strain and bending elasticity to model deformations, and handles contact. Given a target expression, an external object in contact with the face, and a material stiffness distribution, the simulation produces contact-aware skin deformation for that facial expression. A key feature of our solution is the use of an *actuation pressure field* with an upper limit. This enables high enough contact forces to deform the shell away from the target facial expression and thus yield realistic skin-level deformations. The simulation is differentiable, a requirement of the stiffness reconstruction procedure discussed in the next section.

For the remainder of this chapter, we assume that the face rig to be enhanced is represented by a blendshape rig [Lewis et al. 2014], although this is not a strict requirement of the proposed concept. Any rigging approach able to provide a set of representative facial expressions for the material estimation stage, and a sequence of targets during runtime for the forward pipeline is compatible with our method. A blendshape rig produces a face mesh with a target expression $\mathbf{x}$ via linear interpolation of a set of deformation modes (or *blendshapes*) as

$$\mathbf{x} = \mathbf{X} + \sum_{b=1}^{n_b} \mathbf{u}_b \, w_b, \tag{8.1}$$

where n_b is the number of blendshapes, $\{\mathbf{u}\}$ is the set of blendshapes and $\{w\}$ are corresponding interpolation weights that define the expression. Note that this is a purely geometric approach without the involvement of physics.

With this context, IFA's forward simulation pipeline proceeds in three stages:

1. A coarse version of the rig uses the expression input (blendshape weights $\mathbf{w}$) to generate a coarse surface target expression.

2. Actuation forces drive the simulation shell toward this target and the dynamic simulation is solved, including actuation, passive elasticity and contact.

3. The original resolution face mesh in neutral pose is warped using the resulting coarse displacement field, followed by a fine-detail correction step to recover small-scale features. If fine-detail reproducibility is not necessary, the latter step can be skipped for extra performance gains.

IFA can be used with any shell solver, provided (a) it offers convergence guarantees to ensure reproducibility and (b) it is differentiable so that we can perform stiffness reconstruction via inverse simulation. Our implementation uses an efficient differentiable linear elasticity solver with constant second-order derivatives, based on the Fast Mass-Springs [Liu et al. 2013] method. This elasticity solver is highly efficient thanks to a precomputed Cholesky factorization of the global stiffness matrix, a well-known technique in simulation in computer graphics [Liu et al. 2013; Bouaziz et al. 2014; Overby et al. 2017; Kugelstadt et al. 2018].

More sophisticated combinations of discretization, constitutive models, and solvers could be employed, e.g., Neo-Hookean elasticity on a triangulated surface solved with Projective Dynamics [Bouaziz et al. 2014]. Though such an approach might yield improved material response compared to a spring-based model, its higher per-element cost due to material evaluation and other discretization considerations, such as per-element rotation extraction, can noticeably reduce the framerate despite remaining real-time. Given our emphasis on performance, we present our pipeline using the spring-based approach.

Finally, our method can be categorized as a two-scale solution, where the main simulation runs at a coarse resolution and finer details are recovered afterward. While this design has parallels to existing multi-scale methods [Bickel et al. 2007, 2008], note that our fine-detail correction aims to geometrically match the input data, rather than introduce new features via simulation or other means.

8.3.1 Shell Simulation

Our dynamic system is governed by the equations of motion

$$\mathbf{M}\dot{\mathbf{v}} = \mathbf{f}(\mathbf{x}) = -\nabla_{\mathbf{x}}\phi(\mathbf{x}), \quad \dot{\mathbf{x}} = \mathbf{v}, \tag{8.2}$$

where $\mathbf{x}$ is the vector of node positions, $\mathbf{v}$ the velocity, $\mathbf{M}$ the lumped mass matrix (with surface density ρ), and $\phi(\mathbf{x})$ the assembled simulation potential. We employ an optimization-based time-integration scheme to evolve the positions of the simulation mesh [Martin et al. 2011]. Each time step Δt is advanced via backward Euler by minimizing

$$E(\mathbf{x}) = \frac{1}{2\,\Delta t^2}\left\|\mathbf{M}^{\frac{1}{2}}(\mathbf{x} - \tilde{\mathbf{x}})\right\|_2^2 + \phi(\mathbf{x}) \tag{8.3}$$

with respect to $\mathbf{x}$, given $\mathbf{v} = (\mathbf{x} - \mathbf{x}^{\text{prev}})/\Delta t$, where $\tilde{\mathbf{x}} = \mathbf{x}^{\text{prev}} + \Delta t\,\mathbf{v}^{\text{prev}}$ is the inertial estimate from the current state $(\mathbf{x}^{\text{prev}}, \mathbf{v}^{\text{prev}})$. Following the notation of Section 2.3, E is the global incremental potential assembled from local contributions. For IFA, the non-inertial part is decomposed as

$$\phi = \phi_{\text{act}} + \phi_{\text{strain}} + \phi_{\text{bend}} + \phi_{\text{damp}}, \tag{8.4}$$

where ϕ_{act} is the actuation potential, ϕ_{strain} accounts for in-plane strain (mass-springs), ϕ_{bend} handles thin-shell bending, and ϕ_{damp} represents the lagged damping contribution.

Actuation. We incorporate actuation forces to guide the mesh toward user-specified facial expressions via

$$\phi_{\text{act}}(\mathbf{x}) = \sum_{i=1}^{n_v} \phi_{\text{act}}^i(\mathbf{x}_i) = \sum_{i=1}^{n_v} \frac{1}{2}\frac{p^{\max} A_i}{\epsilon^r}\left\|\bar{\mathbf{x}}_i^t - \mathbf{x}_i\right\|_2^2, \tag{8.5}$$

where n_v is the number of vertices, $\bar{\mathbf{x}}_i^t$ is the (clamped) target position of vertex i, $\mathbf{x}_i$ its simulated position, $p^{\max}$ the maximum actuation pressure, A_i the Voronoi area of the vertex in the rest pose, and ϵ^r the reproducibility tolerance. The term $\frac{p^{\max} A_i}{\epsilon^r}$ sets the per-vertex actuation stiffness, corresponding to a maximum force $p^{\max} A_i$ at a distance ϵ^r. To ensure that the actuation force never exceeds this bound while maintaining a constant global Hessian, we do *not* directly use the vertices $\mathbf{x}^t$ from the coarse rig. Instead, for each component $j \in \{1, \ldots, 3n_v\}$, we clamp:

$$\bar{\mathbf{x}}_j^t = \mathbf{x}_j + \max\left(-\epsilon^r, \min\left(\mathbf{x}_j^t - \mathbf{x}_j, \epsilon^r\right)\right), \tag{8.6}$$

thus pulling the shell toward $\mathbf{x}^t$ without exceeding a pressure field of $p^{\max}$. As a result, actuation forces remain significant within ϵ^r but crucially, do not grow unbounded when large deviations occur due to contact, which would result in unnaturally large forces, e.g. when bending an ear. Since this bound is only required for deformations beyond ϵ^r, it is not necessary during material reconstruction and therefore does not affect differentiability.

Strain. We model in-plane elasticity via springs using the edge-based potential

$$\phi_{\text{strain}}(\mathbf{x}) = \sum_{i=1}^{n_e} \phi^i_{\text{strain}}(\mathbf{x}^i) = \sum_{i=1}^{n_e} \frac{1}{2} l_i^r \kappa_i^s \left\| \mathbf{p}_{i1} - \mathbf{p}_{i2} - \mathbf{d}_i \right\|_2^2, \tag{8.7}$$

for edge i connecting points $\mathbf{p}_{i1}$ and $\mathbf{p}_{i2}$, n_e the number of edges, l_i^r the edge's rest length and κ_i^s its strain stiffness. $\mathbf{d}_i$ is the target distance vector

$$\mathbf{d}_i = \left\| \mathbf{p}_{i1}^r - \mathbf{p}_{i2}^r \right\|_2 \frac{\hat{\mathbf{p}}_{i1} - \hat{\mathbf{p}}_{i2}}{\left\| \hat{\mathbf{p}}_{i1} - \hat{\mathbf{p}}_{i2} \right\|_2}, \tag{8.8}$$

where $\mathbf{p}^r$ are rest positions, and $\hat{\mathbf{p}}$ are per-time-step-constant (or *lagged*) positions.

Bending. We add a bending potential ϕ_{bend} for each internal edge (or dihedral) following Volino and Magnenat-Thalmann [2006]

$$\phi_{\text{bend}}(\mathbf{x}) = \sum_{i=1}^{n_d} \phi^i_{\text{bend}}(\mathbf{x}^i) = \sum_{i=1}^{n_d} \frac{1}{2} \frac{\|\mathbf{e}_i^0\|_2}{\bar{h}_i} \kappa_i^b \left\| \mathbf{q}_{i1} - \mathbf{q}_{i2} - \mathbf{d}_i \right\|_2^2, \tag{8.9}$$

where n_d is the number of dihedrals, $\mathbf{e}_i^0$ the common edge at rest pose, $\bar{h}_i$ is a third of the average of the heights of the two triangles incident to the common edge [Grinspun et al. 2003], and κ_i^b the bending stiffness. The points $\mathbf{q}_{i1}$ and $\mathbf{q}_{i2}$ are the closest points between the common edge and the line that connects the other two vertices of the dihedral, computed at rest pose.

Damping. We penalize nodal momentum to reduce spurious long-lasting oscillations using

$$\phi_{\text{damp}}(\mathbf{x}) = \frac{1}{2\,\Delta t} \nu \left\| \mathbf{M}^{\frac{1}{2}} (\mathbf{x} - \mathbf{x}^{\text{prev}}) \right\|_2^2, \tag{8.10}$$

where ν is a small damping coefficient.

Time Step Solve. Each time step requires solving

$$\mathbf{H}\,\Delta\mathbf{x} = -\mathbf{g}, \qquad \mathbf{g} = \nabla_{\mathbf{x}} E(\mathbf{x}), \qquad \mathbf{H} = \nabla_{\mathbf{x}}^2 E(\mathbf{x}), \tag{8.11}$$

where $\Delta\mathbf{x}$ is the displacement update. Because our chosen local potentials have constant second derivatives, the assembled global Hessian $\mathbf{H}$ is constant and hence is factorized via Cholesky before the simulation begins. At runtime, only the assembled gradient $\mathbf{g}$ must be updated, making each solve a straightforward forward/backward substitution. This *direct* solve guarantees stable and accurate results, suitable for the reproducibility requirement.

Time Sub-stepping. We split large time steps into sub-steps [Macklin et al. 2019] and perform exactly one elastic solve in each. For the per-time-step-constant positions, we use a blend between the current configuration and the explicitly integrated positions

$$\hat{\mathbf{x}} = \alpha^{\mathrm{e}}\,\tilde{\mathbf{x}} + (1 - \alpha^{\mathrm{e}})\,\mathbf{x}^{\mathrm{prev}}, \tag{8.12}$$

where $\alpha^{\mathrm{e}} = \frac{1}{2}$. For our application, this approach results in significantly better inertial behavior, i.e. less numerical damping, than performing multiple elastic iterations on a large time step size, at the same computational cost.

Constraints. We incorporate contact and maximum depth constraints into our forward simulation solver. To avoid refactoring the global matrix when new constraints become active, we do not incorporate constraints into the global elastic solve; instead, we opt for direct positional corrections [Müller et al. 2007]. This approximation works well in practice for our application and suits our strict computational budget.

Contacts are detected using analytic Signed Distance Fields (SDF), but discretized fields can also be used [Koschier et al. 2017]. Maximum depth constraints limit how far in the negative normal direction each vertex can deviate from the target, which approximates a bone substructure. After each elastic solve, vertices of the simulation mesh are projected to the closest feasible point from their invalid position. An approximated correction force is calculated based on the acceleration required to perform such correction, which can be used for further actions, such as controlling colliding object dynamics. Finally, velocities are determined considering the initial and final (corrected) vertex positions and the time step size.

8.3.2 Warping and Fine-Detail Correction

After each simulation step, we obtain a *coarse displacement field* $\mathbf{u} = \mathbf{x} - \mathbf{X}$, where $\mathbf{x}$ are the simulated node positions and $\mathbf{X}$ their rest state. To recover high-resolution output, we *warp* the original rig face mesh $\mathbf{X}_o$ by interpolating $\mathbf{u}$ onto the fine mesh, thereby transferring simulated contact deformations while retaining the rig's overall resolution. However, the coarse simulation mesh cannot capture details finer than its resolution, therefore some high-frequency features contained in the blendshapes (e.g., fine wrinkles) may be lost. Figure 8.3 illustrates this issue for a frowning expression, where subtle creases in the nose bridge are missing in the warped result.

Since the target high-resolution geometry $\mathbf{x}_o^t$ is known from the original rig, we can precompute a *detail correction* $\mathbf{u}_c$ that, when added back after warping, reintroduces the transient small-scale features in the final surface. In blendshape rigs, we define a correction $\mathbf{u}_{c,b}$ for each blendshape b:

$$\mathbf{u}_{c,b} = \mathbf{x}_{o,b} - \mathbf{x}_{w,b}, \tag{8.13}$$

where $\mathbf{x}_{o,b}$ is the original blendshape geometry, and $\mathbf{x}_{w,b}$ the corresponding simulated and warped version. Clipping away very small corrections, e.g. below 10^{-4} m, results in sparse distributions localized around features like folds, wrinkles, or around the eyelids (see Figure 8.4). Note that $\{\mathbf{u}_c\}$ are *not* rotation invariant, which is acceptable because these details are already defined in the target configuration and do not require transformation from the neutral pose. While this assumption does not strictly hold during interactions, we found that incorporating such invariant did not offer a notable visual improvement, but it did add a significant computational cost.

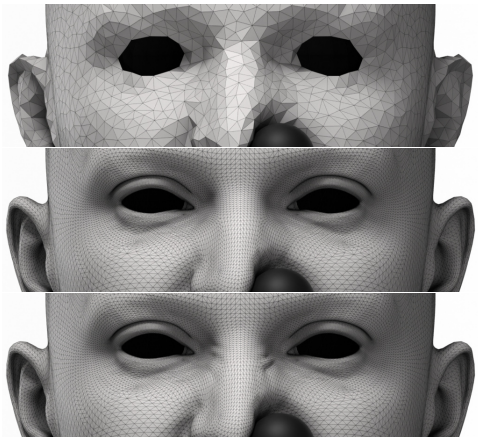


Figure 8.3: *Warping and fine-detail correction for a frowning expression with contact.* Top*: coarse simulation result.* Middle*: warped original mesh without fine-detail correction.* Bottom*: final mesh after applying the corrections, restoring wrinkles at the nose bridge.*

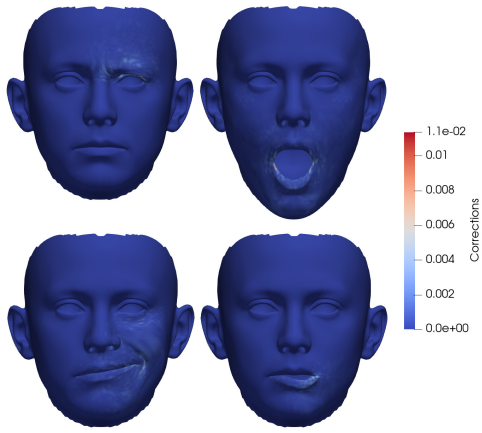


Figure 8.4: *Deviations between the warped mesh and the original rig output for four blendshapes of the Proteus 8k rig. The maximum difference for the entire rig, 0.032 m, occurs at the eyelids.*

The final mesh geometry that the IFA pipeline outputs is then

$$\mathbf{x}_o = \text{warp}\Big(\mathbf{X}_o,\ \text{sim}\big(\mathbf{X} + \sum_{b=1}^{n_b} \mathbf{u}_b w_b\big)\Big) + \sum_{b=1}^{n_b} \mathbf{u}_{c,b} w_b, \tag{8.14}$$

where $\mathbf{u}_b$ are the coarse blendshape displacements used to compute the coarse target expression. Equation (8.14) highlights our *two-scale* approach: a coarse resolution is used for simulation, while the geometric detail is preserved at a high resolution. Algorithm 4 summarizes the main forward pipeline.

Algorithm 4: Forward Simulation Pipeline

foreach frame f **do**
  $\mathbf{x}^f \leftarrow$ coarse_rig$(\mathbf{w}^f)$ ▷ End of frame shell target
  foreach sub-step $k \in \{0, \ldots, n_{\text{substeps}} - 1\}$ **do**
    Update colliders
    $\beta \leftarrow \frac{k}{n_{\text{substeps}}}$
    $\mathbf{x}^t \leftarrow \beta\,\mathbf{x}^f + (1-\beta)\,\mathbf{x}^{f-1}$ ▷ Interpolate from frame targets
    $\bar{\mathbf{x}}^t \leftarrow$ cwise_maxmin$(-\boldsymbol{\epsilon}^r, \mathbf{x}^t, \boldsymbol{\epsilon}^r)$ ▷ Cap actuation forces using Eq. (8.6)
    Assemble $\mathbf{g} = \nabla_{\mathbf{x}} E(\mathbf{x}; \bar{\mathbf{x}}^t)$
    $\Delta\mathbf{x} \leftarrow -\mathbf{H}^{-1}\mathbf{g}$
    $\mathbf{x} \leftarrow \mathbf{x} + \Delta\mathbf{x}$ ▷ Forward and Backward Substitution
    $\mathbf{x} \leftarrow$ constraints$(\mathbf{x})$ ▷ Apply positional constraints
    $\mathbf{v} \leftarrow \frac{\mathbf{x} - \mathbf{x}^{\text{prev}}}{\Delta t^s}$
  $\mathbf{x}^o \leftarrow$ warp_and_correct$(\mathbf{X}^o, \mathbf{x})$ ▷ Transfer deformations and restore details

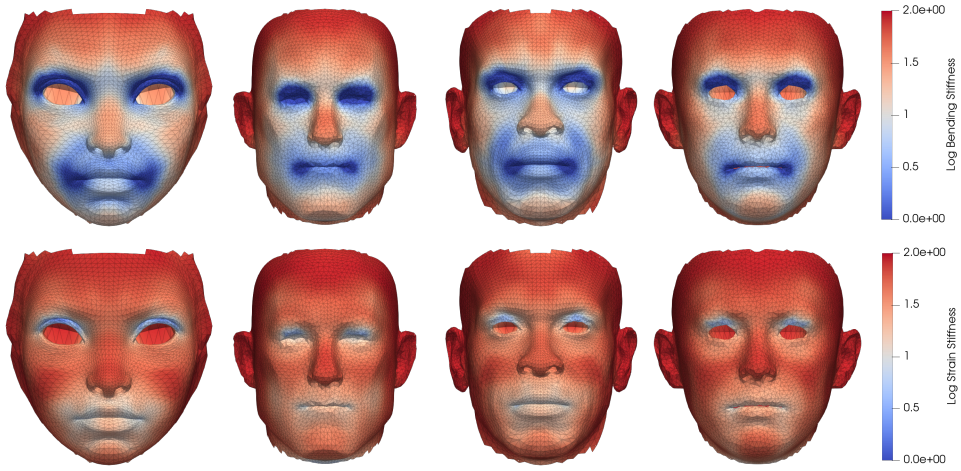


Figure 8.5: *Reconstructed bending (top) and strain (bottom) stiffness fields for the highest-resolution (8k triangles) simulation meshes of the four rigs.*

8.4 Material Stiffness Reconstruction

The forward simulation pipeline described in the previous section relies on carefully chosen material stiffness distributions to reproduce target facial expressions within the reproducibility tolerance ϵ^r, while also responding realistically to contact. For reproducibility, the stiffness must be locally feasible across the entire shell and for all facial expressions. Further, a uniform stiffness fails to capture the stark difference upon contact between soft areas (e.g., cheeks, lips) and stiffer regions (e.g., nose, ears). Meanwhile, material parameters cannot simply be transferred from one character to another, since each has distinct facial geometry and deformation patterns. Using overly soft parameters (e.g. zero stiffness) trivially meets the reproducibility requirement but cannot generate convincing contact deformations. Conversely, too high stiffness will result in the passive shell not deforming enough to match the output of the rig during forward simulation.

To achieve an artist-free, scalable pipeline, suitable material parametrizations must be determined automatically and without the need for parameter fine-tuning. We do this via inverse simulation in a precomputation step, performed once per character identity and for the chosen mesh resolution. Using a differentiable simulation, we solve for non-uniform strain and bending stiffness distributions that allow, in the absence of external interactions, strict reproduction of the original rig expressions within ϵ^r in a quasistatic regime. In our blendshape-based setup, we use the set of non-corrective blendshape expressions for this stage.

As illustrated in Figure 8.5, the resulting stiffness fields produced in the inverse simulation step exhibit locally softer material in high-deformation zones (e.g., cheeks and lips) and stiffer material where bone or cartilage is near the skin surface (e.g., forehead, ears, nose). Our automatic reconstruction method leverages this correlation between deformation and stiffness, enabling IFA rigs to produce realistic contact responses across different regions of the face without the need for additional priors.

8.4.1 Inverse Simulation

The inverse problem in IFA follows the setting introduced in Chapter 4: a quasistatic simulation state is defined implicitly by minimizing a global potential E, and material parameters are optimized through adjoint gradients and an outer optimization loop. Here, the design variables are the non-uniform strain and bending stiffness fields of the facial shell.

We denote by κ^s and κ^b continuous scalar fields of strain and bending stiffness, respectively, discretized at the mesh vertices. Each element's stiffness is computed by averaging its vertex values. Formally, we seek:

$$(\kappa^s,\, \kappa^b) = \arg\min_{\kappa^s,\, \kappa^b} \mathcal{L}, \tag{8.15}$$

where the global loss $\mathcal{L}$ consists of:

$$\mathcal{L} = \mathcal{L}^r + \mathcal{L}^s + \mathcal{L}^b + \mathcal{L}^{ls} + \mathcal{L}^{lb}. \tag{8.16}$$

Here, $\mathcal{L}^r$ is a barrier-type shape-matching loss for reproducibility, $\mathcal{L}^s$ and $\mathcal{L}^b$ are target strain and bending stiffness losses, and $\mathcal{L}^{ls}$ and $\mathcal{L}^{lb}$ are their corresponding smoothing regularizers.

Reproducibility Barrier. The term $\mathcal{L}^r$ enforces all expressions to remain within ϵ^r of their targets after simulation. We use a log barrier for such constraint defined as

$$\mathcal{L}^r = \sum_{b=1}^{n_f} \sum_{i=1}^{n_v} -\lambda^r \log\left(1 - \left(\frac{\|\mathbf{x}_{b,i} - \mathbf{x}^t_{b,i}\|_2}{\epsilon^r}\right)^2\right), \tag{8.17}$$

where λ^r is the barrier stiffness, $\mathbf{x}_{b,i}$ the simulated position of vertex i in expression b, and $\mathbf{x}^t_{b,i}$ its corresponding target. Violations of the reproducibility constraint incur an infinite penalty, so a valid minimizer (if it exists) must remain strictly within ϵ^r of each target.

Target Stiffness. Although zero stiffness trivially achieves reproducibility, we typically want to be as close as possible to user-defined target stiffness values that promote the desired contact responses. For strain, we write:

$$\mathcal{L}^s = \sum_{i=1}^{n_v} \frac{1}{2} A_i \lambda^{ts} \left\|\kappa^s_i - \kappa^{ts}\right\|_2^2, \tag{8.18}$$

where λ^{ts} is a scaling factor, A_i the Voronoi area of vertex i in the rest pose, κ^s_i the strain stiffness, and κ^{ts} a target stiffness. Although κ^{ts} can be tuned on a per-identity or per-scene basis to match artistic or stylistic requirements, this is not strictly necessary. In our experiments, we show that a fixed value works well across different character styles and resolutions. The corresponding edge-based smoothing term is

$$\mathcal{L}^{ls} = \sum_{i=1}^{n_e} \frac{1}{2} \lambda^l \left(\frac{\kappa^s_{i1} - \kappa^s_{i2}}{\|\mathbf{e}^0_i\|_2}\right)^2, \tag{8.19}$$

where λ^l is the smoothing parameter, κ^s_{i1} and κ^s_{i2} are the stiffness values at the edge's vertices, and $\|\mathbf{e}^0_i\|_2$ its rest length. Analogous definitions hold for bending stiffness κ^b.

8.4.2 Adjoint Method

Let $\theta = (\kappa^s, \kappa^b)$. To solve the above optimization problem using gradient-based minimization methods we need $\frac{\mathrm{d}\mathcal{L}(\mathbf{x},\theta)}{\mathrm{d}\theta}$, that is, the *total* derivative of the loss with respect to the unknowns θ. In this quasistatic inverse stage, $E(\mathbf{x}, \theta)$ denotes the assembled global shell potential, using the same local actuation, strain, and bending potentials as the forward formulation but without inertial or damping terms. Following the adjoint method introduced in Foundations Section 4.3, this derivative is computed as

$$\frac{\mathrm{d}\mathcal{L}}{\mathrm{d}\theta} = \frac{\partial \mathcal{L}}{\partial \theta} + \lambda^T \frac{\partial^2 E}{\partial \mathbf{x}\, \partial \theta}, \quad \text{for} \quad \left(\frac{\partial^2 E}{\partial \mathbf{x}^2}\right)^T \lambda = -\left(\frac{\partial \mathcal{L}}{\partial \mathbf{x}}\right)^T, \tag{8.20}$$

where λ is the adjoint state vector. For each candidate θ, we first solve the quasistatic stationarity system $\nabla_{\mathbf{x}} E(\mathbf{x}, \theta) = \mathbf{0}$ for all target expressions, then compute λ and assemble the gradient. The dimensionality of the inverse problem is $2n_v$ (the nodal strain and bending stiffness values), whereas the quasistatic simulation has $3n_v n_b$ degrees of freedom (the 3D positions of all the simulation vertices for all the facial expressions).

Unlike for the forward simulation where we can use position estimations ($\hat{\mathbf{p}}$ in Eq. (8.8)) from the previous time step, this is not an option in quasistatics. Therefore, we solve the inverse simulation with $\hat{\mathbf{x}} = \mathbf{x}$ (no lagging). Also, since stiffness values typically span several orders of magnitude, we instead solve for $\log(\theta)$ to keep optimization variables within a manageable range and improve convergence.

8.4.3 Optimization

As in the outer optimization setting discussed in Section 4.4, the parameter-space optimization must respect feasibility of every forward solve. The barrier term in Eq. (8.17) is undefined outside $\|\mathbf{x} - \mathbf{x}^t\|_\infty < \epsilon^r$, so the optimizer must enforce feasible iterates. Consequently, the initial guess (κ^s, κ^b) must already be feasible (starting from $\mathbf{x} = \mathbf{x}^t$ suffices), and the optimizer must never step outside feasible space, which can be enforced with a line search. To improve efficiency, it is possible to significantly reduce the dimensionality of the quasistatic simulation by only considering regions of the facial expressions featuring non-zero deformations. Finally, adding a mild regularization to penalize excessively stiff values can prevent ill-conditioning and improve optimization stability.

8.4.4 Geometrical Concerns

As described in Section 8.3.1, we impose a maximum depth constraint to produce realistic contact responses. Without it, an object could easily be pushed into the face interior. To remain fully anatomy-free, we estimate a maximum depth for each vertex δ_i from the provided rig expressions, taking the largest normal displacement observed across all expressions. This heuristic allows realistic deeper normal displacements in fleshy areas such as the cheeks, while keeping thinner areas, such as the forehead, shallow. Ears and noses, which show minimal normal-direction deformation, are identified (e.g., via Shape Diameter Function [Shapira et al. 2008]) and assigned a fixed depth δ^{thin}. Jaw-intensive expressions are excluded from this depth computation to avoid overly softening the mandible area.

We omit the eyelids from the shell simulation and directly prescribe their displacements since they are difficult to handle reliably and we are not interested in simulating elasticity and

Table 8.2: Vertices *and* Triangles *refer to the original input mesh.* Fit Elements *indicates the total number of elastic elements used in the inverse simulation.* IFA *represents the full forward simulation pipeline. For reference,* Orig. Rig *shows timings for the original rig run in a sparse Eigen implementation.*

	Vertices	Triangles	Fit Elements	Fit Time [s]	IFA [ms]	Orig. Rig [ms]
Aura 2k	5230	10363	56175	44.488	0.390	0.296
Aura 4k	5230	10363	109411	152.170	0.578	0.296
Aura 6k	5230	10363	162863	400.183	0.691	0.296
Aura 8k	5230	10363	216579	1487.670	0.834	0.296
Bowen 2k	21706	43258	48166	46.334	1.048	1.188
Bowen 4k	21706	43258	91913	106.222	1.075	1.188
Bowen 6k	21706	43258	137219	231.864	1.154	1.188
Bowen 8k	21706	43258	180267	668.068	1.273	1.188
Jupiter 2k	5273	10454	48497	18.180	0.368	0.258
Jupiter 4k	5273	10454	94248	98.230	0.560	0.258
Jupiter 6k	5273	10454	137734	324.941	0.648	0.258
Jupiter 8k	5273	10454	186378	1095.340	0.764	0.258
Proteus 2k	21117	42051	50072	53.374	1.035	1.103
Proteus 4k	21117	42051	95571	150.082	1.051	1.103
Proteus 6k	21117	42051	139480	295.872	1.106	1.103
Proteus 8k	21117	42051	184760	896.287	1.245	1.103

contact in there. Lastly, we use uniform isotropic decimation to provide a consistent contact response throughout the face, even for very coarse resolutions, and minimize visual artifacts that could arise from contact with elongated triangles.

8.5 Results

In this section, we present performance metrics for our method and demonstrate its applicability through representative examples. All experiments were conducted on a laptop with a 12th Gen Intel® Core™ i7-12700H CPU at 2.7 GHz (14 cores, 20 threads) and 32 GB of RAM. We utilize four blendshape rigs from the open-source release by Kavan et al. [2024], each containing around 250 blendshapes. Although these rigs include the full character bust, we extract the face region for the simulation, and normalize it to a height of 1 m. Table 8.2 indicates the resolution of each extracted face mesh.

8.5.1 Inverse Simulation: Stiffness Reconstruction

We begin by examining the inverse simulation material stiffness reconstruction in four face rigs and four simulation resolutions each. We use the 72 non-corrective blendshapes provided in each input rig as the target expressions that our pipeline should reproduce. All examples in this chapter use the same reconstruction parameters, listed in Table 8.3, demonstrating that our method does not require fine-tuning per character, resolution, or artistic style.

The target strain and bending stiffness are chosen based on realistic ear interactions since they get assigned this target stiffness due to their lack of deformation in the expressions. The maximum actuation pressure $p^{\max}$ determines how closely the shell can approach this target stiffness while still maintaining reproducibility. Its value was established empirically by testing

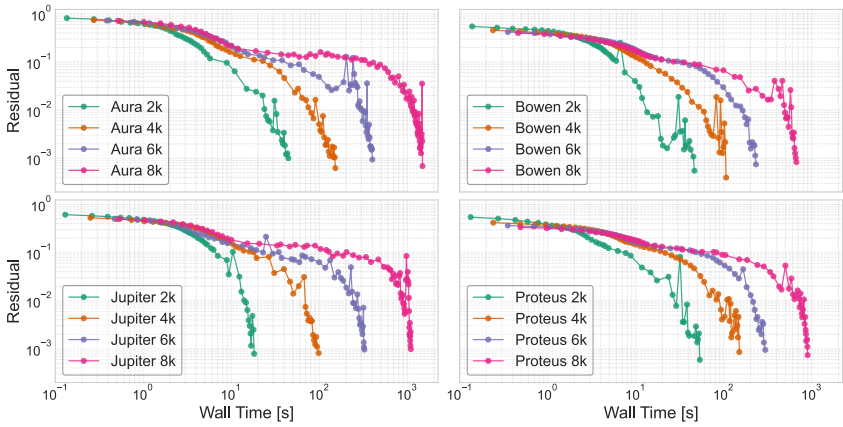


Figure 8.6: *Convergence of the material stiffness optimization for all characters and resolutions. On average, L-BFGS requires 48, 58, 60, and 70 iterations for 2k, 4k, 6k, and 8k resolutions, respectively.*

contact responses across the face for a sample of characters and resolutions, and selecting the one that provided the most satisfactory outcome. Note that these parameters are not tailored to any specific identity or resolution and that they work well across different rigs.

Quasistatic shell solves use the open-source STARK library (Chapter 6). The Newton solver uses Project-On-Demand indefiniteness stabilization [Longva et al. 2026] with block-diagonal preconditioned conjugate gradient (PCG). The solver runs in parallel with double-precision floating-point arithmetic.

We employ L-BFGS [Nocedal and Wright 2006] with history size 10 for the material optimization, solving for both strain and bending stiffness simultaneously. The partial derivatives needed by the adjoint method are generated by SymX (Chapter 5). Figure 8.6 reports the convergence of our material parametrization across the four characters and four resolutions. We find L-BFGS to be well-suited as it exhibits near-quadratic convergence when approaching the solution while keeping the number of iterations relatively low, even at higher resolutions.

Here we note some measures that improve the convergence and stability of L-BFGS for our problem. We cap step sizes at 0.5 in log space to curb overly large stiffness changes before reaching the near-quadratic regime. The optimization starts with a low stiffness (10^{-4}) so that L-BFGS can accumulate curvature information cheaply. A backtracking line search employs three checks in sequence: (1) ensuring reproducibility constraints remain satisfied, (2) assessing whether the simulator requires excessive Newton or CG iterations, and (3) verifying sufficient descent [Nocedal and Wright 2006]. Should the line search fail, the L-BFGS history is reset, reverting to gradient-based steps but avoiding stagnation. Though the problem is essentially nested with three layers of optimization (L-BFGS, Newton, and PCG), these strategies yield robust and efficient performance.

Finally, Figure 8.7 illustrates that our stiffness reconstruction converges consistently across multiple simulation resolutions of the same rig.

8.5.2 Forward Pipeline Performance

Our implementation of the forward IFA pipeline runs single-threaded in floating-point precision. This approach accommodates broader interactive applications (e.g., telepresence or

Table 8.3: *Stiffness Reconstruction Parameters.*

Symbol	Description	Value
ϵ_{opt}	Optimization tolerance	10^{-3}
ϵ_{sim}	Newton tolerance	10^{-3} N
ϵ^r	Reproducibility tolerance	10^{-2} m
$p^{\max}$	Maximum actuation pressure	10^2 N m^{-2}
κ^{ts}	Target strain stiffness	10^2 N m^{-1}
κ^{tb}	Target bending stiffness	10^2 N m^{-1}
λ^r	Barrier parameter	0.002
λ^{ts}, λ^{tb}	Target stiffness scaling	1
λ^l	Smoothness regularization	5×10^{-7}

Table 8.4: *Forward Simulation Parameters.*

Symbol	Description	Value
Δt	Time step	0.01 s
Δt^s	Sub-stepping time step	0.002 s
ρ	Surface density	0.1 kg m^{-2}
ν	Damping coefficient	0.01
δ^{thin}	Thin features depth	0.05 m
$\varepsilon^{\max}$	Max strain threshold	1.0

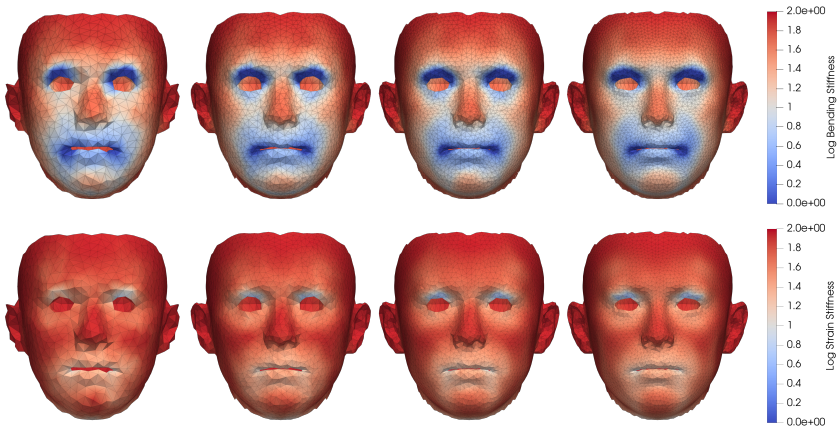


Figure 8.7: *Reconstructed bending (top) and strain (bottom) stiffness at four simulation resolutions for the Proteus rig (2k, 4k, 6k, and 8k triangles).*

games) where the pipeline operates alongside other processes, and also leaves room for running multiple characters in parallel. We use Eigen [Guennebaud et al. 2010] for most linear algebra routines, except for the linear system solve, for which we employ an open-source optimized and vectorized version of forward/backward substitution released along with the method by Kugelstadt et al. [2021].

Figure 8.8 presents the runtimes for four characters across four resolutions. On average,

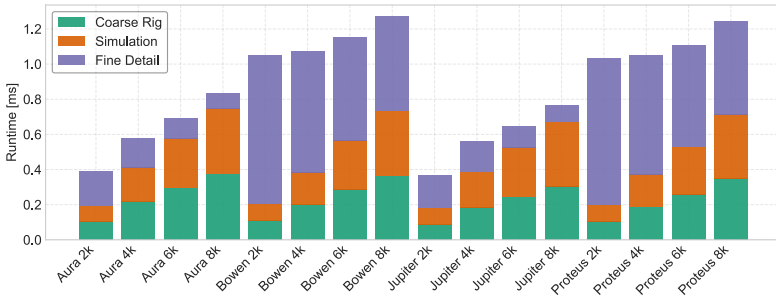


Figure 8.8: *Average per-frame runtimes for the forward IFA pipeline on four characters and four resolutions, shown as a stacked bar plot for coarse rig, simulation, and fine-detail correction stages.*

48.2% of the simulation time is spent building the gradient, 41.7% on the linear system solve, and 10.1% on the remaining components. As expected, the runtime of the coarse rig and simulation grows roughly linearly with the shell mesh resolution. Meanwhile, the fine-detail correction stage depends on the resolution of the original input rig (see Table 8.2), as do the original rig timings.

All times lie between 0.39 and 1.27 milliseconds per frame update, comfortably within the runtime budget required by high refresh rate applications.

8.5.3 Reproducibility

We now compare the output of our method with the original input rig by running an example animation sequence for all characters at different resolutions. As with the stiffness reconstruction, the same simulation parameters apply to all characters and resolutions (see Table 8.4).

A split-screen comparison in Figure 8.9 shows that the IFA pipeline perceptually reproduces the original rig's resolution and expressiveness for all characters and resolutions. The key benefit is that the IFA rig performs a contact-aware *simulation*, whereas the original rig is purely *geometrical*. Since IFA effectively decouples the render and the simulation resolutions, it adapts to different computational budgets (for both inverse and forward components) without sacrificing visual fidelity. Figure 8.10 provides color-coded deviations over representative frames.

Notably, none of the expressions shown in Figure 8.9 corresponds directly to a single blendshape from the stiffness-fitting set, because all expressions in this animation are linear combinations derived by the rig. Nonetheless, our optimization strategy remains effective as long as the runtime expressions lie within the rig's expected expression space (i.e., $\|\mathbf{w}\|_\infty \leq 1$). If extrapolated expressions are expected to appear, the material optimization process should be made aware of this, for example fitting stiffness on "exaggerated" blendshapes, e.g. $1.2\mathbf{w}$, representative of the use at run time. Still, minor discrepancies can arise due to dynamic effects during fast motions, such as the rapid mouth movement illustrated in the third row of Figure 8.9. Reducing shell density or adjusting damping can mitigate these dynamic artifacts if desired.

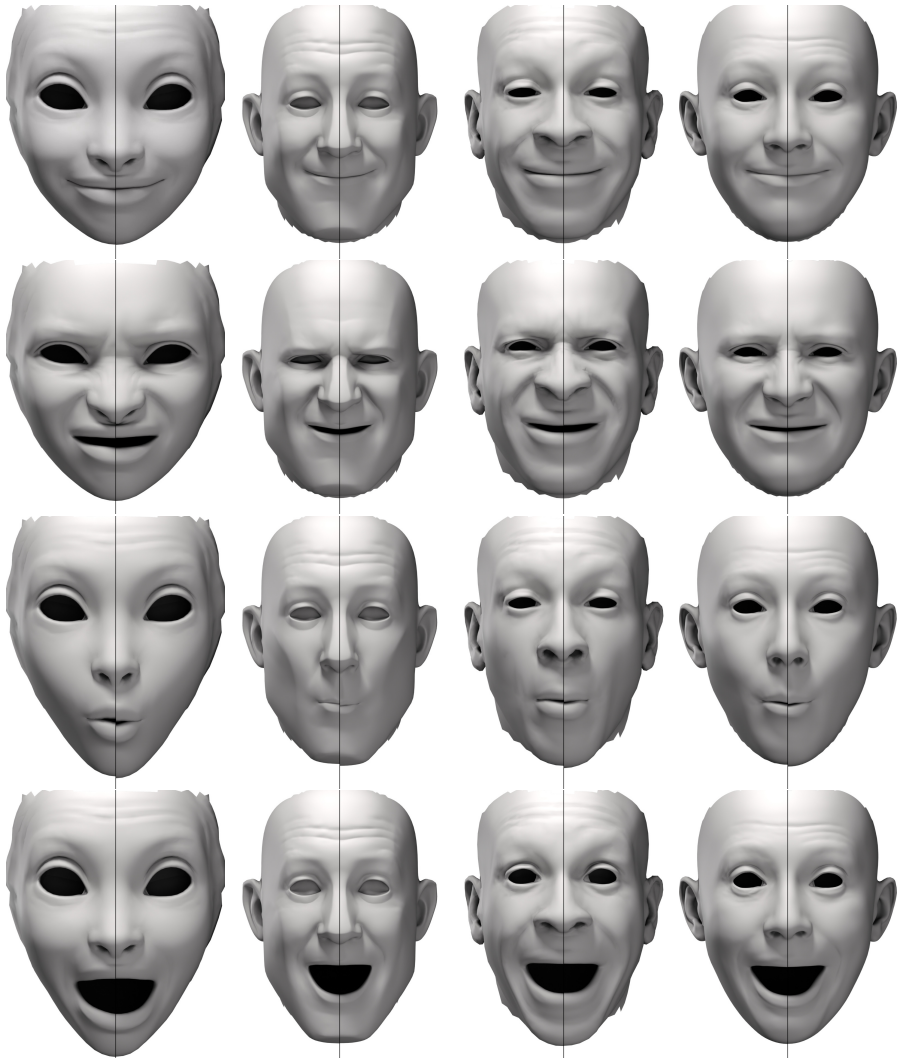


Figure 8.9: *Split-screen comparison between the output of the IFA simulation pipeline (left) and the original rig (right) for four different characters on the same animation sequence. One character per column: Aura 2k, Bowen 6k, Jupiter 4k and Proteus 8k. Each row corresponds to the same frame of the animation sequence.*

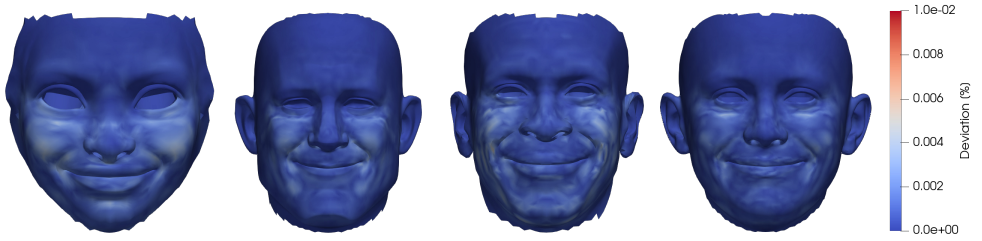


Figure 8.10: *Deviation between the output of the IFA pipeline and the original rig for a given frame and the exemplary characters and resolutions used previously: Aura 2k, Bowen 6k, Jupiter 4k and Proteus 8k. The corresponding maximum deviations for the entire sequence for each model are: 3.4%, 1.8%, 2.5%, 3.2%, respectively.*

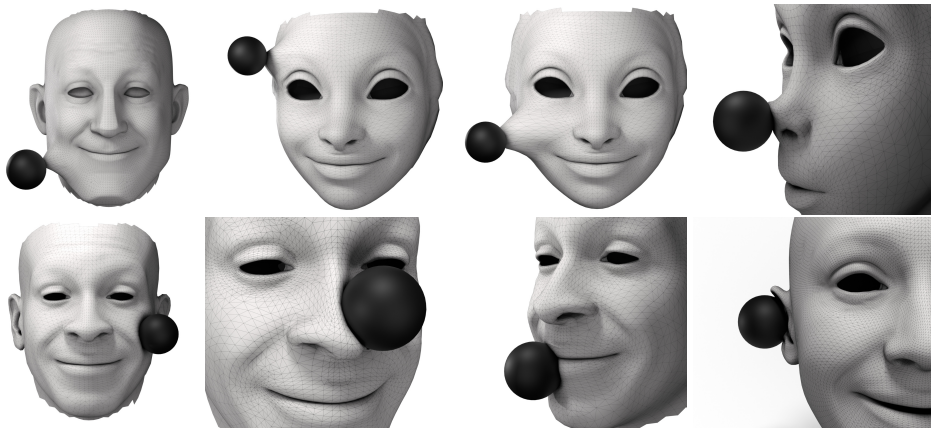


Figure 8.11: *A variety of contact and pull interactions performed in real time on different characters. Frames are rendered offline for improved lighting and camera settings.*

8.5.4 Interactions

We evaluate our simulation pipeline's contact capabilities using a variety of interactions, characters, and resolutions. Figures 8.1 and 8.11 show different contact interactions performed in real-time sessions.

In all examples, the corresponding blendshape rig is actively driving facial expressions. We use a sphere as the SDF-based collider, though any rigid shape is possible. The sphere moves due to a tracking force that follows the mouse and that is bounded in magnitude, ensuring safe interactions. Skin pulling is achieved by pinning simulation mesh vertices that contact the sphere, triggered by a user action. All interactions are frictionless.

These examples confirm that our method successfully incorporates realistic skin-level deformations from contact into existing facial animation rigs across various characters and simulation resolutions. Contact at the cheeks, forehead, nose, lips, and ears appear realistic, the coarse simulation resolution is hardly perceptible, and the input fidelity is not compromised.

8.6 Limitations and Future Work

While the IFA pipeline addresses the problem of contact in an animation setting, as presented in this chapter it does not *improve* the input rig's expressions (apart from adding dynamics). In any case, our method remains compatible with any approach that modifies the rig before feeding it into our contact simulation.

Because IFA uses a surface-only simulation, capturing volumetric effects (e.g., volume conservation) is challenging. In the future, we believe an approximate volume-based measure could be derived by analyzing the volume swept by the surface under contact, or by incorporating volumetric elements specifically for protruding regions such as the nose and lips.

Moreover, our method is designed to simulate local deformations caused by contact. Movements affecting the entire head (e.g., a strong push to the forehead) cannot emerge within the proposed method alone. However, such effects can be integrated if the skeletal rig is used in tandem with IFA's computed contact forces. Furthermore, our maximum depth calculation is a heuristic that performs well in practice without anatomical priors. However, more sophisticated methods (or even a skull if available) can be readily incorporated. Handling self-contact (e.g., between the lips) remains a task for future exploration.

Finally, another interesting avenue for future work is to explore mechanisms to switch between the original rig and the IFA rig at runtime. This could allow resource savings on battery-powered devices when no interaction is taking place, provided seamless transitions can hide any visual discontinuities.

8.7 Conclusions

We presented a novel surface-only real-time simulation pipeline that augments existing face rigs with shell mechanics and contact handling, backed by strong fidelity preservation guarantees. Our method requires only the rig itself and standard simulation parameters; no artist intervention or anatomical priors are needed. Additionally, we introduced a new material reconstruction procedure for shells, using inverse simulation, to obtain non-uniform stiffness fields that support realistic contact while ensuring reproducibility. This setup step runs in a few minutes on commodity hardware.

We have demonstrated that our simulation solution perceptually reproduces the original rig without compromising the resolution and expressiveness while delivering realistic interactions on different areas of the face, including cheeks, forehead, nose, lips and ears. Our method generalizes across different characters (including heavily stylized ones) and simulation mesh resolutions without the need for per-case parameter fine-tuning.

Within the thesis, IFA serves as an application chapter where the previous simulation and differentiation tools are combined in a concrete physically-based simulation pipeline. The chapter therefore demonstrates the practical value of differentiable simulation and simulation infrastructure in a production-oriented setting.

CHAPTER 9

Fast Octree Neighborhood Search for SPH Simulations

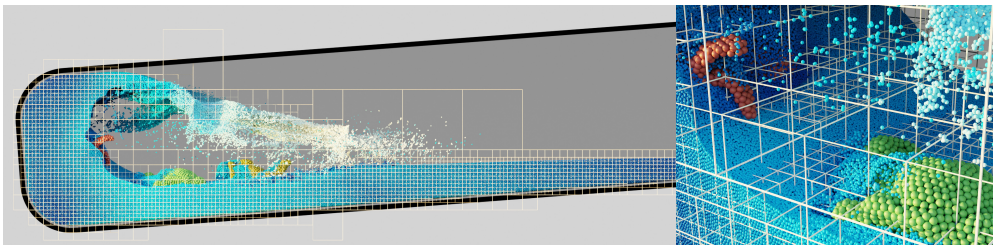


Figure 9.1: *Multi-resolution Smoothed Particle Hydrodynamics simulation of a swinging box that contains fluid and deformable objects. 5.6 million particles (from blue to white) are used to discretize the fluid and a total of 12 thousand particles (in red, green and yellow) are used to discretize the three deformable objects. The fluid particles have a* 2.5× *smaller radius than the ones used for the solids. The octree used to accelerate the neighborhood search is shown as a white wireframe. On the left, a far view shows the adaptivity of our acceleration structure at a global scale. On the right, a close-up picture better shows the difference in particle sizes and how the octree behaves around that area of interest. For better visibility on the right picture, we show the octree uniformly coarsened by one level.*

This chapter presents the thesis contribution on hardware-aware simulation infrastructure. It is adapted from the original TreeNSearch (TNS) publication [Fernández-Fernández et al. 2022], where the method is developed and evaluated for neighborhood search in Smoothed Particle Hydrodynamics (SPH). The implementation is open source[1].

Although the concrete application is particle-based fluid simulation, the chapter addresses a broader simulation infrastructure question: practical simulation performance depends not only on mathematical models, but also on data structures, memory access, data flow, branch behavior, vectorization, and task decomposition. This hardware-aware perspective connects

[1]The TreeNSearch source code is available at https://github.com/InteractiveComputerGraphics/TreeNSearch.

to other chapters of the thesis, including the collision-processing infrastructure of STARK (Chapter 6) and the evaluation and assembly pipelines of SymX and IFA (Chapters 5 and 8).

9.1 Introduction

Smoothed Particle Hydrodynamics (SPH) is an established meshless method for simulating fluids, elastic solids and granular material. SPH expresses quantities of interest for a given particle, such as density or pressure forces, as a sum of contributions from *neighboring* particles in close proximity. The identification of neighboring particles, the neighborhood search, is therefore an integral component of all SPH simulators. Two particles are neighbors if the distance between them is smaller than the support radius of the SPH kernel. Historically, most SPH simulators in computer graphics have used constant support radii. This assumption has been exploited by specialized fixed-radius neighborhood search algorithms.

A key idea often used in fixed-radius neighborhood search methods is the placement of each particle in a sparse uniform grid with cell size equal to the support radius. Then all neighbors of a particle are guaranteed to be located in the cell of the particle or in any of the adjacent neighbor cells. This way, the fixed-radius approach performs an almost minimal number of distance comparisons to determine if two particles are truly neighbors. On the other hand, the small number of particles considered for each grid cell makes it difficult to fully exploit vectorization and pipelining in modern CPU hardware, which are key enablers for high performance. Furthermore, the small size of the cells necessitates a larger number of cells, leading to noticeable computational costs for managing the acceleration structure itself.

We find that if we use appropriately sized, larger, and therefore fewer, cells, the computational cost of the acceleration structure is significantly reduced, but the overall cost of the brute force distance comparisons is only marginally increased. Therefore, the overall cost is lower.

By aggregating particles into these larger cells, we find that it is favorable to exchange the sparse uniform grid acceleration structure with a specialized octree algorithm where the tree leaf nodes consist of clusters of cells. The top-down octree construction clusters cells so that each leaf node has approximately the same number of particles, regardless of the number of cells, which benefits parallelism and ensures that the brute force stage always works on a near-optimal number of particles, thereby maximizing throughput. Contrary to conventional wisdom, which suggests using the support radius as cell size is essential to performance, we demonstrate through numerous examples that our method is consistently faster than state-of-the-art fixed-radius methods.

A seemingly limitless number of choices can be made in the design of a tailor-made acceleration structure. In this chapter we present high-level decisions in detail that enable high throughput for performance-critical parts of the algorithm. In this context, we put special emphasis on vectorization and a branchless implementation.

Some state-of-the-art methods *z-sort* the particle data and then rely on this ordering for gathering particle data for distance comparisons. Sorting particle data in this way is in any case beneficial to SPH methods, as it improves cache locality for the SPH loops [Ihmsen et al. 2011]. However, it is generally not necessary to do this for every time step of the simulation; amortizing this cost over several time steps is usually preferable. Our method does not require z-sorted particle data to produce correct output. Instead, we propose a technique in which we

take advantage of *almost sorted* data for a more compact representation of particle data in our octree.

Although the fixed-radius assumption has proven to be effective for simulating single-phase fluids, it nonetheless imposes severe limitations on complex multi-physics simulations, such as multi-phase fluids or fluid-solid interaction. In these settings, it is natural to want to use different support radii for different objects. Here, slow-moving, viscous fluids or elastic solids generally require fewer particles for high-fidelity results compared to the intricate patterns formed by swirling water.

To ensure efficient simulations in these settings, we must do away with the fixed-radius assumption. Our proposed octree-based method is also capable of handling SPH simulations where particles have different support radii. Our results therefore challenge the prevalent notion that the fixed-radius assumption is the key to fast neighborhood search in SPH simulators.

We demonstrate the performance of our method across both fixed-radius and multi-resolution examples. Our fast neighborhood search requires only roughly 5%-10% of the total simulation time across all of our examples, and consistently outperforms a representative state-of-the-art fixed-radius method across all of our fixed-radius benchmarks, with a speedup of up to 1.9×. Figure 9.1 illustrates the distribution of particles in our octree in a large-scale multi-physics scene.

9.2 Related Work

SPH is a popular method in computer graphics for the simulation of a wide range of materials and effects. The discussion that follows reflects the state of the art at the time the original research was conducted [Fernández-Fernández et al. 2022]. We first review relevant work on SPH and then focus on neighborhood search methods. The section concludes by classifying the main approaches and identifying the limitations that motivate our work.

9.2.1 Smoothed Particle Hydrodynamics

While the SPH formulation was originally introduced in the field of astrophysics [Gingold and Monaghan 1977], today's application areas also include the field of computer graphics. Developments in SPH include simulations of incompressible fluids [Ihmsen et al. 2014a; Bender and Koschier 2017], highly viscous materials [Takahashi et al. 2015; Weiler et al. 2018], deformable solids [Peer et al. 2017; Kugelstadt et al. 2021] and snow [Gissler et al. 2020]. Broader overviews of SPH research can be found in surveys by Ihmsen et al. [2014b] and Koschier et al. [2022]. For simulating the aforementioned effects, the SPH method relies on locally computing the required physical quantities by interpolation using the discrete particles inside a given support radius, which is typically dependent on the particle radius.

In computer graphics it is common practice to use the same support radius for all particles. However, to improve runtime, incorporating different particle resolutions may be considered to significantly reduce the amount of particles without sacrificing visual quality. For example, in simulations between a low and highly viscous fluid a fine resolution is needed to capture the high frequency motions of the former material, while larger particles can be used for the latter. Thus, choosing an independent, appropriate resolution may help to speed up the simulation significantly.

Incorporating this concept, Desbrun and Cani [1999] spatially adapt the particle resolution based on the density deviation of neighboring particles. The idea of particle splitting and

merging was also adopted in several later publications [Adams et al. 2007; Winchenbach et al. 2017; Winchenbach and Kolb 2021]. Another approach is to utilize different, distinct particle resolutions in desired regions of interest [Solenthaler and Gross 2011; Horvath and Solenthaler 2013]. While all of the aforementioned methods show a significant improvement in simulation quality or efficiency, special attention must be given to computing the neighborhood of each particle. As denoted by Xia and Liang [2016], incorporating a single, standard fixed-radius neighborhood search algorithm like compact hashing [Ihmsen et al. 2011] must use the largest support radius of all particles to avoid artifacts. This introduces significant computational overhead for smaller particles since their associated computations will include several particles outside their support radius. Horvath and Solenthaler [2013] as well as Solenthaler and Gross [2011] execute one neighborhood search for each level of detail in their multi-resolution approaches. This leads either to an increased memory footprint at the interface of different resolutions or requires an unphysical boundary region. In summary, efficiently determining the neighborhood in spatially adaptive SPH simulations is a challenging task.

9.2.2 Neighborhood Search

In SPH-based simulations, the neighborhood search step is an integral component since a field quantity for a fluid particle is computed by interpolating over its neighboring particles within the support radius of the SPH kernel. Therefore, different approaches have been proposed in the computer graphics community. An overview of CPU- and GPU-based algorithms can be found in the survey of Ihmsen et al. [2014b]. We focus on CPU-based algorithms, which, despite the advent of efficient GPU simulators, remain relevant especially for very large simulations that are limited by the amount of available memory, such as those shown by Band et al. [2020].

State-of-the-art neighborhood search methods are mainly based on either spatial grids, Verlet lists [Verlet 1967] or hierarchical data structures. In computer graphics, neighborhood-search methods often rely on uniform grids. These store the particles inside a uniform background grid and calculate the neighbors of a particle by querying the adjacent cells. Most grid-based approaches are either based on Cell-Linked Lists (CLL) [Green 2010; Domínguez et al. 2010; Pelfrey and House 2010; Band et al. 2020] or hash maps [Ihmsen et al. 2011; Tang et al. 2018; Winchenbach and Kolb 2020]. While offering fast construction and query times from the standpoint of computational complexity, grid-based methods may suffer from high cache miss rates on the CPU if the particles are not spatially indexed and sorted. Regardless of the spatial sorting, the resulting neighbor lists are recomputed in each time step.

Leveraging the temporal coherence of neighboring particles, the idea of Verlet lists is to extend the amount of potential neighbors in relation to a particle's velocity. Thus, these predicted neighbors may help to speed up the neighborhood search [Viccione et al. 2008; Willis et al. 2018]. The main disadvantages of using Verlet lists is the computational overhead if the prediction fails to capture the movement of particles and the increased memory footprint. The latter also results in them being unfavorable for large-scale and GPU-based simulations.

Both of the aforementioned concepts can be modified to suit adaptive particle radii. Winchenbach and Kolb [2020] use a hashed, multi-resolution grid, which exploits the self-similarity of Morton codes for the particle sorting. This way, it is possible to query the acceleration structure at different resolution levels depending on the support radius of any individual particle. Regarding Verlet lists, Winchenbach et al. [2016] compute memory constrained neighbor lists in a predictor-corrector fashion which allows for variable support

radii. In this method, the support radius of a particle is adjusted so that a maximum fixed number of neighbors is attained.

Tree-based neighborhood search methods have also been used to tackle the neighborhood search in spatially adaptive SPH simulations [Hernquist and Katz 1989; Harada et al. 2007; Xia and Liang 2016]. Since this acceleration structure yields a multi-resolution view of the simulation domain, neighboring particles can easily be determined by a traversal of the tree.

The previously denoted categories of neighborhood search algorithms have also been adapted to better suit the strengths of the used underlying hardware. Examples leveraging the streaming architecture of GPUs include grid-based [Green 2010; Gross et al. 2019], Verlet list-based [Willis et al. 2018] and tree-based methods [Xia and Liang 2016].

Our novel approach can be classified as a tree-based method. However, instead of using the acceleration structure for tree-traversal based neighbor checks, the employed octree functions as a domain decomposition. In contrast to traditional grid-based methods, the resulting task size for the brute force stage adapts to the underlying fluid geometry.

9.3 Method

In the following, we define the neighborhood search problem in the context of SPH (Section 9.3.1), and give a high-level overview of how our method solves the problem (Section 9.3.2). We provide a detailed account of the implementation details of our method in Section 9.3.3. In Section 9.3.4, we discuss the required extensions that enable our method to be used for variable support radii. Finally, we discuss the differences to other fast fixed-radius methods in Section 9.3.5.

9.3.1 Problem description

In the standard fixed-radius SPH formulation, a scalar field f is approximated by interpolating values from a discrete set of particles as

$$f(\mathbf{x}_i) = \sum_{j \in \mathcal{N}_i} \frac{m_j}{\rho_j} f(\mathbf{x}_j) W(\mathbf{x}_i - \mathbf{x}_j, h), \tag{9.1}$$

where m_j, ρ_j and $\mathbf{x}_j$ are the mass, the density and the position of particle j. W is a compactly supported kernel with smoothing length h and support radius r. This means that only the particles within the support radius r of $\mathbf{x}_i$ have a non-zero contribution to the value of $f(\mathbf{x}_i)$. Hence, we only sum up contributions of particles j that are in the neighborhood $\mathcal{N}_i = \{j \mid \left\|\mathbf{x}_i - \mathbf{x}_j\right\|_2 \leq r\}$ of particle i. Therefore, SPH simulators require efficient neighborhood search methods to determine the neighborhood $\mathcal{N}_i$ for each particle i.

9.3.2 Method overview

A global brute force approach to the neighborhood search problem has a complexity of $O(n^2)$, which is prohibitive even for medium sized simulations. To address this, our method, like most other approaches, uses an acceleration structure to decompose the global problem into many self-contained subproblems so that checking particle pairs with a large distance is avoided.

Our method employs a novel two stage acceleration structure: In a problem reduction preprocessing step, we assign all particles to the cells of an encapsulating uniform grid. Then,

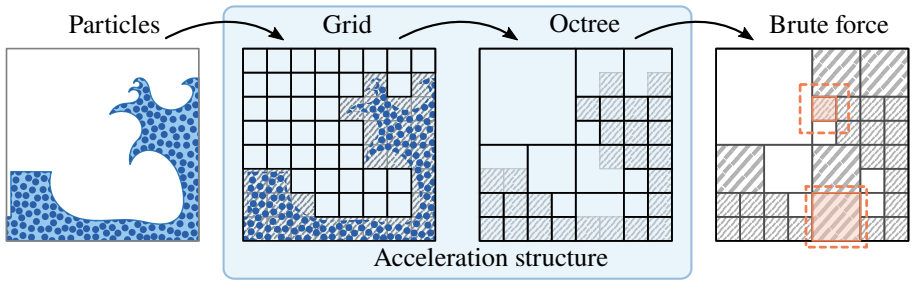


Figure 9.2: *Overview of our method. The input to our method is a list of particles from an SPH simulation. To build an acceleration structure for our method we first map particles to cells of a background grid using Morton z-indices. The set of all non-empty cells is then geometrically subdivided using an octree until all octree leaves approximately contain a specified number of particles. As the octree only works on cells we can avoid looping over any particles in this stage. Finally, for each octree leaf a brute force neighborhood search is performed.*

we use an octree to adaptively find clusters of non-empty grid cells which in total approximately contain a specified number of particles. Finally, we use a brute force approach on the particles of each cluster to generate the corresponding neighbor lists. All stages are performed using optimized SIMD code which is discussed in Section 9.3.3. An overview of our approach is shown in Figure 9.2.

State-of-the-art solutions for the neighborhood search problem that solely use uniform grids rely on two operations where modern computer architectures are slow: random memory reads and hard-to-predict branching. Our method, on the other hand, is designed with modern CPU architectures in mind. For most of the execution time, the CPU executes branchless code on contiguous data with an optimized layout. This is further enhanced by our octree being able to generate clusters of a specific number of particles, which can be processed with higher computational efficiency in comparison to the clusters generated by uniform grids, which are typically very small. These are the main reasons why our octree achieves significant runtime improvements when compared with uniform grid approaches.

9.3.3 Implementation

In this section we discuss the three main stages (see Figure 9.2) of our method in more detail.

Particles to Cells

We employ a uniform grid to classify the particles into cells in order to speed up the construction of the octree, which is the main component of the acceleration structure. In contrast to uniform grid methods, we do not use the grid for queries of neighboring cells. Its only purpose is to reduce the number of entities to be processed in the octree.

To classify the particles into cells, first we compute a bounding box containing all particles. Then we define a uniform background grid with cell size $d \in \mathbb{R}$, which encloses the bounding box of the particles and use the minimum point of the box as origin $\mathbf{o} \in \mathbb{R}^3$. Our results show that a good choice for the cell size is 1.5× the support radius (Section 9.4). The integer coordinates $(k,l,m)_c \in \mathbb{N}_0^3$ of a cell c that contains a particle i are given by the component-wise

floor function

$$(k, l, m)_c = \left\lfloor \frac{\mathbf{x}_i - \mathbf{o}}{d} \right\rfloor . \tag{9.2}$$

By subtracting the origin, all cell coordinates are non-negative. We denote by $\mathcal{P}_c$ the set of particle indices assigned to c and by $\Omega_c \subset \mathbb{R}^3$ the physical domain of the cell.

The input to our octree construction algorithm (Section 9.3.3) is an array of cells. Conceptually, each cell contains a set of particles. One way to compute the cells is to first sort all the particles according to their Morton code [Morton 1966] with respect to the uniform grid. We refer to this process as *z-sorting*. Then all the particles inside a given cell will share the same Morton code, and therefore will be next to each other in the sorted particle array. This is the approach used to construct the CLL data structure employed by Band et al. [2020]. The array of cells can then be constructed by identifying sequences of particles that share the same Morton code in the sorted particle array. As the particles for a given cell are contiguous in the particle array, the particle range associated with a cell can be compactly represented by the *begin* and *end* offsets to the particle array. This idea is illustrated in Figure 9.3.

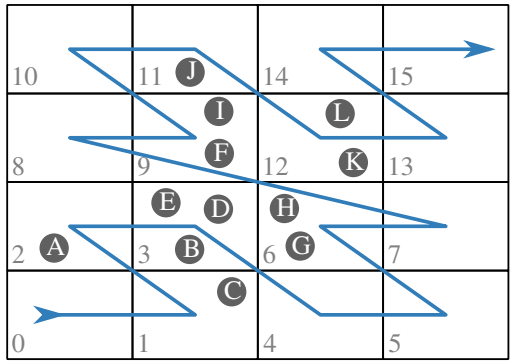
Because we tend to assign approximately 30 particles to each cell (Section 9.4), this means that the octree, which only works with cells and not individual particles, only processes about $1/30$ the number of entities compared to the total particle count, which significantly reduces the computational cost of the acceleration structure. The actual particle data is only gathered in the brute force stage.

A problem with this approach is that a full z-sort on the list of particles can add significant computation time to the neighborhood search. A key component of our approach is the realization that the particle list does not need to be *perfectly* z-sorted before handing off the resulting cells to the octree construction algorithm. Nothing changes in the octree algorithm if two ranges of particles share the same cell in the uniform grid but are not contiguous in the particle array. Because SPH methods tend to occasionally z-sort the particle data for improving cache behavior [Ihmsen et al. 2011], we can therefore exploit the fact that the particle data is expected to be *almost z-sorted* anyway. In the absence of a perfectly z-sorted sequence of particles, our method will still greedily cluster them as described above. In the worst case, if the particles are not close to be z-sorted, our method will still produce the correct result, but performance will degrade: the number of cells in the octree will get closer to the number of particles. We perform a global z-sort of the particles before the first time step of the simulation, and then once every few time steps (Section 9.4).

Octree

We use an octree to divide the global problem into smaller, self-contained subproblems by clustering cells at the leaves of the octree. The octree is aligned to the uniform grid so that any grid cell is geometrically contained in exactly one leaf node. The uniform grid has a power of two number of cells in each direction to ensure that splits in half do not invalidate that restriction. The grid cells that are geometrically contained in an octree leaf node are the *interior* cells for the node. In addition, each leaf node has a set of *exterior* cells. These are the cells that are not geometrically contained inside the leaf node, but have particles that are potential neighbors with the particles in the interior cells.

By construction, each cell is therefore interior in exactly one leaf node, and the complete neighborhood $\mathcal{N}_i$ for any particle i in that cell is a subset of the particles in the leaf node cells (interior and exterior). This implies that the neighborhood search can be broken down into



Particles sorted by z-index

Order	C	A	B	D	E	G	H	F	I	J	K	L
Z-index	1	2	3	3	3	6	6	9	9	11	12	12
Cell offsets	0	1	2	5	7	9	10	12				

(7 cells)

Particles almost sorted

Order	A	B	C	D	E	F	G	H	I	J	K	L
Z-index	2	3	1	3	3	9	6	6	9	11	12	12
Cell offsets	0	1	2	3	5	6	8	9	10	12		

(9 cells)

Figure 9.3: *Mapping particles to the background grid. The particle's z-index identifies the cell of a particle. The list of non-empty cells is then given by the offset into the particle array each time the z-index changes between particles adjacent in memory. For perfectly sorted particles the cell offsets are monotonically increasing. This is not the case for our method where we assume only almost sorted particles.*

independent subproblems: the neighborhoods of interior particles in each octree leaf node can be computed independently in parallel.

The octree is constructed in a top-down manner. Consider an octree node $\mathcal{T}$ with domain $\Omega \subset \mathbb{R}^3$. We define $\widetilde{\Omega}$ as the extended domain obtained by enlarging Ω by the support radius in all coordinate directions. Starting with the root node of the octree, each octree node $\mathcal{T}$ is recursively split into eight equally sized child nodes with non-overlapping domains (see Figure 9.4), in the usual octree fashion. A cell c in $\mathcal{T}$ is assigned to a child node $\mathcal{T}_{\text{child}}$ if the cell domain Ω_c overlaps with octree node domain $\widetilde{\Omega}_{\text{child}}$.

If $\mathcal{T}$ contains only a single interior cell, or the total number of particles across all cells in $\mathcal{T}$ is smaller than a constant threshold, we do not subdivide it further, which means that $\mathcal{T}$ is a leaf node. We study the optimal number of particles per leaf node in Section 9.4.

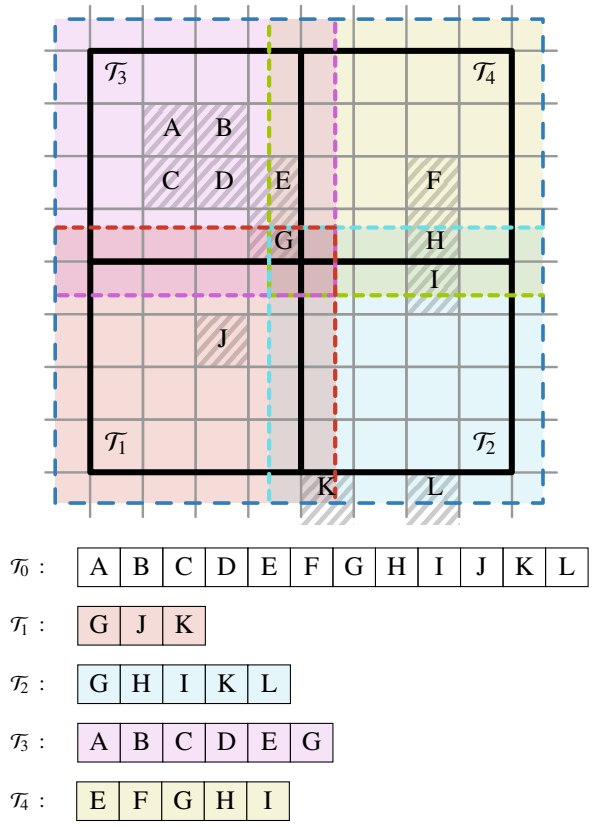


Figure 9.4: *Subdivision of an octree cell $\mathcal{T}_0$. The blue dashed line indicates the parent cell $\mathcal{T}_0$'s enlarged domain $\widetilde{\Omega}_0$ containing 12 non-empty cells A to L (shaded). The thick black lines and colored backgrounds indicate the non-overlapping sub-domains $\Omega_{1\ldots4}$ of the child nodes $\mathcal{T}_{1\ldots4}$ dividing Ω_0 into quadrants (octants in 3D). The non-empty cells are then assigned to all child nodes whose extended subdomains $\widetilde{\Omega}_{1\ldots4}$ (colored dashed lines) overlap with the respective cell.*

Brute force stage

After the octree construction, we are left with a collection of individual subproblems, one per leaf node of the octree. The final task is to compute the neighbor lists of the *interior particles* of each leaf node $\mathcal{T}$, defined as the particles that belong to the interior cells of $\mathcal{T}$. The remaining *exterior* particles are the particles in the exterior cells of $\mathcal{T}$. All neighbors of any interior particle are by construction either internal or external particles of the node.

At this point, only the leaf and cell data for a task is known, therefore, prior to actually computing the pair-wise distances between the particles, the particle data must be gathered. To provide optimal operational conditions for the core of the brute force procedure (computing inter-particle distances), we store the gathered particle coordinates and indices directly in SIMD-friendly data structures. Finally, we calculate the distance from each interior particle i in Ω to each particle in the extended leaf domain $\widetilde{\Omega}$. The neighboring particle indices are then written into the neighbor list of particle i.

```
void pushback_8int(__m256i& src, __m256& cond, int*& dst)
{
    int mask = _mm256_movemask_ps(cond);
    __m256i index_map = lookup_table[mask];
    __m256i shuffled_values = _mm256_permutevar8x32_epi32(src, index_map);
    _mm256_storeu_si256((__m256i*)dst, shuffled_values);
    dst += _mm_popcnt_u32(mask);
}
```

Figure 9.5: *Branchless conditional pushbacks with SIMD. Eight integers are conditionally appended to a list without the use of branches.* `__m256i& src` *is a SIMD type that holds the eight source integers,* `__m256& cond` *is a SIMD type that holds the result of eight floating point comparisons,* `int* dst` *is the destination pointer where the values in* `src` *corresponding to true values in* `cond` *will be copied into.*

The neighbor list data structure is a collection of fixed size blocks of memory. The size of the blocks is chosen large enough to hold a few thousands of single particle neighbor lists. To avoid data races and false sharing, each thread has its own list of these blocks to which it has exclusive access. When a neighbor list cannot be appended to the current active block of a thread, a new block exclusive to that thread is allocated. No memory reallocations of previously written neighbor lists are needed. We use a separate structure for the unlikely cases of a single neighbor list being larger than a block. To access the neighborhood data, a pointer to the beginning of each neighbor list is stored together with the number of neighbors. Particles with consecutive indices are likely to have their neighbor lists stored consecutively in memory, which improves cache coherency during the SPH loops.

Branchless Conditional Pushbacks

A very common operation in our method is to append a value to a list if a certain criterion is met. This *conditional pushback* is a crucial operation during the octree construction and the computation of the neighbor lists. It has three stages: 1. evaluate the condition, 2. if necessary, write the value to the destination pointer, and 3. increment the pointer. Due to branch mispredictions, conditional pushbacks can significantly slow down routines that are otherwise well optimized.

To avoid this issue, we design our solution so that all performance critical conditional pushbacks can be carried out using SIMD permutations. These rearrange data in a SIMD register according to an index map. We can use these permutations for branchless conditional pushbacks by selectively moving the correct items in the SIMD register to the front, using the appropriate index map, and then appending the contents of the register to the destination list. Finally, we increment the destination pointer by the number of conditionals evaluated to true.

In Figure 9.5, we show the actual instructions we use for the case of conditionally appending integer values using the AVX2 instruction set, which gives us access to permutations of up to eight 32-bit integers. Let's assume we have eight integer values (`src`) that we want to selectively copy over to a different list (`dst`) depending whether eight conditional values indicate true or false (`cond`). First, we need to generate a index map from the eight conditional values. While the index map can be generated programmatically, we found that using a precomputed lookup table is faster (line 4). To index the lookup table we need an index between 0 and 255 (all possible eight true/false combinations) from the conditional array. The instruction `movemask` takes the conditional array and returns an integer with its lower bits as one if true and zero if

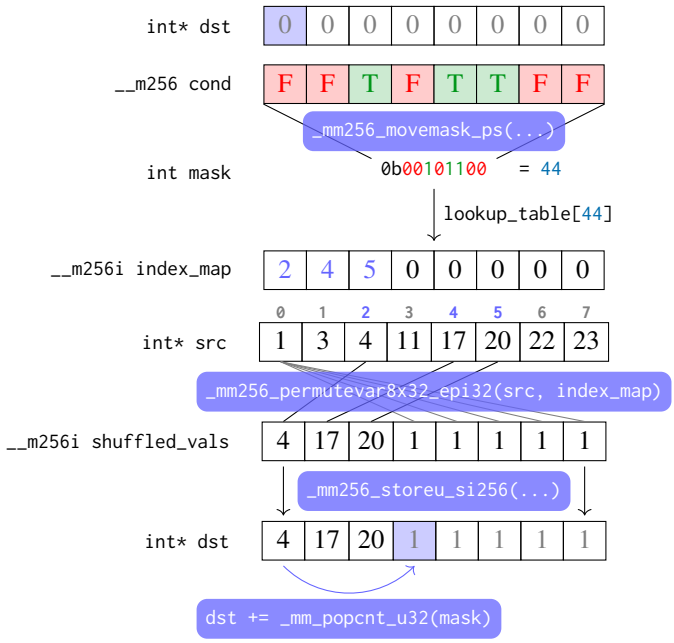


Figure 9.6: *A diagram of the branchless SIMD pushback.*

false which we can use as index (line 3). Then, we can use the `permute` SIMD instruction with the source values and the index map to get the values corresponding to positive conditionals in front of an 8 value array (line 5). This shuffled array is then copied to the destination pointer (line 6). Finally, the destination pointer must be incremented by the exact number of bits equals to one in the mask, corresponding to the number of elements appended to the array, which is determined with the `popcnt` instruction (line 7). See Figure 9.6 for a diagram of the process.

In the end, a procedure that would have taken 8 branches and up to 8 writes and pointer increments, now takes 5 instructions without any branches. In Section 9.4, we present benchmarks that show the improvements this procedure brings to our algorithm in several of its stages.

Z-sort

As discussed in Section 9.3.3, our method takes advantage of the particles being almost z-sorted. We do not need to explicitly sort the particles for our method before every neighborhood search, however, we z-sort them occasionally to keep them approximately sorted at all times.

Since the cells that we use for the octree are effectively a coarse representation of the particles, we can use them to speed up sorting the particles. By sorting the cells instead of the particles we can very significantly decrease the computational time needed for the z-sort operation. Since the particles are almost z-sorted, the cells will also be almost z-sorted, therefore sorting them with a suitable sorting algorithm is very fast. The updated particle order is found by concatenating all the particle indices belonging to the cell in the new cell

order. As shown in Section 9.4, this approach can be significantly faster than explicitly sorting the whole particle array directly.

9.3.4 Variable support radius

While the main objective of our octree design is to maximize performance for fixed-radius neighborhood search problems, we extend our method utilizing the adaptive nature of the octree to also support variable support radii.

Different SPH methods use different definitions for the particle neighborhood in the context of variable support radii. We use the definition proposed by Adams et al. [2007], which states that two particles i and j with positions $\mathbf{x}_i$ and $\mathbf{x}_j$ and support radii r_i and r_j are neighbors if

$$\left\|\mathbf{x}_i - \mathbf{x}_j\right\|_2 \leq \max(r_i, r_j). \tag{9.3}$$

This definition leads to symmetric neighbor lists where a particle inside the support radius of another will always appear in the neighbor list of that second particle, regardless of its support radius.

In the following, we enumerate the changes required for our method to effectively handle variable particle support radii:

1. We define the *octree node support radius* $r_{\mathcal{T}} \in \mathbb{R}$ to be the maximum support radius of all the particles an octree node $\mathcal{T}$ contains. Then, the extended octree node domain $\widetilde{\Omega}_{\mathcal{T}} \subset \mathbb{R}^3$ is the extension of $\Omega_{\mathcal{T}}$ by the parent node $r_{\mathcal{T}}$ in all coordinate directions.

2. A cell c is assigned *exterior cell* of an octree node $\mathcal{T}$ if its domain $\Omega_c \subset \mathbb{R}^3$ intersects the extended octree node search domain $\widetilde{\Omega}_{\mathcal{T}}$.

3. In the brute force stage, the particle support radii are fetched along with their positions and Eq. (9.3) is used to find neighbors.

9.3.5 Discussion

Up to this point our method has been presented in detail. In the next section we will validate it with experiments and compare it with the CLL method introduced by Band et al. [2020], which, to the best of our knowledge, was the fastest solution to the fixed-radius neighborhood search problem for CPUs previous to our work. However, to fully understand those results we first need to discuss the conceptual differences between the CLL method and our approach.

We claim, and show in our experiments, that our method is significantly faster than the CLL method. This may seem counter-intuitive since the CLL method does perform fewer pair-wise comparisons, yet the reason is straightforward: using an octree to decompose the problem into smaller brute force subproblems (clusters of particles) results in more balanced brute force tasks which contain the target amount of particles that optimizes computational throughput.

By construction, uniform grid methods lock themselves into a fairly fixed amount of work per brute force task: a center cell, and its 26 neighbor cells. Since these methods set the cell size equal to the support radius, and for a typical SPH support radius of twice the particle sampling distance, each cell has approximately 8 particles, which makes a total of 216 particles for the 27 cells in each brute force task. This is assuming perfectly packed particle distributions which is not always the case in SPH simulations, e.g., when having thin fluid layers or spray

particles. The problem is that, besides not being well balanced, these tasks do not necessarily have the optimal size to solve the problem in the most efficient way. All the brute force preparation work, mainly searching the cells and fetching the particle data, has to be carried out regardless of the amount of particles in the task. However, there is an optimal number of particles for a brute force task which balances its quadratic complexity in regards of the pair-wise comparisons with the overhead from setting up the brute force itself. As shown in Section 9.4, we find the optimal task size to be around 1000 particles in our benchmark conditions, which is much more than the 216 particles that uniform grid methods operate on per task. The benefits of larger (or rather not very small) tasks go beyond reducing the setup overhead of the brute force, better cache and SIMD utilization being the most important ones.

Another important idea we introduce is challenging the concept that the best cell size for the background uniform grid is the support radius. As described in the previous section, we use a uniform grid in our method as a problem reduction step. Since we want to generate larger brute force tasks in comparison to uniform grid methods, we do not need such small cells anymore. Larger cells, means fewer cells and more particles per cell. Having fewer cells reduces the time to build the octree and the amount of cells at the octree leaves. More particles per cell means that we load particles in larger batches, which better amortizes memory accesses. In fact, our experiments show that an optimal cell size in our benchmark conditions is one that results in about 30 particles per cell on average.

9.4 Results

In this section we discuss several experiments for our neighborhood search approach. Moreover, we compare our method with the state-of-the-art Cell-Linked-Lists method (CLL) [Band et al. 2020] which is currently the fastest known fixed-radius neighborhood search solution. To ensure a fair comparison, we included our optimizations (bruteforce, branchless SIMD pushbacks, neighbor lists layout etc.) into our CLL implementation. The CLL method presented by Band et al. [2020] lacks a description of low level optimizations. Not including our optimizations would have resulted in significantly worse results for the CLL method. However, we did not use variable cell sizes in our CLL solution because using the support radius as cell size is one of, if not the most, fundamental design choices of that method. Additionally, we show that it is possible to use our modified method in multi-resolution SPH simulations while keeping the neighborhood search runtime significantly lower than the other components of the SPH solver.

All benchmarks were performed on an Intel Core i9-9900K processor with 8 physical cores at 3.60 GHz. We used AVX2 instructions to implement the SIMD vectorization. To run the simulations, we integrated our neighborhood search method in the open-source SPH framework SPlisHSPlasH [Bender et al. 2026]. In all benchmarks we used the Divergence Free SPH pressure solver [Bender and Koschier 2017] and an implicit boundary representation [Bender et al. 2020]. In the Beach Scene (see Figure 9.7), we employ the method by Bender et al. [2018] to enhance the turbulent behavior of the water. In the multi-resolution experiments, we extended the pressure solver following the approach of Adams et al. [2007] using averaged kernel values for particles at the interface of the different resolutions. In all experiments we used the cubic spline kernel [Monaghan 1992] and the support radius was twice the initial inter-particle sampling distance.

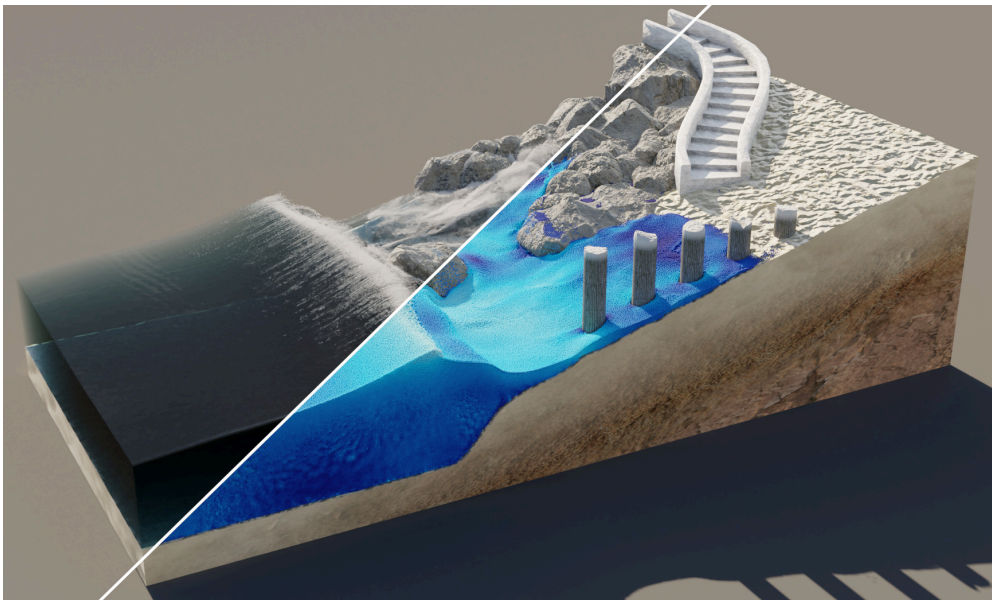


Figure 9.7: *Beach Scene. A wave generator (not rendered) pushes 9 million particles to a sloped beach shore with obstacles. Surface reconstruction on the left. Particle view on the right.*

9.4.1 Fixed-Radius Neighborhood Search

We will use two scenes for these experiments. The first one, the Beach Scene (see Figure 9.7), consists of 9 million particles pushed by a wave generator to an inclined shore. The goal of this scene is to represent high particle counts in a relatively dense distribution. In the second scene for the fixed radius experiments, the Fountain Scene (see Figure 9.8), up to 3.5 million particles are emitted into a fountain with pools and waterfalls. This scene represents simulations with increasing amounts of particles over time where the particles are more distributed over the simulation domain.

Octree Calibration

As discussed in Section 9.3.5, our method can generate brute force tasks with an approximate target number of particles, which makes it possible for it to be tuned to specific hardware conditions. For this experiment we take a representative frame of the Beach Scene and one from the Fountain Scene and study the effect of the uniform grid *cell size* and the number of particles for which we stop the recursion and generate a brute force task, which we call *cap*. We benchmark each parameter combination for each simulation, normalize the respective results so the fastest combination is 1.0 and finally add and renormalize both benchmarks into a final result in Figure 9.9. The results for the individual simulations are in any case very similar.

We can see that the region of optimal values lies for caps between 500 and 1500 and cell sizes between 1.25× and 1.75× the particle support radius. For the rest of the experiments, we use a cap value of 1000 and a cell size of 1.5× the particle support radius. In the Beach Scene, cells contain 30.59 particles on average.

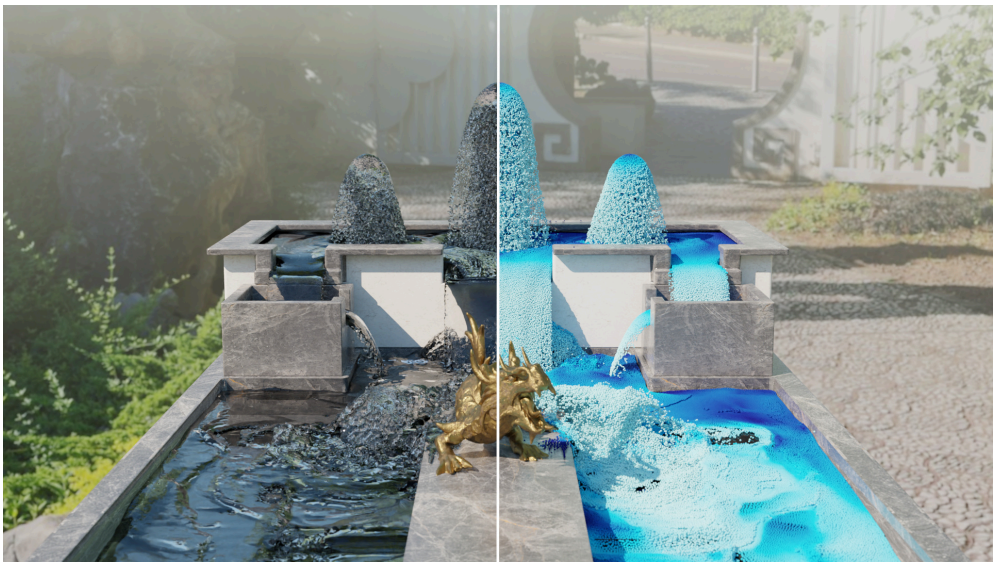


Figure 9.8: *Fountain Scene. Fluid particles are emitted upwards at the top of a fountain with pools and waterfalls. The number of particles increases up to 3.5 million. Surface reconstruction on the left. Particle view on the right.*

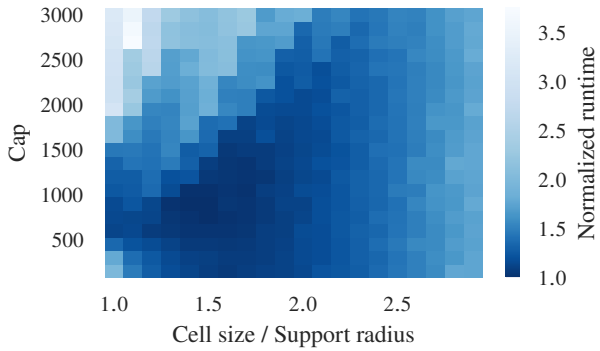


Figure 9.9: *Normalized timings for a combination of cap and cell size parameters of the octree for the Beach and Fountain scenes. The experiment shows that the optimal values for our benchmark conditions are a cap of 1000 particles and a cell size of* 1.5× *the particle support radius.*

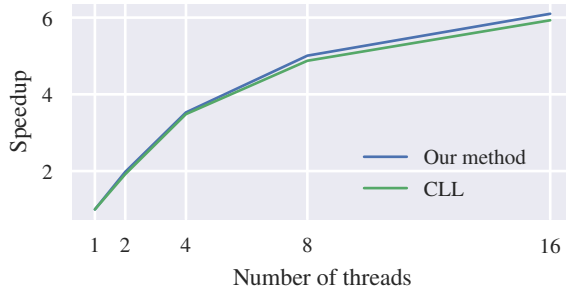


Figure 9.10: *Parallel scalability comparison between our method and the CLL method for the Beach Scene.*

Table 9.1: *CPU metrics generated with* `perf` *for our method and the CLL method on the Beach Scene. The table shows the averaged values of 100 benchmark runs.*

	Our method		CLL	
	scalar	SIMD	scalar	SIMD
Instructions (mil.)	8143	1772	4345	2715
Cache misses (mil.)	10	7	12	11
Branch misses (mil.)	64	2	72	18

Parallel scalability

We use the Beach Scene to investigate the parallel scalability of our method in comparison to the CLL method. The results in Figure 9.10 show that our method scales slightly better in comparison to our implementation of the CLL method.

SIMD and parallelism

In this experiment, we compare our method and the CLL method on the Beach Scene in relation to their CPU usage.

In Table 9.1 we can see CPU metrics gathered with the performance analyzing tool `perf` for both methods. In general we can see the large impact SIMD optimized routines have in both methods: Much fewer total instructions, better cache utilization and greatly reduced branch mispredictions. However, we can observe that our method experiences larger benefits than the CLL method, which is a direct consequence of our acceleration structure being better suited to SIMD optimizations and also to being able to adapt the brute force tasks to optimal sizes for the hardware.

In Table 9.2 we can see runtimes for sequential, parallel, scalar and SIMD modes of both methods on the Beach Scene. While both algorithms scale almost perfectly with the number of cores up to 8, we can see again how our octree takes much better advantage of SIMD instructions. As it is shown in the table, our method is actually slower in scalar mode. This is due to our brute force tasks being larger, and therefore more pair-wise comparisons must be done than in the CLL method. However, when we vectorize the loops removing most branches, we can make much better use of the hardware, resulting in a net improvement over the CLL method.

Table 9.2: *Scalar/SIMD and sequential/parallel runtime comparison between our method and the CLL method. Speedup factor with respect to the respective sequential scalar timing in parentheses.*

	Our method	CLL
Sequential Scalar	12.87 s (×1.0)	8.47 s (×1.0)
Sequential SIMD	1.64 s (×7.9)	3.15 s (×2.7)
Parallel Scalar	1.55 s (×8.3)	1.02 s (×8.3)
Parallel SIMD	0.26 s (×49.0)	0.52 s (×16.2)

Table 9.3: *Timings for the different stages of our method in scalar and SIMD mode on the Beach Scene averaged to a single time step.*

	Scalar		SIMD		
Stage	Time	%	Time	%	Speedup
Particles to cells	0.007 s	0.4%	0.003 s	1.3%	2.0×
Octree	0.060 s	3.9%	0.010 s	4.0%	5.7×
Brute force	1.480 s	95.7%	0.248 s	94.7%	6.0×
Total	1.547 s	100.0%	0.262 s	100.0%	5.9×

Stages of our method

Table 9.3, shows the runtime for the main stages of our method in scalar and SIMD mode. A remarkable result is that our acceleration structure, constituted by the *particles to cells* and the *octree* stages, takes up only 5.3% of the total neighborhood search runtime. The rest, 94.7%, of the execution time is spent in the highly optimized brute force stage, 28.26% of which is gathering the relevant particle data.

Z-sort

As discussed in Section 9.3.3, every few time steps we z-sort the particles with the help of the last updated cell list. For the Beach Scene, sorting the cells to generate the new particle order takes 4.94 ms on average while actually sorting the particles directly takes 83.0 ms. We found that updating the particle data order every 10 steps is frequent enough while being unnoticeable in relation to the whole operation of the SPH solver.

Dense vs. Sparse particle distributions

In this experiment we compare our method and the CLL method in two synthetic scenes, one with a dense particle distribution and one with a sparse particle distribution. In the first case, particles are sampled with the SPH sampling distance used in the other experiments which results in 30 neighbors on average. In the second case, particles will be sampled slightly further apart than support radius so particles do not have neighbors. In the latter setting, the cell size in both methods is chosen to be the support radius so that each cell contains just one particle. The results can be seen in Figure 9.11.

In general, we can see quite consistent runtimes when scaling up the number of particles, from 175 thousand to up to 27 million, for both methods. Our method solves the dense problem 1.65× faster than the sparse one. The CLL method suffers much more since it fails to adapt

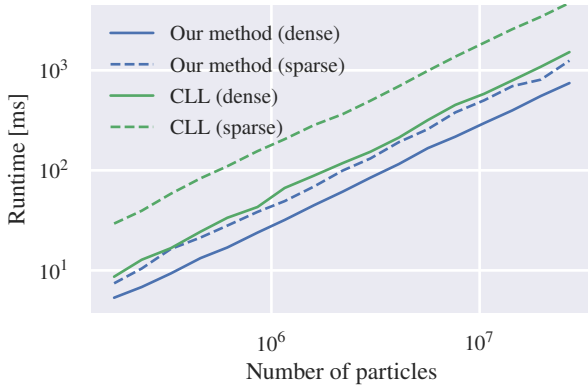


Figure 9.11: *Problem size scalability comparison for a dense (solid lines) and a sparse (dashed lines) particle distributions.*

the task sizes and it can solve the dense problem 3.24× faster than the sparse one. Our method was on average 1.95× faster than CLL for the dense problem.

Time series

In Figure 9.12, we compare the total neighborhood search runtime per iteration between our method and the CLL method for the Beach and Fountain scenes. In the Beach Scene, both methods have very consistent execution times throughout the simulation. Our method presents a speedup of 1.87× over the CLL method. On the other hand, in the Fountain Scene, the speedup ranges from 1.6× to 2.2×, with an average of 1.9×. This evolution is due to the increasing number of particles and their changing distribution in space over time. To demonstrate that the larger cell size alone is not responsible for the observed speedup, we additionally run the experiment setting the cell size to the support radius. Our method is still 1.6× faster on average in this case.

For reference, in the Beach Scene our neighborhood search took on average 5.6% (258 ms) of the total simulation time (4587 ms) per simulation step. Also in the Beach Scene, the extra memory that our method needed over the CLL approach, the space required to allocate the octree, was less than 40 MB. The particle positions themselves took 103 MB.

9.4.2 Multi-Resolution Neighborhood Search

We demonstrate the applicability of our neighborhood search method in the context of multi-resolution SPH simulations.

Comparison with CLL for variable support radii

To show the need for special handling when the support radius is not globally constant, we compare the CLL method against our method which can handle different support radii. This experiment consists of two large blocks of fluid particles side by side sampled regularly on a grid. One of the blocks contains 1 million small particles, with sampling distance of d_0 and a corresponding support radius of $r_0 = 2d_0$. For the other coarser grid, we use a sampling

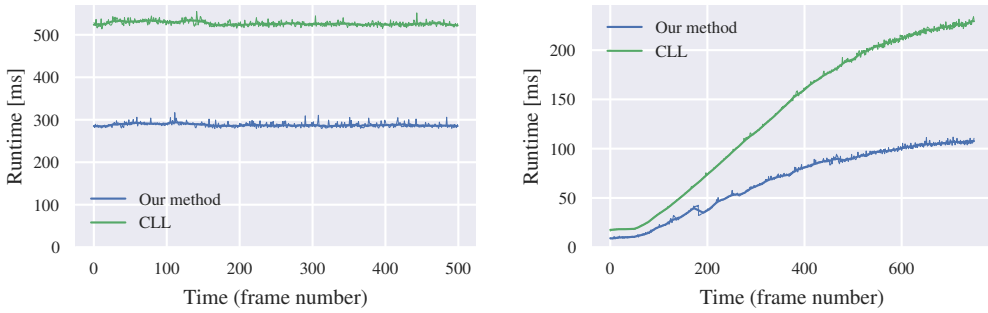


Figure 9.12: *Comparison between the total runtime of our method and the CLL method for the Beach (left) and Fountain (right) scenes. The moving average has been overlaid to the original data for clarity.*

Table 9.4: *Runtime comparison between the CLL method and our method for multi-resolution simulations. Since the CLL method has to globally use the largest support radius, its runtime scales very poorly in multi-resolution simulations in comparison to our method.*

Scale ratio (α)	1.0×	1.5×	2.0×	2.5×	3.0×
Fine particles	1.00 M	1.00 M	1.00 M	1.00 M	1.00 M
Coarse particles	1.00 M	0.29 M	0.12 M	0.06 M	0.04 M
CLL	0.109 s	0.111 s	0.232 s	1.328 s	3.961 s
Ours	0.065 s	0.051 s	0.047 s	0.046 s	0.049 s

distance $d_1 = \alpha d_0$ with a corresponding support radius of $r_1 = 2\alpha d_0$, where α is the scale ratio for the particle sampling and support radius between the two grids. Particles fully inside their own grids have, on average, approximately 30 neighbors.

Since the CLL method does not support different support radii, the largest radius must be used globally. As α grows, the small particles in the fine grid will find more and more neighbors. However, these extra neighbors are not needed for the small particles since their support radius is smaller than the one used by the CLL method. This increase in the number of unwanted neighbors is cubic in the support radius and, as we can see in Table 9.4, it slows down the neighborhood search very significantly as the support radius ratio increases.

Multi-Resolution, Single Phase

In Figure 9.13, we show the classic double dam break simulation, in this case with a center obstacle, but in the multi-resolution setting. We simulate a single fluid phase, the liquid, but we employ two different particle resolutions, each of them with a constant particle radius. The fine particle set has a slightly lower density so it settles on top after the first splash to provide high detail fluid motions while the coarse particle set stays below. The fine particle set contains 3.6 million particles of 5 mm particle radius. The coarse set has 450 thousand particles of 10 mm particle radius, twice the radius of the fine set.

Multi-Resolution, Fluid-Solid

In Figure 9.14, we show an example of a multi-resolution simulation with one fluid phase and three elastic objects solved with the method by Kugelstadt et al. [2021]. By using fewer

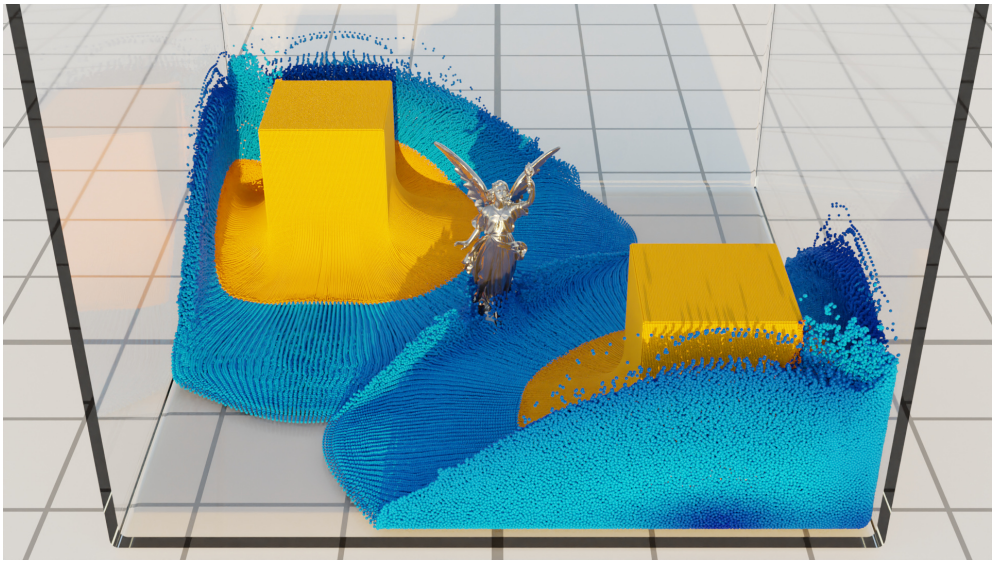


Figure 9.13: *Multi-resolution single-phase double dam break scene with the Stanford Lucy statue as obstacle in the center. Two fluid particle sets with different radii are used: 450 thousand coarse particles (blue color map) and 3.6 million fine particles (yellow color map). The small particles have half the radius of the large ones.*

particles in the elastic bodies, we can alleviate the main drawback of the elastic solver, the quadratic memory complexity and initialization times. Having a coarser discretization for the elastic objects with respect to the fluid does not have a noticeable impact in the result after the surface reconstruction. There are 5.6 million particles of 10 mm particle radius in the fluid phase and a total of 12 thousand particles of 25 mm in the solid phase, a factor of 2.5 larger.

Multi-Resolution, Two Phase

Another use case for SPH simulations with different resolution particles is the simulation of fluids with different viscosities, modeled using the method by Weiler et al. [2018], in different resolutions. In Figure 9.15, we show a two-phase simulation where a highly viscous material experiences deformations at a much lower frequency than a low viscous fluid. In this setting, we can simulate the low viscous fluid with much higher particle resolution and save computational effort on the highly viscous one. The low viscous fluid is discretized with up to 5 million particles of 8.4 mm particle radius and the highly viscous one with 90 thousand particles of a particle radius of 21 mm, a factor of 2.5 larger radius.

In all non-fixed radius simulations, the relative runtime of our neighborhood search method ranged between 5% and 10% of the total simulation time.

9.5 Limitations

Our method inherits the usual corner cases of octrees. The recursive uniform subdivision of the octree can lead to linear chains in the tree where all particles repeatedly end up in the same

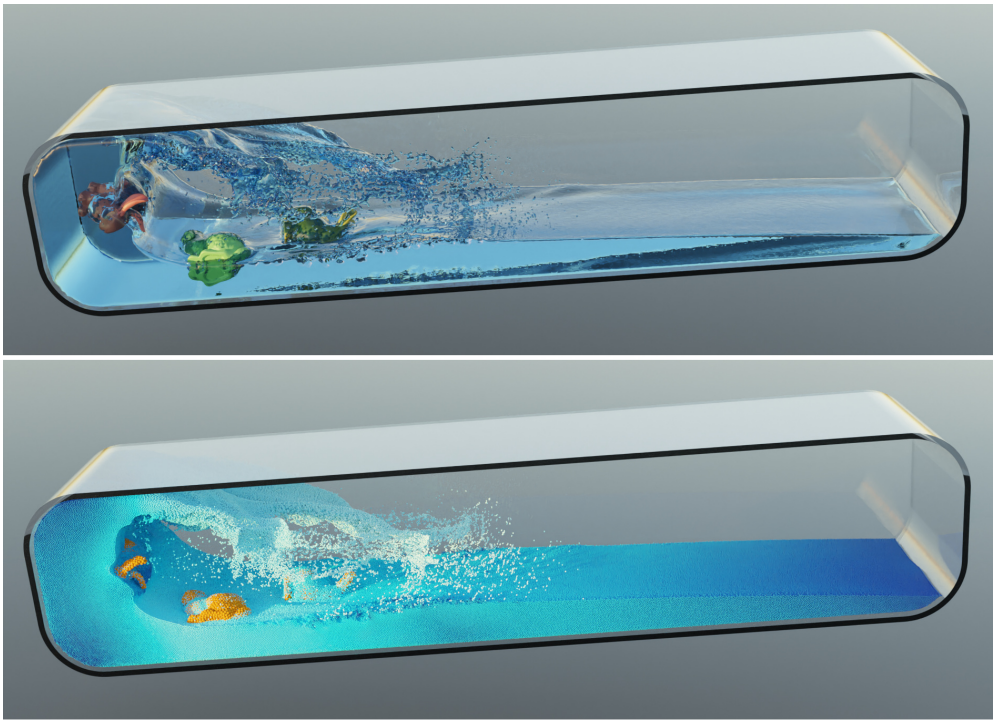


Figure 9.14: *Multi-resolution fluid-solid simulation of a high-resolution fluid and low-resolution deformable Stanford bunnies. A rigid box swings with 5.6 million fluid particles and a total of 12 thousand solid particles inside. Particles belonging to the deformable bodies have a* 2.5× *larger radius than the ones of the fluid phase. Surface view on the top, particle view on the bottom.*

child node. In simulation settings, this realistically only happens in cases where there is a very large distance between groups of particles. This would usually only cost a modest number of traversals over the particle data. However, if there are local instabilities in the simulation, it is possible for single particles to rapidly travel large distances away from the intended simulation domain. This is easily remedied by ignoring particles far outside of some predefined region of interest during the uniform grid construction.

It is also possible to detect and skip linear chains during the octree construction by jumping ahead to the smallest octree node that is large enough to contain the extended domains of all cells of the current octree node. We did not find it necessary to explicitly handle this problem for any of our simulations.

Our method may also be suboptimal for large ratios in support radii in certain configurations. For example, if most particles have a much greater radius than the particle with the smallest radius, then many grid cells will only contain a single particle with a large radius, reducing the effectiveness of the particle-to-cell aggregation. However, this is not the case in a typical SPH simulation, and our focus was in maximizing the performance on the fixed-radius case.

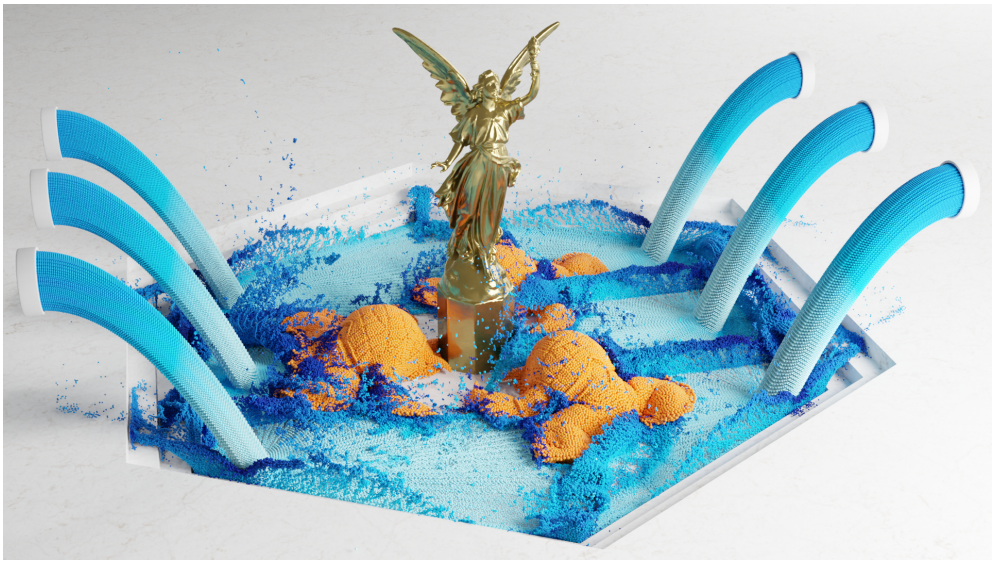


Figure 9.15: *Multi-resolution two-phase simulation of high- and low-viscosity fluids. Six emitters pour up to 5 million fluid particles into a pool with a static rigid body (Stanford Lucy) and three highly viscous Stanford Armadillos. The 90 thousand particles of the viscous material have a* 2.5× *larger radius than the fluid particles.*

9.6 Conclusion and Future Work

Neighborhood search is a critical component in SPH simulations. We presented an approach to the neighborhood search problem that departs from the long-standing assumption that the optimal cell size to use in the neighborhood search is the kernel support radius. We showed that, due to the ability to generate more balanced tasks and to better utilize the CPU, octrees are better suited than uniform grids, even for fixed-radius neighborhood search problems. In addition, we show that uniform grids, which we also use as a problem reduction step, are more effective when the cell size is not chosen to be the kernel support radius, but it is instead tuned to maximize computation throughput during the brute force stage. We included discussions and extensive experimentation to show the behavior of our method in comparison to a state-of-the-art method for fixed-radius neighborhood search. Our method achieved a speedup of up to 1.9× in real-world large SPH simulations in comparison to the well-established Cell-Linked-List method, taking up around 5% of the total SPH simulation runtime.

Additionally, we show that it is possible to exploit the adaptive nature of our octree acceleration structure to enhance the method to support variable support radii. We use this version of our neighborhood search in multi-resolution SPH simulations, including fluid-solid interaction and highly viscous fluids, with radius ratios of up to 3. The runtime of the neighborhood search was 10% or less of the total simulation runtime.

Part IV

Conclusions and Outlook

CHAPTER 10

Concluding Remarks

10.1 Conclusion

This thesis studied differentiation and high-performance infrastructure for multiphysics simulation. The central viewpoint is that modern physical simulation demands infrastructure that addresses robustness, differentiability, and performance together. Optimization-based simulation provides a common language for this infrastructure by expressing inertia, elasticity, contact, friction, constraints, and actuation as contributions to a global objective. Once written in this form, a simulator needs reliable derivatives, sparse assembly, robust nonlinear solvers, sensitivity analysis, dynamic data structures, and hardware-aware kernels. The contribution chapters developed these ingredients at different layers of the simulation stack and answered the research questions posed in Section 1.1.

SymX, presented in Chapter 5, addressed RQ1 by representing complex multiphysics simulations with concise symbolic expressions and automating differentiation, compilation, and assembly. This was shown to reduce the implementation cost of complex variational models by effectively exploiting the local-to-global structure of the problem. The chapter showed heterogeneous object representations, multiple materials, and challenging frictional contact scenarios expressed in short symbolic forms. This infrastructure carries forward to the following chapters for strong coupling and inverse simulation.

STARK, presented in Chapter 6, addressed RQ2 by integrating rigid bodies, deformable solids, cloth, shells, joints, motors, contact, and friction in one strongly coupled variational simulator. As a result, it provides a simulator stack for challenging robotics-adjacent scenarios, especially robot-deformable interaction and manipulation. The robot-towel experiments demonstrate the value of this coupling by reproducing laboratory outcomes in challenging contact-rich scenes.

Progressively Projected Newton's Method, presented in Chapter 7, addressed RQ3 by improving the solver layer. PPN showed that robust Newton-type simulation does not require every local Hessian to be projected at every iteration. By starting from the unmodified Hessian and projecting only selected element contributions when indefiniteness is detected, PPN greatly reduces eigendecomposition cost and preserves more useful curvature information. The method therefore improves the efficiency of second-order simulation pipelines such as those enabled by the two previous chapters.

IFA, presented in Chapter 8, addressed RQ4 by applying differentiable shell simulation to interactive facial animation. The simulation infrastructure is extended with adjoint-based inverse simulation to reconstruct non-uniform shell stiffness fields, allowing a coarse simulation mesh to reproduce an existing rig within a strict tolerance. The fitted model then adds real-time contact response while preserving the high-resolution appearance of the original rig through warping and fine-detail correction. This chapter shows how differentiable simulation can be layered on top of symbolic representation, and how inverse simulation can learn from and enhance data-driven animation.

TreeNSearch, presented in Chapter 9, addressed RQ5 by proposing hardware-aware data structures and algorithms for proximity detection in particle-based fluid simulation. The method challenges the assumption that minimizing candidate pair counts is sufficient for high performance. By using an octree to create SIMD-friendly and well-balanced brute-force tasks, it improves CPU throughput and extends naturally to variable support radii. This contribution broadens the thesis beyond math-focused infrastructure and shows that data layout, branching, task size, and memory traffic are also central to simulation performance.

Taken together, the contributions show that problem formulation, mathematical models, simulation infrastructure, and performance are tightly connected. With the right foundations and tools, complex problems become more accessible to the people carrying out the research and more efficient for the machines executing the solves. Improving performance while reducing the burden of testing, maintenance, and low-level optimization frees researchers from work that distracts from scientific discovery. As a result, infrastructure improves the scientific cycle by reducing the time between insight, implementation, and evaluation. This compounds human effectiveness since more ideas can be tested, compared, and refined within the same time span.

10.2 Outlook

The results of this thesis suggest several directions beyond the specific applications addressed in the contribution chapters.

Richer symbolic representations. SymX currently represents expressions through a scalar symbolic engine. This is flexible, but many mechanical models are naturally written in vector, matrix, or tensor notation. Tensor-aware symbolic rules could preserve this structure longer, apply tensor identities directly, and produce more compact derivatives. They could also reduce generated-code size by emitting compact loops for tensor contractions instead of fully unrolled scalar arithmetic. Together with stronger expression simplification, such representations would further reduce the gap between the notation used to design a model and the kernels used to run it.

Heterogeneous execution. Another natural direction is GPU and mixed CPU-GPU execution for generated kernels, sparse assembly, and nonlinear solves. The evaluation of local energies and derivatives is well suited to massive parallelism, and code generation could target device kernels in addition to CPU code. The harder challenge is preserving the flexibility expected from research simulators. Line search, continuous collision detection, inversion checks, residual definitions, preconditioners, and application-specific callbacks often require user code that does not fit neatly into a fully generated GPU pipeline. As such, while parts of

the pipeline could be made more efficient, scientists writing simulation software would have to choose between writing GPU kernels for custom needs or paying the synchronization cost at each callback.

Broader variational mechanics. The rigid body formulation used in this thesis follows a compact variational model for generalized rigid body dynamics [Macklin et al. 2020]. Recent work on rigid and affine body simulation, intersection-free rigid dynamics, and differentiable rotation dynamics suggests richer mechanical models that could be integrated into the same infrastructure [Ferguson et al. 2021; Lan et al. 2022; Romanyà-Serrasolsas et al. 2025]. Higher-order spatial and temporal representations provide a related opportunity. They can improve accuracy, reduce locking, and enable coarser discretizations, as demonstrated for embedded finite elements [Longva et al. 2020]. While symbolic representations significantly lower the barrier to entry, several parts of the pipeline remain difficult, such as high-order collision detection and handling.

Dynamic differentiable simulation. This thesis discusses static differentiability via the adjoint method. This is sufficient for many parameter-identification and structural optimization problems, including the facial-stiffness reconstruction in Chapter 8. Many robotics problems, however, depend essentially on dynamics. Balance during locomotion, reaching and grasping, friction-sensitive manipulation, and contact-rich control all depend on the timing of forces and impacts. Extending the adjoint infrastructure from isolated equilibria to dynamic trajectories would therefore open a path toward optimizing shapes, controls, actuation, and interaction parameters in settings where motion itself is the object of study.

Bibliography

Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: A System for Large-Scale Machine Learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, pages 265–283, 2016.

Bart Adams, Mark Pauly, Richard Keiser, and Leonidas J. Guibas. Adaptively Sampled Particle Fluids. *ACM Transactions on Graphics*, 26(3):48, 2007. doi:10.1145/1276377.1276437.

Martin Alnæs, Jan Blechta, Johan Hake, August Johansson, Benjamin Kehlet, Anders Logg, Chris Richardson, Johannes Ring, Marie E Rognes, and Garth N Wells. The FEniCS Project Version 1.5. *Archive of Numerical Software*, 3(100), 2015.

Joel A E Andersson, Joris Gillis, Greg Horn, James B Rawlings, and Moritz Diehl. CasADi – A Software Framework for Nonlinear Optimization and Optimal Control. *Mathematical Programming Computation*, 11(1):1–36, 2019. doi:10.1007/s12532-018-0139-4.

Yunfei Bai and C. Karen Liu. Coupling Cloth and Rigid Bodies for Dexterous Manipulation. In *7th Int. Conf. on Motion in Games*, 2014.

Stefan Band, Christoph Gissler, and Matthias Teschner. Compressed Neighbour Lists for SPH. *Computer Graphics Forum*, 39(1):531–542, 2020. doi:10.1111/cgf.13890.

Jernej Barbič. Exact Corotational Linear FEM Stiffness Matrix. Technical report, University of Southern California, 2012.

V. Barrielle and N. Stoiber. Realtime Performance-Driven Physical Simulation for Facial Animation. *Computer Graphics Forum*, 38(1):151–166, 2019. doi:10.1111/cgf.13450.

Vincent Barrielle, Nicolas Stoiber, and Cédric Cagniart. BlendForces: A Dynamic Framework for Facial Animation. *Computer Graphics Forum*, 35(2):341–352, 2016. doi:10.1111/cgf.12836.

Jan Bender and Dan Koschier. Divergence-Free SPH for Incompressible and Viscous Fluids. *IEEE Transactions on Visualization and Computer Graphics*, 23(3):1193–1206, 2017. doi:10.1109/TVCG.2016.2578335.

Jan Bender, Matthias Müller, Miguel A Otaduy, and Matthias Teschner. Position-based Methods for the Simulation of Solid Objects in Computer Graphics. In *Eurographics (State of the Art Reports)*, pages 1–22, 2013.

Jan Bender, Kenny Erleben, and Jeff Trinkle. Interactive Simulation of Rigid Body Dynamics in Computer Graphics. *Computer Graphics Forum*, 33(1):246–270, 2014. doi:10.1111/cgf.12272.

Jan Bender, Dan Koschier, Tassilo Kugelstadt, and Marcel Weiler. Turbulent Micropolar SPH Fluids with Foam. *IEEE Transactions on Visualization and Computer Graphics*, 25(6): 2284–2295, 2018.

Jan Bender, Tassilo Kugelstadt, Marcel Weiler, and Dan Koschier. Implicit Frictional Boundary Handling for SPH. *IEEE Transactions on Visualization and Computer Graphics*, 26(10): 2982–2993, 2020. doi:10.1109/TVCG.2020.3004245.

Jan Bender et al. SPlisHSPlasH Library, 2026. URL `https://github.com/InteractiveComputerGraphics/SPlisHSPlasH`.

Miklós Bergou, Max Wardetzky, David Harmon, Denis Zorin, and Eitan Grinspun. A Quadratic Bending Model for Inextensible Surfaces. In *Proceedings of the Eurographics Symposium on Geometry Processing*, pages 227–230, 2006. doi:10.2312/SGP/SGP06/227-230.

Gilbert Louis Bernstein, Chinmayee Shah, Crystal Lemire, Zachary Devito, Matthew Fisher, Philip Levis, and Pat Hanrahan. Ebb: A DSL for Physical Simulation on CPUs and GPUs. *ACM Transactions on Graphics*, 35(2), 2016. doi:10.1145/2892632.

Bernd Bickel, Mario Botsch, Roland Angst, Wojciech Matusik, Miguel Otaduy, Hanspeter Pfister, and Markus Gross. Multi-scale Capture of Facial Geometry and Motion. *ACM Transactions on Graphics*, 26(3):33–es, 2007. doi:10.1145/1276377.1276419.

Bernd Bickel, Manuel Lang, Mario Botsch, Miguel A. Otaduy, and Markus Gross. Pose-space Animation and Transfer of Facial Details. In *Proceedings of the 2008 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pages 57–66, 2008.

J. Blinn. Consider the Lowly 2 x 2 Matrix. *IEEE Computer Graphics and Applications*, 16 (2):82–88, 1996. doi:10.1109/38.486688.

Sofien Bouaziz, Sebastian Martin, Tiantian Liu, Ladislav Kavan, and Mark Pauly. Projective Dynamics: Fusing Constraint Projections for Fast Simulation. *ACM Transactions on Graphics*, 33(4), 2014. doi:10.1145/2601097.2601116.

George E. Brown, Matthew Overby, Zahra Forootaninia, and Rahul Narain. Accurate Dissipative Forces in Optimization Integrators. *ACM Transactions on Graphics*, 37(6):1–14, 2018. doi:10.1145/3272127.3275011.

Prashanth Chandran and Gaspard Zoss. Anatomically Constrained Implicit Face Models . In *2024 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 2220–2229, 2024. doi:10.1109/CVPR52733.2024.00216.

Isaac Chao, Ulrich Pinkall, Patrick Sanan, and Peter Schröder. A Simple Geometric Model for Elastic Deformations. *ACM Transactions on Graphics*, 29(4), 2010. doi:10.1145/1778765.1778775.

Anka He Chen, Ziheng Liu, Yin Yang, and Cem Yuksel. Vertex Block Descent. *ACM Transactions on Graphics*, 43(4):1–16, 2024a. doi:10.1145/3658179.

Honglin Chen, Hsueh-Ti Derek Liu, Alec Jacobson, David I.W. Levin, and Changxi Zheng. Trust-Region Eigenvalue Filtering for Projected Newton. In *SIGGRAPH Asia 2024 Conference Papers*, 2024b. doi:10.1145/3680528.3687650.

Honglin Chen, Hsueh-Ti Derek Liu, David I.W. Levin, Changxi Zheng, and Alec Jacobson. Stabler Neo-Hookean Simulation: Absolute Eigenvalue Filtering for Projected Newton. In *Special Interest Group on Computer Graphics and Interactive Techniques Conference Conference Papers '24*, pages 1–10, 2024c. doi:10.1145/3641519.3657433.

Yunuo Chen, Minchen Li, Lei Lan, Hao Su, Yin Yang, and Chenfanfu Jiang. A Unified Newton Barrier Method for Multibody Dynamics. *ACM Transactions on Graphics*, 41(4), 2022. doi:10.1145/3528223.3530076.

Yunuo Chen, Tianyi Xie, Cem Yuksel, Danny Kaufman, Yin Yang, Chenfanfu Jiang, and Minchen Li. Multi-Layer Thick Shells. In *ACM SIGGRAPH 2023 Conference Proceedings*, pages 1–9, 2023. doi:10.1145/3588432.3591489.

Yuan-Yuan Cheng, Ligang Liu, and Xiao-Ming Fu. Eigenvalue Blending for Projected Newton. *Computer Graphics Forum*, 2025. doi:10.1111/cgf.70027.

Byungkuk Choi, Haekwang Eom, Benjamin Mouscadet, Stephen Cullingford, Kurt Ma, Stefanie Gassel, Suzi Kim, Andrew Moffat, Millicent Maier, Marco Revelant, Joe Letteri, and Karan Singh. Animatomy: An Animator-Centric, Anatomically Inspired System for 3D Facial Modeling, Animation and Transfer. In *SIGGRAPH Asia 2022 Conference Papers*, pages 1–9, 2022. doi:10.1145/3550469.3555398.

Matthew Cong, Kiran S. Bhat, and Ronald Fedkiw. Art-directed Muscle Simulation for High-end Facial Animation. In *Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pages 119–127, 2016.

Coppelia Robotics, Ltd. CoppeliaSim, 2026. URL `https://www.coppeliarobotics.com`.

Stelian Coros, Miles Macklin, Bernhard Thomaszewski, and Nils Thürey. Differentiable Simulation. In *SIGGRAPH Asia 2021 Courses*, 2021. doi:10.1145/3476117.3483433.

Erwin Coumans and Yunfei Bai. PyBullet, a Python Module for Physics Simulation for Games, Robotics and Machine Learning, 2026. URL `http://pybullet.org`.

Cyberbotics Ltd. Webots, 2026. URL `http://www.cyberbotics.com`.

Deshana Desai, Etai Shuchatowitz, Zhongshi Jiang, Teseo Schneider, and Daniele Panozzo. ACORNS: An Easy-to-use Code Generator for Gradients and Hessians. *SoftwareX*, 17: 100901, 2022. doi:10.1016/j.softx.2021.100901.

Mathieu Desbrun and Marie-Paule Cani. *Space-Time Adaptive Simulation of Highly Deformable Substances*. PhD thesis, INRIA, 1999.

Zachary DeVito, Niels Joubert, Francisco Palacios, Stephen Oakley, Montserrat Medina, Mike Barrientos, Erich Elsen, Frank Ham, Alex Aiken, Karthik Duraisamy, Eric Darve, Juan Alonso, and Pat Hanrahan. Liszt: A Domain Specific Language for Building Portable Mesh-Based PDE Solvers. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, 2011. doi:10.1145/2063384.2063396.

Zachary DeVito, Michael Mara, Michael Zollhöfer, Gilbert Bernstein, Jonathan Ragan-Kelley, Christian Theobalt, Pat Hanrahan, Matthew Fisher, and Matthias Niessner. Opt: A Domain Specific Language for Non-Linear Least Squares Optimization in Graphics and Imaging. *ACM Transactions on Graphics*, 36(5), 2017. doi:10.1145/3132188.

Ana Dodik, Ahmed Mahmoud, and Justin Solomon. Iskra: A System for Inverse Geometry Processing. *ACM Transactions on Graphics*, 45(4), 2026. doi:10.1145/3811371.

J. M. Domínguez, A. J. C. Crespo, M. Gómez-Gesteira, and J. C. Marongiu. Neighbour Lists in Smoothed Particle Hydrodynamics. *International Journal for Numerical Methods in Fluids*, 67(12):2026–2042, 2010. doi:10.1002/fld.2481.

Tom Erez, Yuval Tassa, and Emanuel Todorov. Simulation Tools for Model-based Robotics: Comparison of Bullet, Havok, MuJoCo, ODE and PhysX. In *IEEE Int. Conf. on Robotics and Automation (ICRA)*, 2015.

Zachary Ferguson, Minchen Li, Teseo Schneider, Francisca Gil-Ureta, Timothy Langlois, Chenfanfu Jiang, Denis Zorin, Danny M. Kaufman, and Daniele Panozzo. Intersection-free Rigid Body Dynamics. *ACM Transactions on Graphics*, 40(4):1–16, 2021. doi:10.1145/3450626.3459802.

Zachary Ferguson et al. IPC Toolkit, 2026. URL `https://ipc-sim.github.io/ipc-toolkit/`.

José Antonio Fernández-Fernández, Lukas Westhofen, Fabian Löschner, Stefan Rhys Jeske, Andreas Longva, and Jan Bender. Fast Octree Neighborhood Search for SPH Simulations. *ACM Transactions on Graphics (SIGGRAPH Asia)*, 41(6):242:1–242:13, 2022. doi:10.1145/3550454.3555523.

José Antonio Fernández-Fernández, Ralph Lange, Stefan Laible, Kai O. Arras, and Jan Bender. STARK: A Unified Framework for Strongly Coupled Simulation of Rigid and Deformable Bodies with Frictional Contact. In *2024 IEEE International Conference on Robotics and Automation (ICRA)*, pages 16888–16894, 2024. doi:10.1109/ICRA57147.2024.10610574.

José Antonio Fernández-Fernández, Ryan Goldade, Ladislav Kavan, Jan Bender, and Philipp Herholz. Interactive Facial Animation: Enhancing Facial Rigs with Real-Time Shell and Contact Simulation. *Proceedings of the ACM on Computer Graphics and Interactive Techniques*, 8(4), 2025a. doi:10.1145/3747860.

José Antonio Fernández-Fernández, Fabian Löschner, Lukas Westhofen, Andreas Longva, and Jan Bender. SymX: Energy-based Simulation from Symbolic Expressions. *ACM Transactions on Graphics*, 45(1):5:1–5:19, 2025b. doi:10.1145/3764928.

José Antonio Fernández-Fernández, Fabian Löschner, and Jan Bender. Progressively Projected Newton's Method. *Computer Graphics Forum*, 45(2):e70386, 2026. doi:10.1111/cgf.70386.

Franka Emika. Franka Hand Product Manual, 2022.

Xiao-Ming Fu and Yang Liu. Computing Inversion-free Mappings by Simplex Assembly. *ACM Transactions on Graphics*, 35(6):1–12, 2016. doi:10.1145/2980179.2980231.

Theodore Gast, Craig Schroeder, Alexey Stomakhin, Chenfanfu Jiang, and Joseph Teran. Optimization Integrator for Large Time Steps. *IEEE Transactions on Visualization and Computer Graphics*, 21(10):1103–1115, 2015. doi:10.1109/TVCG.2015.2459687.

Robert A Gingold and Joseph J Monaghan. Smoothed Particle Hydrodynamics: Theory and Application to Non-spherical Stars. *Monthly Notices of the Royal Astronomical Society*, 181(3):375–389, 1977.

Christoph Gissler, Andreas Henne, Stefan Band, Andreas Peer, and Matthias Teschner. An Implicit Compressible SPH Solver for Snow Simulation. *ACM Transactions on Graphics*, 39(4):1–16, 2020.

Simon Green. Particle Simulation Using CUDA. Technical report, NVIDIA, 2010.

Andreas Griewank and Andrea Walther. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. Society for Industrial and Applied Mathematics, second edition, 2008. doi:10.1137/1.9780898717761.

Eitan Grinspun, Anil N. Hirani, Mathieu Desbrun, and Peter Schröder. Discrete Shells. In *Proceedings of the 2003 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pages 62–67, 2003.

Julian Gross, Marcel Köster, and Antonio Krüger. Fast and Efficient Nearest Neighbor Search for Particle Simulations. In *Computer Graphics and Visual Computing (CGVC)*, 2019. doi:10.2312/CGVC.20191258.

Gaël Guennebaud, Benoît Jacob, et al. Eigen, 2010. URL `https://libeigen.gitlab.io`.

Takahiro Harada, Seiichi Koshizuka, and Yoichiro Kawaguchi. Sliced Data Structure for Particle-based Simulations on GPUs. In *Proceedings of the 5th International Conference on Computer Graphics and Interactive Techniques in Australia and Southeast Asia - GRAPHITE '07*, 2007. doi:10.1145/1321261.1321271.

Laurent Hascoet and Valérie Pascual. The Tapenade Automatic Differentiation Tool: Principles, Model, and Specification. *ACM Transactions on Mathematical Software*, 39(3):1–43, 2013. doi:10.1145/2450153.2450158.

F. Hecht. New Development in FreeFEM++. *J. Numer. Math.*, 20(3–4):251–265, 2012. URL `https://freefem.org/`.

Philipp Herholz, Xuan Tang, Teseo Schneider, Shoaib Kamil, Daniele Panozzo, and Olga Sorkine-Hornung. Sparsity-Specific Code Optimization Using Expression Trees. *ACM Transactions on Graphics*, 41(5), 2022. doi:10.1145/3520484.

Philipp Herholz, Tuur Stuyck, and Ladislav Kavan. A Mesh-based Simulation Framework Using Automatic Code Generation. *ACM Transactions on Graphics*, 43(6):1–17, 2024. doi:10.1145/3687986.

Lars Hernquist and Neal Katz. TREESPH-A Unification of SPH with the Hierarchical Tree Method. *The Astrophysical Journal Supplement Series*, 70:419–446, 1989.

Daniel Holz, Stefan Rhys Jeske, Fabian Löschner, Jan Bender, Yin Yang, and Sheldon Andrews. Multiphysics Simulation Methods in Computer Graphics. *Computer Graphics Forum*, 2025. doi:10.1111/cgf.70082.

Christopher Jon Horvath and Barbara Solenthaler. Mass Preserving Multi-Scale SPH. Technical report, University of California, Los Angeles, 2013.

Yuanming Hu, Tzu-Mao Li, Luke Anderson, Jonathan Ragan-Kelley, and Frédo Durand. Taichi: A Language for High-Performance Computation on Spatially Sparse Data Structures. *ACM Transactions on Graphics*, 38(6), 2019. doi:10.1145/3355089.3356506.

Isabella Huang, Yashraj Narang, Clemens Eppner, Balakumar Sundaralingam, Miles Macklin, Ruzena Bajcsy, Tucker Hermans, and Dieter Fox. DefGraspSim: Physics-Based Simulation of Grasp Outcomes for 3D Deformable Objects. *IEEE Robotics and Automation Letters*, 7 (3), 2022.

Kemeng Huang, Floyd M. Chitalu, Huancheng Lin, and Taku Komura. GIPC: Fast and Stable Gauss-Newton Optimization of IPC Barrier Energy. *ACM Transactions on Graphics*, 43 (2):1–18, 2024. doi:10.1145/3643028.

Alexandru-Eugen Ichim, Ladislav Kavan, Merlin Nimier-David, and Mark Pauly. Building and Animating User-specific Volumetric Face Rigs. In *Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pages 107–117, 2016.

Alexandru-Eugen Ichim, Petr Kadleček, Ladislav Kavan, and Mark Pauly. Phace: Physics-based Face Modeling and Animation. *ACM Transactions on Graphics*, 36(4):153:1–153:14, 2017. doi:10.1145/3072959.3073664.

Markus Ihmsen, Nadir Akinci, Markus Becker, and Matthias Teschner. A Parallel SPH Implementation on Multi-core CPUs. *Computer Graphics Forum*, 30:99–112, 2011. doi:10.1111/j.1467-8659.2010.01832.x.

Markus Ihmsen, Jens Cornelis, Barbara Solenthaler, Christopher Horvath, and Matthias Teschner. Implicit Incompressible SPH. *IEEE Transactions on Visualization and Computer Graphics*, 20(3):426–435, 2014a. doi:10.1109/tvcg.2013.105.

Markus Ihmsen, Jens Orthmann, Barbara Solenthaler, Andreas Kolb, and Matthias Teschner. SPH Fluids in Computer Graphics. In *Eurographics 2014 - State of the Art Reports*, pages 21–42, 2014b. doi:10.2312/EGST.20141034.

Wenzel Jakob, Sébastien Speierer, Nicolas Roussel, and Delio Vicini. Dr.Jit: A Just-in-Time Compiler for Differentiable Rendering. *ACM Transactions on Graphics*, 41(4), 2022. doi:10.1145/3528223.3530099.

Stefan Rhys Jeske, Lukas Westhofen, Fabian Löschner, José Antonio Fernández-Fernández, and Jan Bender. Implicit Surface Tension for SPH Fluid Simulation. *ACM Transactions on Graphics*, 43(1):13:1–13:14, 2023. doi:10.1145/3631936.

Kai Jia. SANM: A Symbolic Asymptotic Numerical Solver with Applications in Mesh Deformation. *ACM Transactions on Graphics*, 40(4), 2021. doi:10.1145/3450626.3459755.

Petr Kadleček and Ladislav Kavan. Building Accurate Physics-Based Face Models from Data. *Proc. ACM Comput. Graph. Interact. Tech.*, 2(2):15:1–15:16, 2019. doi:10.1145/3340256.

Ladislav Kavan, John Doublestein, Martin Prazak, Matthew Cioffi, and Doug Roble. Compressed Skinning for Facial Blendshapes. In *ACM SIGGRAPH 2024 Conference Papers*, 2024. doi:10.1145/3641519.3657477.

L. Kharevych, Weiwei Yang, Y. Tong, E. Kanso, J. E. Marsden, P. Schröder, and M. Desbrun. Geometric, Variational Integrators for Computer Animation. In *Proceedings of the 2006 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pages 43–51, 2006.

Chung Min Kim, Michael Danielczuk, Isabella Huang, and Ken Goldberg. IPC-GraspSim: Reducing the Sim2Real Gap for Parallel-Jaw Grasping with the Incremental Potential Contact Model. In *IEEE Int. Conf. on Robotics and Automation (ICRA)*, 2022.

Jungmin Kim, Min Gyu Choi, and Young J. Kim. Real-time Muscle-Based Facial Animation Using Shell Elements and Force Decomposition. In *Symposium on Interactive 3D Graphics and Games*, pages 1–9, 2020. doi:10.1145/3384382.3384531.

Theodore Kim. A Finite Element Formulation of Baraff-Witkin Cloth. *Computer Graphics Forum*, 39(8):171–179, 2020. doi:10.1111/cgf.14111.

Theodore Kim and David Eberle. Dynamic Deformables: Implementation and Production Practicalities (Now with Code!). In *ACM SIGGRAPH 2022 Courses*, pages 1–259, 2022. doi:10.1145/3532720.3535628.

Theodore Kim, Fernando De Goes, and Hayley Iben. Anisotropic Elasticity for Inversion-Safety and Element Rehabilitation. *ACM Transactions on Graphics*, 38(4), 2019. doi:10.1145/3306346.3323014.

Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. In *International Conference on Learning Representations (ICLR)*, 2015.

Fredrik Kjolstad, Shoaib Kamil, Jonathan Ragan-Kelley, David I. W. Levin, Shinjiro Sueda, Desai Chen, Etienne Vouga, Danny M. Kaufman, Gurtej Kanwar, Wojciech Matusik, and Saman Amarasinghe. Simit: A Language for Physical Simulation. *ACM Transactions on Graphics*, 35(2), 2016. doi:10.1145/2866569.

N. Koenig and A. Howard. Design and Use Paradigms for Gazebo, an Open-source Multi-robot Simulator. In *IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS)*, 2004.

Marian Körber, Johann Lange, Stephan Rediske, Simon Steinmann, and Roland Glück. Comparing Popular Simulation Environments in the Scope of Robotics and Reinforcement Learning, 2021. doi:10.48550/arXiv.2103.04616.

Dan Koschier, Crispin Deul, Magnus Brand, and Jan Bender. An hp-Adaptive Discretization Algorithm for Signed Distance Field Generation. *IEEE Transactions on Visualization and Computer Graphics*, 23(10):2208–2221, 2017. doi:10.1109/TVCG.2017.2730202.

Dan Koschier, Jan Bender, Barbara Solenthaler, and Matthias Teschner. A Survey on SPH Methods in Computer Graphics. *Computer Graphics Forum*, 41(2), 2022. doi:10.1111/cgf.14508.

Yeara Kozlov, Derek Bradley, Moritz Bächer, Bernhard Thomaszewski, Thabo Beeler, and Markus Gross. Enriching Facial Blendshape Rigs with Physical Simulation. *Computer Graphics Forum*, 36(2):75–84, 2017. doi:10.1111/cgf.13108.

Yeara Kozlov, Hongyi Xu, Moritz Bächer, Derek Bradley, Markus Gross, and Thabo Beeler. Data-Driven Physical Face Inversion, 2019. doi:10.48550/arXiv.1907.10402.

Tassilo Kugelstadt, Dan Koschier, and Jan Bender. Fast Corotated FEM Using Operator Splitting. *Computer Graphics Forum*, 37(8), 2018. doi:10.1111/cgf.13520.

Tassilo Kugelstadt, Jan Bender, José Antonio Fernández-Fernández, Stefan Rhys Jeske, Fabian Löschner, and Andreas Longva. Fast Corotated Elastic SPH Solids with Implicit Zero-Energy Mode Control. *Proceedings of the ACM on Computer Graphics and Interactive Techniques*, 4(3):33:1–33:21, 2021. doi:10.1145/3480142.

Lei Lan, Danny M. Kaufman, Minchen Li, Chenfanfu Jiang, and Yin Yang. Affine Body Dynamics: Fast, Stable and Intersection-free Simulation of Stiff Materials. *ACM Transactions on Graphics*, 41(4):1–14, 2022. doi:10.1145/3528223.3530064.

Lei Lan, Minchen Li, Chenfanfu Jiang, Huamin Wang, and Yin Yang. Second-Order Stencil Descent for Interior-Point Hyperelasticity. *ACM Transactions on Graphics*, 42(4):108:1–108:16, 2023. doi:10.1145/3592104.

Jeongseok Lee, Michael X. Grey, Sehoon Ha, Tobias Kunz, Sumit Jain, Yuting Ye, Siddhartha S. Srinivasa, Mike Stilman, and C. Karen Liu. DART: Dynamic Animation and Robotics Toolkit. *Journal of Open Source Software*, 3(22), 2018.

Yuencheng Lee, Demetri Terzopoulos, and Keith Waters. Realistic Modeling for Facial Animation. In *Proceedings of the 22nd Annual Conference on Computer Graphics and Interactive Techniques*, pages 55–62, 1995. doi:10.1145/218380.218407.

J. P. Lewis, Ken Anjyo, Taehyun Rhee, Mengjie Zhang, Fred Pighin, and Zhigang Deng. Practice and Theory of Blendshape Facial Models. *Eurographics State of the Art Reports*, 2014. doi:10.2312/egst.20141042.

Junxuan Li, Chen Cao, Gabriel Schwartz, Rawal Khirodkar, Christian Richardt, Tomas Simon, Yaser Sheikh, and Shunsuke Saito. URAvatar: Universal Relightable Gaussian Codec Avatars. In *SIGGRAPH Asia 2024 Conference Papers*, 2024a. doi:10.1145/3680528.3687653.

Minchen Li, Zachary Ferguson, Teseo Schneider, Timothy Langlois, Denis Zorin, Daniele Panozzo, Chenfanfu Jiang, and Danny M. Kaufman. Incremental Potential Contact: Intersection-and Inversion-Free, Large-Deformation Dynamics. *ACM Transactions on Graphics*, 39(4), 2020. doi:10.1145/3386569.3392425.

Minchen Li, Danny M. Kaufman, and Chenfanfu Jiang. Codimensional Incremental Potential Contact. *ACM Transactions on Graphics*, 40(4):1–24, 2021. doi:10.1145/3450626.3459767.

Xuan Li, Minchen Li, Xuchen Han, Huamin Wang, Yin Yang, and Chenfanfu Jiang. A Dynamic Duo of Finite Elements and Material Points. In *ACM SIGGRAPH 2024 Conference Papers*, 2024b. doi:10.1145/3641519.3657449.

Huancheng Lin, Floyd M. Chitalu, and Taku Komura. Isotropic ARAP Energy Using Cauchy-Green Invariants. *ACM Transactions on Graphics*, 41(6), 2022. doi:10.1145/3550454.3555507.

Alexander D. Lindsay, Derek R. Gaston, Cody J. Permann, Jason M. Miller, David Andrs, Andrew E. Slaughter, Fande Kong, Joshua Hansel, Robert W. Carlsen, Casey Icenhour, Logan Harbour, Guillaume L. Giudicelli, Roy H. Stogner, Peter German, Jacob Badger, Sudipta Biswas, Leora Chapuis, Christopher Green, Jason Hales, Tianchen Hu, Wen Jiang, Yeon Sang Jung, Christopher Matthews, Yinbin Miao, April Novak, John W. Peterson, Zachary M. Prince, Andrea Rovinelli, Sebastian Schunert, Daniel Schwen, Benjamin W. Spencer, Swetha Veeraraghavan, Antonio Recuero, Dewen Yushu, Yaqi Wang, Andy Wilkins, and Christopher Wong. 2.0 - MOOSE: Enabling Massively Parallel Multiphysics Simulation. *SoftwareX*, 20:101202, 2022. doi:10.1016/j.softx.2022.101202.

C. Karen Liu and Dan Negrut. The Role of Physics-Based Simulators in Robotics. *Annual Review of Control, Robotics, and Autonomous Systems*, 4(1):35–58, 2021.

Tiantian Liu, Adam W. Bargteil, James F. O'Brien, and Ladislav Kavan. Fast Simulation of Mass-spring Systems. *ACM Transactions on Graphics*, 32(6):1–7, 2013. doi:10.1145/2508363.2508406.

Stephen Lombardi, Jason Saragih, Tomas Simon, and Yaser Sheikh. Deep Appearance Models for Face Rendering. *ACM Transactions on Graphics*, 37(4), 2018. doi:10.1145/3197517.3201401.

Andreas Longva, Fabian Löschner, Tassilo Kugelstadt, José Antonio Fernández-Fernández, and Jan Bender. Higher-Order Finite Elements for Embedded Simulation. *ACM Transactions on Graphics*, 39(6), 2020. doi:10.1145/3414685.3417853.

Andreas Longva, Fabian Löschner, José Antonio Fernández-Fernández, Egor Larionov, Uri M. Ascher, and Jan Bender. Pitfalls of Projection: A Study of Newton-Type Solvers for Incremental Potentials. *ACM Transactions on Graphics*, 2026.

Fabian Löschner, José Antonio Fernández-Fernández, Stefan Rhys Jeske, Andreas Longva, and Jan Bender. Micropolar Elasticity in Physically-Based Animation. *Proceedings of the ACM on Computer Graphics and Interactive Techniques*, 6(3):1–24, 2023. doi:10.1145/3606922.

Fabian Löschner, José Antonio Fernández-Fernández, Stefan Rhys Jeske, and Jan Bender. Curved Three-Director Cosserat Shells with Strong Coupling. *Computer Graphics Forum*, 43(8), 2024. doi:10.1111/cgf.15183.

Shugao Ma, Tomas Simon, Jason Saragih, Dawei Wang, Yuecheng Li, Fernando de la Torre, and Yaser Sheikh. Pixel Codec Avatars. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 64–73, 2021.

M. Macklin, K. Erleben, M. Müller, N. Chentanez, S. Jeschke, and T. Y. Kim. Primal/Dual Descent Methods for Dynamics. In *Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, 2020. doi:10.1111/cgf.14104.

Miles Macklin, Kier Storey, Michelle Lu, Pierre Terdiman, Nuttapong Chentanez, Stefan Jeschke, and Matthias Müller. Small Steps in Physics Simulation. In *Proceedings of the 18th Annual ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pages 1–7, 2019. doi:10.1145/3309486.3340247.

Ahmed Mahmoud, Rahul Goel, Jonathan Ragan-Kelley, and Justin Solomon. Locality-Aware Automatic Differentiation on the GPU for Mesh-Based Computations. *ACM Transactions on Graphics*, 45(4), 2026. doi:10.1145/3811338.

Viktor Makoviychuk, Lukasz Wawrzyniak, Yunrong Guo, Michelle Lu, Kier Storey, Miles Macklin, David Hoeller, Nikita Rudin, Arthur Allshire, Ankur Handa, et al. Isaac Gym: High Performance GPU-based Physics Simulation for Robot Learning. *arXiv preprint arXiv:2108.10470*, 2021.

Maplesoft. Maple, 2026. URL `https://www.maplesoft.com/`.

Michael Mara, Felix Heide, Michael Zollhöfer, Matthias Nießner, and Pat Hanrahan. Thallo – Scheduling for High-Performance Large-Scale Non-Linear Least-Squares Solvers. *ACM Transactions on Graphics*, 40(5), 2021. doi:10.1145/3453986.

Sebastian Martin, Bernhard Thomaszewski, Eitan Grinspun, and Markus Gross. Example-based Elastic Materials. *ACM Transactions on Graphics*, 30(4), 2011. doi:10.1145/2010324.1964967.

Aleka McAdams, Yongning Zhu, Andrew Selle, Mark Empey, Rasmus Tamstorf, Joseph Teran, and Eftychios Sifakis. Efficient Elasticity for Character Skinning with Contact and Collisions. *ACM Transactions on Graphics*, 30(4), 2011. doi:10.1145/2010324.1964932.

Aaron Meurer, Christopher P. Smith, Mateusz Paprocki, Ondrej Certik, Sergey B. Kirpichev, Matthew Rocklin, AMiT Kumar, Sergiu Ivanov, Jason K. Moore, Sartaj Singh, Thilina Rathnayake, Sean Vig, Brian E. Granger, Richard P. Muller, Francesco Bonazzi, Harsh Gupta, Shivam Vats, Fredrik Johansson, Fabian Pedregosa, Matthew J. Curry, Andy R. Terrel, Stepan Roucka, Ashutosh Saboo, Isuru Fernando, Sumith Kulal, Robert Cimrman, and Anthony Scopatz. SymPy: Symbolic Computing in Python. *PeerJ Computer Science*, 3:e103, 2017. doi:10.7717/peerj-cs.103.

JJ Monaghan. Smoothed Particle Hydrodynamics. *Annual Review of Astronomy and Astrophysics*, 30(1):543–574, 1992. doi:10.1146/annurev.astro.30.1.543.

Guy M. Morton. A Computer Oriented Geodetic Data Base and a New Technique in File Sequencing. Technical report, IBM Ltd., Ottawa, Canada, 1966.

William S. Moses and Valentin Churavy. Instead of Rewriting Foreign Code for Machine Learning, Automatically Synthesize Fast Gradients. In *Proceedings of the 34th International Conference on Neural Information Processing Systems*, 2020.

Matthias Müller and Markus Gross. Interactive Virtual Materials. In *Proceedings of Graphics Interface 2004*, pages 239–246, 2004.

Matthias Müller, Bruno Heidelberger, Marcus Hennix, and John Ratcliff. Position Based Dynamics. *Journal of Visual Communication and Image Representation*, 18(2):109–118, 2007.

Rahul Narain, Matthew Overby, and George E. Brown. ADMM ⊇ Projective Dynamics: Fast Simulation of General Constitutive Models. In *ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pages 1–8, 2016.

Jorge Nocedal and Stephen J. Wright. *Numerical Optimization*. Springer, second edition, 2006. doi:10.1007/978-0-387-40065-5.

J. F. Nolan. Analytical Differentiation on a Digital Computer. Master's thesis, Massachusetts Institute of Technology, 1953.

NVIDIA Corporation. Isaac Sim, 2026a. URL https://developer.nvidia.com/isaac-sim.

NVIDIA Corporation. NVIDIA PhysX, 2026b. URL https://developer.nvidia.com/physx-sdk.

M. Ortiz and L. Stainier. The Variational Formulation of Viscoplastic Constitutive Updates. *Computer Methods in Applied Mechanics and Engineering*, 171(3):419–444, 1999. doi:10.1016/S0045-7825(98)00219-9.

Matthew Overby, George E. Brown, Jie Li, and Rahul Narain. ADMM ⊇ Projective Dynamics: Fast Simulation of Hyperelastic Models with Dynamic Constraints. *IEEE Transactions on Visualization and Computer Graphics*, 23(10):2222–2234, 2017. doi:10.1109/TVCG.2017.2730875.

Hyojoon Park, Sangeetha Grama Srinivasan, Matthew Cong, Doyub Kim, Byungsoo Kim, Jonathan Swartz, Ken Museth, and Eftychios Sifakis. Near-realtime Facial Animation by Deep 3D Simulation Super-Resolution. *ACM Transactions on Graphics*, 43(5):158:1–158:20, 2024. doi:10.1145/3670687.

Frederick I. Parke. Computer Generated Animation of Faces. In *Proceedings of the ACM Annual Conference - Volume 1*, volume 1, pages 451–457, 1972. doi:10.1145/800193.569955.

Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic Differentiation in PyTorch. In *NIPS 2017 Workshop on Autodiff*, 2017.

Andreas Peer, Christoph Gissler, Stefan Band, and Matthias Teschner. An Implicit SPH Formulation for Incompressible Linearly Elastic Solids. *Computer Graphics Forum*, 37(6):135–148, 2017. doi:10.1111/cgf.13317.

Brandon Pelfrey and Donald House. Adaptive Neighbor Pairing for Smoothed Particle Hydrodynamics. In *Advances in Visual Computing*, pages 192–201, 2010. doi:10.1007/978-3-642-17274-8_19.

Stephen M. Platt and Norman I. Badler. Animating Facial Expressions. *SIGGRAPH Comput. Graph.*, 15(3):245–252, 1981. doi:10.1145/965161.806812.

R. Radovitzky and M. Ortiz. Error Estimation and Adaptive Meshing in Strongly Nonlinear Dynamic Problems. *Computer Methods in Applied Mechanics and Engineering*, 172(1): 203–240, 1999. doi:10.1016/S0045-7825(98)00230-8.

Florian Rathgeber, David A Ham, Lawrence Mitchell, Michael Lange, Fabio Luporini, Andrew TT McRae, Gheorghe-Teodor Bercea, Graham R Markall, and Paul HJ Kelly. Firedrake: Automating the Finite Element Method by Composing Abstractions. *ACM Transactions on Mathematical Software (TOMS)*, 43(3):1–27, 2016.

Magí Romanyà-Serrasolsas, Juan J. Casafranca, and Miguel A. Otaduy. Painless Differentiable Rotation Dynamics. *ACM Transactions on Graphics*, 44(4):1–13, 2025. doi:10.1145/3730944.

M. Romeo and S. C. Schvartzman. Data-Driven Facial Simulation. *Computer Graphics Forum*, 39(6):513–526, 2020. doi:10.1111/cgf.14089.

Patrick David Sanan. *Geometric Elasticity for Graphics, Simulation, and Computation*. PhD thesis, California Institute of Technology, 2014. doi:10.7907/DF7X-F354.

P. Schmidt, J. Born, D. Bommes, M. Campen, and L. Kobbelt. TinyAD: Automatic Differentiation in Geometry Processing Made Simple. *Computer Graphics Forum*, 41(5):113–124, 2022. doi:10.1111/cgf.14607.

Craig Schroeder. Practical Course on Computing Derivatives in Code. In *ACM SIGGRAPH 2019 Courses*, 2019. doi:10.1145/3305366.3328073.

Lior Shapira, Ariel Shamir, and Daniel Cohen-Or. Consistent Mesh Partitioning and Skeletonisation Using the Shape Diameter Function. *The Visual Computer*, 24:249–259, 2008.

Alvin Shi and Theodore Kim. A Unified Analysis of Penalty-Based Collision Energies. *Proceedings of the ACM on Computer Graphics and Interactive Techniques*, 6(3):1–19, 2023. doi:10.1145/3606934.

Soshi Shimada, Vladislav Golyanik, Patrick Pérez, and Christian Theobalt. Decaf: Monocular Deformation Capture for Face and Hand Interactions. *ACM Transactions on Graphics*, 42 (6), 2023. doi:10.1145/3618329.

Anna Shtengel, Roi Poranne, Olga Sorkine-Hornung, Shahar Z. Kovalsky, and Yaron Lipman. Geometric Optimization via Composite Majorization. *ACM Transactions on Graphics*, 36 (4):1–11, 2017. doi:10.1145/3072959.3073618.

Eftychios Sifakis, Igor Neverov, and Ronald Fedkiw. Automatic Determination of Facial Muscle Activations from Sparse Motion Capture Marker Data. *ACM Transactions on Graphics*, 24(3):417–425, 2005. doi:10.1145/1073204.1073208.

Breannan Smith, Fernando de Goes, and Theodore Kim. Stable Neo-Hookean Flesh Simulation. *ACM Transactions on Graphics*, 37(2), 2018. doi:10.1145/3180491.

Breannan Smith, Fernando de Goes, and Theodore Kim. Analytic Eigensystems for Isotropic Distortion Energies. *ACM Transactions on Graphics*, 38(1), 2019. doi:10.1145/3241041.

Russell L. Smith. ODE: Open Dynamics Engine, 2026. URL `https://www.ode.org`.

Barbara Solenthaler and Markus Gross. Two-Scale Particle Simulation. *ACM Transactions on Graphics*, 30(4), 2011. doi:10.1145/2010324.1964976.

Olga Sorkine and Marc Alexa. As-rigid-as-possible Surface Modeling. In *Proceedings of the Fifth Eurographics Symposium on Geometry Processing*, pages 109–116, 2007. doi:10.2312/SGP/SGP07/109-116.

Sangeetha Grama Srinivasan, Qisi Wang, Junior Rojas, Gergely Klár, Ladislav Kavan, and Eftychios Sifakis. Learning Active Quasistatic Physics-based Models from Data. *ACM Transactions on Graphics*, 40(4):1–14, 2021. doi:10.1145/3450626.3459883.

Alexey Stomakhin, Russell Howes, Craig Schroeder, and Joseph M. Teran. Energetically Consistent Invertible Elasticity. In *Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pages 25–32, 2012.

Tetsuya Takahashi, Yoshinori Dobashi, Issei Fujishiro, Tomoyuki Nishita, and Ming C. Lin. Implicit Formulation for SPH-Based Viscous Fluids. *Computer Graphics Forum*, 34(2): 493–502, 2015. doi:10.1111/cgf.12578.

Min Tang, Zhongyuan Liu, Ruofeng Tong, and Dinesh Manocha. PSCC: Parallel Self-Collision Culling with Spatial Hashing on GPUs. *Proceedings of the ACM on Computer Graphics and Interactive Techniques*, 1(1):1–18, 2018. doi:10.1145/3203188.

Alessandro Tasora, Radu Serban, Hammad Mazhar, Arman Pazouki, Daniel Melanz, Jonathan Fleischmann, Michael Taylor, Hiroyuki Sugiyama, and Dan Negrut. Chrono: An Open Source Multi-physics Dynamics Engine. In Tomáš Kozubek, editor, *High Performance Computing in Science and Engineering*, 2016.

Joseph Teran, Eftychios Sifakis, Geoffrey Irving, and Ronald Fedkiw. Robust Quasistatic Finite Elements and Flesh Simulation. In *Proceedings of the 2005 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pages 181–190, 2005. doi:10.1145/1073368.1073394.

Emanuel Todorov, Tom Erez, and Yuval Tassa. MuJoCo: A Physics Engine for Model-based Control. In *IEEE/RSJ Int. Conf. on Intelligent Robots and Systems*, 2012.

Loup Verlet. Computer "Experiments" on Classical Fluids. I. Thermodynamical Properties of Lennard-Jones Molecules. *Physical Review*, 159(1):98–103, 1967. doi:10.1103/physrev.159.98.

G. Viccione, V. Bovolin, and E. Pugliese Carratelli. Defining and Optimizing Algorithms for Neighbouring Particle Identification in SPH Fluid Simulations. *International Journal for Numerical Methods in Fluids*, 58(6):625–638, 2008. doi:10.1002/fld.1761.

Pascal Volino and Nadia Magnenat-Thalmann. Simple Linear Bending Stiffness in Particle Systems. In *Proceedings of the 2006 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pages 101–105, 2006.

Nicolas Wagner, Ulrich Schwanecke, and Mario Botsch. Neural Volumetric Blendshapes: Computationally Efficient Physics-Based Facial Blendshapes, 2022. doi:10.48550/arXiv.2212.14784.

Nicolas Wagner, Mario Botsch, and Ulrich Schwanecke. SoftDECA: Computationally Efficient Physics-Based Facial Animations. In *Proceedings of the 16th ACM SIGGRAPH Conference on Motion, Interaction and Games*, pages 1–11, 2023. doi:10.1145/3623264.3624439.

Nicolas Wagner, Mario Botsch, and Ulrich Schwanecke. NePHIM: A Neural Physics-Based Head-Hand Interaction Model. *Computer Graphics Forum*, 44(2):e70045, 2025. doi:10.1111/cgf.70045.

Zhendong Wang, Yin Yang, and Huamin Wang. Stable Discrete Bending by Analytic Eigensystem and Adaptive Orthotropic Geometric Stiffness. *ACM Transactions on Graphics*, 42 (6):1–16, 2023. doi:10.1145/3618372.

Keith Waters. A Muscle Model for Animation Three-dimensional Facial Expression. In *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*, pages 17–24, 1987. doi:10.1145/37401.37405.

Marcel Weiler, Dan Koschier, Magnus Brand, and Jan Bender. A Physically Consistent Implicit Viscosity Solver for SPH Fluids. *Computer Graphics Forum*, 37(2):145–155, 2018.

Lukas Westhofen, José Antonio Fernández-Fernández, Stefan Rhys Jeske, and Jan Bender. Strongly Coupled Simulation of Magnetic Rigid Bodies. *Computer Graphics Forum*, 43 (8), 2024. doi:10.1111/cgf.15185.

James S. Willis, Matthieu Schaller, Pedro Gonnet, Richard G. Bower, and Peter W. Draper. An Efficient SIMD Implementation of Pseudo-Verlet Lists for Neighbour Interactions in Particle-Based Codes. *Advances in Parallel Computing*, 32, 2018. doi:10.3233/978-1-61499-843-3-507.

Rene Winchenbach and Andreas Kolb. Multi-Level Memory Structures for Simulating and Rendering Smoothed Particle Hydrodynamics. *Computer Graphics Forum*, 39(6):527–541, 2020.

Rene Winchenbach and Andreas Kolb. Optimized Refinement for Spatially Adaptive SPH. *ACM Transactions on Graphics*, 40(1):1–15, 2021. doi:10.1145/3363555.

Rene Winchenbach, Hendrik Hochstetter, and Andreas Kolb. Constrained Neighbor Lists for SPH-Based Fluid Simulations. In *Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pages 49–56, 2016. doi:10.2312/SCA.20161222.

Rene Winchenbach, Hendrik Hochstetter, and Andreas Kolb. Infinite Continuous Adaptivity for Incompressible SPH. *ACM Transactions on Graphics*, 36(4), 2017.

Inc. Wolfram Research. Mathematica, Version 13.2, 2026. URL `https://www.wolfram.com/mathematica`.

Peter Wriggers. *Nonlinear Finite Element Methods*. Springer Berlin Heidelberg, 2008. doi:10.1007/978-3-540-71001-1.

Haomiao Wu and Theodore Kim. An Eigenanalysis of Angle-Based Deformation Energies. *Proceedings of the ACM on Computer Graphics and Interactive Techniques*, 6(3):1–19, 2023. doi:10.1145/3606929.

Qingxuan Wu, Zhiyang Dou, Sirui Xu, Soshi Shimada, Chen Wang, Zhengming Yu, Yuan Liu, Cheng Lin, Zeyu Cao, Taku Komura, Vladislav Golyanik, Christian Theobalt, Wenping Wang, and Lingjie Liu. DICE: End-to-end Deformation Capture of Hand-Face Interactions from a Single Image, 2024. doi:10.48550/arXiv.2406.17988.

Xilin Xia and Qiuhua Liang. A GPU-Accelerated Smoothed Particle Hydrodynamics (SPH) Model for the Shallow Water Equations. *Environmental Modelling & Software*, 75:28–43, 2016. doi:10.1016/j.envsoft.2015.10.002.

Tianyi Xie, Minchen Li, Yin Yang, and Chenfanfu Jiang. A Contact Proxy Splitting Method for Lagrangian Solid-Fluid Coupling. *ACM Transactions on Graphics*, 42(4):122:1–122:14, 2023. doi:10.1145/3592115.

Lingchen Yang, Byungsoo Kim, Gaspard Zoss, Baran Gözcü, Markus Gross, and Barbara Solenthaler. Implicit Neural Representation for Physics-driven Actuated Soft Bodies. *ACM Transactions on Graphics*, 41(4):1–10, 2022. doi:10.1145/3528223.3530156.

Lingchen Yang, Gaspard Zoss, Prashanth Chandran, Paulo Gotardo, Markus Gross, Barbara Solenthaler, Eftychios Sifakis, and Derek Bradley. An Implicit Physical Face Model Driven by Expression and Style. In *SIGGRAPH Asia 2023 Conference Papers*, pages 1–12, 2023. doi:10.1145/3610548.3618156.

Lingchen Yang, Gaspard Zoss, Prashanth Chandran, Markus Gross, Barbara Solenthaler, Eftychios Sifakis, and Derek Bradley. Learning a Generalized Physical Face Model from Data. *ACM Transactions on Graphics*, 43(4):94:1–94:14, 2024. doi:10.1145/3658189.

Chang Yu, Yi Xu, Ye Kuang, Yuanming Hu, and Tiantian Liu. MeshTaichi: A Compiler for Efficient Mesh-Based Operations. *ACM Transactions on Graphics*, 41(6), 2022. doi:10.1145/3550454.3555430.

Yujin Robot. Kobuki Description Model, 2013. URL `https://github.com/yujinrobot/kobuki_description`.

ChangAn Zhu and Chris Joslin. Fabrig: A Cloth-Simulated Transferable 3D Face Parameterization. In *SIGGRAPH Asia 2024 Conference Papers*, 2024. doi:10.1145/3680528.3687695.