SymX: Energy-based Simulation from Symbolic Expressions

JOSÉ ANTONIO FERNÁNDEZ-FERNÁNDEZ, RWTH Aachen University, Germany
FABIAN LÖSCHNER, RWTH Aachen University, Germany
LUKAS WESTHOFEN, RWTH Aachen University, Germany
ANDREAS LONGVA, RWTH Aachen University, Germany
JAN BENDER, RWTH Aachen University, Germany

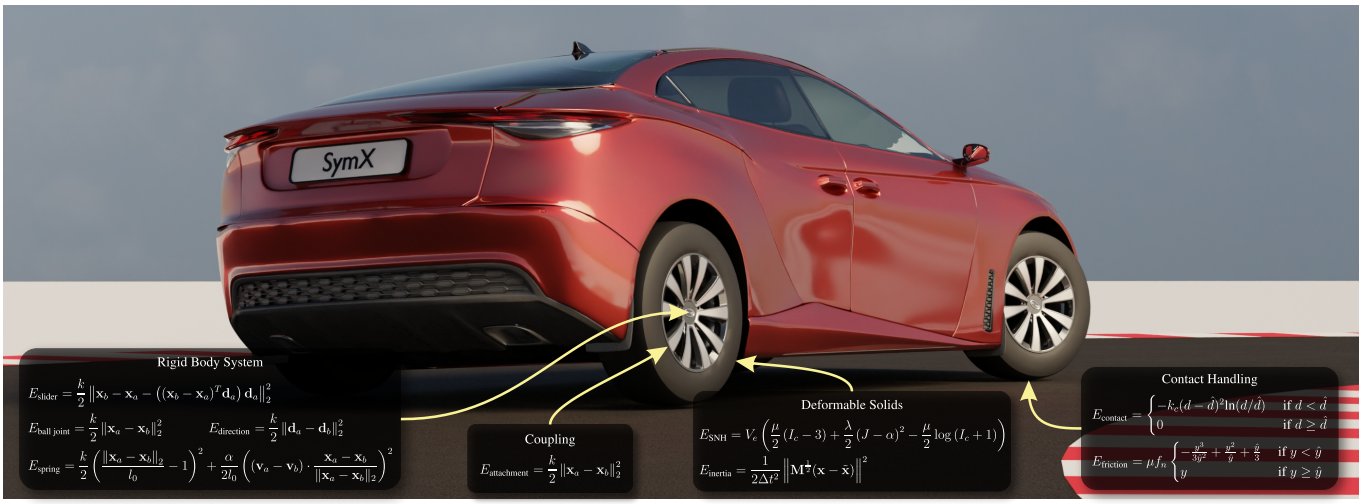


Fig. 1. Simulation based on an optimization time integrator of a car drifting through a tight hairpin corner with strong coupling between rigid bodies and deformable solids. The simulation model consists of nine non-linear potential energies: FEM with linear tetrahedra and the Stable Neo-Hookean material model [Smith et al. 2018] for the tires, constraint-based energies [Macklin et al. 2020] for the rigid body components (sliders, ball joints, direction joints, and damped springs), attachment constraints for the coupling of the rigid body system for the suspension with the tires and a frictional contact potential based on the Incremental Potential Contact method [Li et al. 2020]. All energies are succinctly defined using SymX, which can automatically compute the global gradient and Hessian used to solve the optimization time integration.

Optimization time integrators are effective at solving complex multi-physics problems including deformable solids with non-linear material models, contact with friction, strain limiting, etc. For challenging problems, Newton-type optimizers are often used, which necessitates first- and second-order derivatives of the global non-linear objective function. Manually differentiating, implementing, testing, optimizing, and maintaining the resulting code is extremely time-consuming, error-prone, and precludes quick changes to the model, even when using tools that assist with parts of such pipeline.

We present SymX[1], an open source framework that computes the required derivatives of the different energy contributions by symbolic differentiation, generates optimized code, compiles it on-the-fly, and performs the global assembly. The user only has to provide the symbolic expression of each energy for a single representative element in its corresponding discretization and our system will determine the assembled derivatives for the whole simulation. We demonstrate the versatility of SymX in complex simulations featuring different non-linear materials, high-order finite elements, rigid body systems, adaptive discretizations, frictional contact, and coupling of multiple interacting physical systems.

SymX's derivatives offer performance on par with SymPy, an established off-the-shelf symbolic engine, and produces simulations at least one order of magnitude faster than TinyAD, an alternative state-of-the-art integral solution.

CCS Concepts: • **Computing methodologies** → **Physical simulation**.

Additional Key Words and Phrases: physically-based simulation, symbolic differentiation, optimization time integration

ACM Reference Format:
José Antonio Fernández-Fernández, Fabian Löschner, Lukas Westhofen, Andreas Longva, and Jan Bender. 2025. SymX: Energy-based Simulation from Symbolic Expressions. *ACM Trans. Graph.* 44, 4 (September 2025), 20 pages. https://doi.org/10.1145/3764928

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

Authors' Contact Information: José Antonio Fernández-Fernández, RWTH Aachen University, Aachen, Germany, fernandez@cs.rwth-aachen.de; Fabian Löschner, RWTH Aachen University, Aachen, Germany, loeschner@cs.rwth-aachen.de; Lukas Westhofen, RWTH Aachen University, Aachen, Germany, l.westhofen@cs.rwth-aachen.de; Andreas Longva, RWTH Aachen University, Aachen, Germany, longva@cs.rwth-aachen.de; Jan Bender, RWTH Aachen University, Aachen, Germany, bender@cs.rwth-aachen.de.

© 2025 ACM.
This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in *ACM Transactions on Graphics*, https://doi.org/10.1145/3764928.

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 [Andrews et al. 2022], 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 in recent years [Brown et al. 2018; Chen et al. 2022; Gast et al. 2015; Narain et al. 2016]. 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 energy and all energy potentials, e.g., using Newton's method. While first-order methods, such as Projective Dynamics [Bouaziz et al. 2014], may be used to solve this type of problem, in this work we focus exclusively on second-order methods due to their strong convergence and robustness guarantees [Li et al. 2020].

In this context, first- and second-order derivatives of many, and possibly very complex, energy expressions are required for the minimization process. 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.

There already exist tools which try to solve some of these problems, e.g., by computing the required derivatives using automatic differentiation or frameworks and languages that assist with the assembly process. However, there is no tool capable of automating the whole pipeline in an effective manner. In this work, 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 paper we show in a detailed analysis that existing tools 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 Fig. 1. SymX has already been used in research on complex materials and interactions [Löschner et al. 2023, 2024; Westhofen et al. 2024], in differentiable simulation [Fernández-Fernández et al. 2025], and as the core of STARK [Fernández-Fernández et al. 2024], a simulator for strong coupling between deformable and rigid bodies for use in robotics. Section 6 of this document includes example applications using SymX for non-linear materials, high-order finite elements, rigid body systems, adaptive discretizations, frictional contact, and coupling of multiple interacting physical systems.

2 Related Work

In this section we first cover simulation methods that require first- and second-order derivatives to guarantee robustness. 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 4 for an in-depth discussion about the feasibility of applying specific methods and tools listed herein to our application.

2.1 Optimization Time Integrators

Using an incremental potential formulation [Ortiz and Stainier 1999] for dynamic problems is a common approach in computational mechanics. Derived or related methods also have become popular in computer animation where they are often referred to as optimization time integrators.

Formulating the dynamic systems as a scalar optimization problem instead of a non-linear system of equations was shown to be favorable for robustness and efficiency of the implementation [Gast et al. 2015; Kharevych et al. 2006]. While this robustness is usually associated with Newton-style methods that use a full Hessian, local approaches such as Projective Dynamics [Bouaziz et al. 2014; Narain et al. 2016] can be used in case of stricter performance constraints. To fulfill high accuracy requirements, Li et al. [2019] proposed a different method using domain decomposition that improves efficiency especially in case of extreme non-linear and high-speed deformations. Recently, optimization-based contact models gained considerable popularity. The Incremental Potential Contact (IPC) approach [Li et al. 2020] and its extension to Codimensional IPC [Li et al. 2021] excel at providing robust interpenetration-free frictional contact handling. The characteristic robustness and convergence of such methods is subject to having access to second-order derivative information of the underlying global objective function. While contact potentials with barriers in general appear to be a promising choice for many applications, introducing them to orthogonal

phenomenological research projects or existing multi-physics systems [Holz et al. 2025] can require significant development effort. As we show later, our framework allows users to easily integrate models inspired by IPC in already complex simulation settings.

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 it is out of the scope of our work to give an extensive review of the field, we point the interested reader to the book by Griewank et al. [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 2023] or Maple [Maplesoft 2023]) 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. Concurrently with our work, Herholz et al. [2024] expands on the method by incorporating assembly instead of generating code for the entire problem.

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 fulfil all the requirements established in Section 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.

3 Problem Definition

Many physical models used in simulation satisfy the following ordinary differential equation

$$\mathbf{M}\dot{\mathbf{v}} = \mathbf{f}(\mathbf{x}) = -\nabla E(\mathbf{x}), \qquad \dot{\mathbf{x}} = \mathbf{v}. \tag{1}$$

Here $\mathbf{x}$ is a vector containing some variant of positional degrees of freedom of the discrete system, $\mathbf{v}$ similarly contains the velocity degrees of freedom, $\mathbf{M}$ is the *mass matrix*, which might be constant or depend on $\mathbf{x}$, $\mathbf{f}$ is a discrete representation of the forces acting on the system and E is a scalar potential function. This ODE does not readily hold for rigid bodies without the introduction of a kinematic map [Bender et al. 2014], but in the interest of a simpler presentation we leave this aspect out of the present discussion. In general, dissipative forces, friction for instance, might not have an associated scalar potential E in the formulation above. In such cases, it is often possible to work around this restriction by *lagging* the dissipative forces in question in some fashion [Li et al. 2020].

To compute one time step of size Δt for this problem, and without loss of generality, we may for example use the reformulation of Backward Euler as an optimization problem (cf. [Gast et al. 2015; Kugelstadt et al. 2018; Narain et al. 2016]) to obtain the *incremental potential*

$$E_{\mathrm{BE}}(\mathbf{x}) := \frac{1}{2\Delta t^2}\left\|\mathbf{M}^{\frac{1}{2}}(\mathbf{x}-\tilde{\mathbf{x}})\right\|^2 + E(\mathbf{x}) = E_{\mathrm{inertia}}(\mathbf{x}) + E(\mathbf{x}), \tag{2}$$

where $\tilde{\mathbf{x}} = \mathbf{x}(t) + \Delta t\mathbf{v}(t) + (\Delta t)^2\mathbf{M}^{-1}\mathbf{f}_{\mathrm{ext}}$ and $\mathbf{f}_{\mathrm{ext}}$ is the vector of external forces which are constant during a time step. The associated update rules are

$$\begin{aligned}\mathbf{x}(t+\Delta t) &= \min_{\mathbf{x}} E_{\mathrm{BE}}(\mathbf{x})\\ \mathbf{v}(t+\Delta t) &= \frac{1}{\Delta t}\left(\mathbf{x}(t+\Delta t) - \mathbf{x}(t)\right).\end{aligned} \tag{3}$$

Note that many other integration methods permit a similar reformulation as an optimization problem, such as the midpoint rule [Dinev et al. 2018], the trapezoidal rule, BDF2 and TR-BDF2 [Brown et al. 2018; Chen et al. 2022]. We can describe the associated minimization problem as a sum of energy functions

$$\min_{\mathbf{u}} \sum_i E_i(\mathbf{u}; \mathcal{P}_i) \tag{4}$$

in which we have used the state vector $\mathbf{u}$ to describe the degrees of freedom, typically positions or velocities. The abstract quantity $\mathcal{P}_i$ represents the parameters of the energy function E_i, i.e., the data of the problem that is not dependent on the state $\mathbf{u}$.

Usually, energies can be decomposed into a number of smaller contributions. For example, the total strain energy $E_{\mathrm{strain}} = \sum_e E_{\mathrm{strain},e}$ for a deformable finite element model is the sum of the individual element strain energies $E_{\mathrm{strain},e}$. To capture this inherent structure of the problem, we introduce abstract *elements* to the formulation. In practice, an element is an entity that has a contribution to the global potential energy, e.g., a tetrahedral finite element to simulate a deformable solid, a rigid body or a contact point between two objects. Each energy E_i then gets associated with a set of elements $\mathcal{E}_i$ where it is defined and evaluated. We now replace Eq. (4) with our general problem formulation

$$\min_{\mathbf{u}} E(\mathbf{u}) = \sum_i \sum_{e\in\mathcal{E}_i} E_i(\mathbf{R}_e\mathbf{u}; \mathcal{P}_{i,e}), \tag{5}$$

where $\mathbf{u}$ denotes the *global* degrees of freedom and $\mathbf{R}_e$ is the selection operator that extracts the degrees of freedom specific to the element e. $\mathcal{P}_{i,e}$ are the parameters specific to element e for energy i. In other words, $\mathbf{R}_e$ maps global to *element-local* quantities, and in consequence an energy E_i operates only on element-local inputs of the same size. Its definition is independent of a specific element instance; only the parameters change. This concept can also be applied to models that require (possibly non-linear) mappings, e.g. between coordinate systems for rigid bodies, by simply folding such mappings into the definition of E_i and defining different sets of elements $\mathcal{E}_i$ with an associated energy for different compositions of mappings.

To summarize, each energy function $E_i(\hat{\mathbf{u}}; \mathcal{P}_i)$ is therefore a function of a generic vector $\hat{\mathbf{u}}$ with *fixed* input size, evaluated for each associated element in $\mathcal{E}_i$. It is then possible to symbolically represent, differentiate and generate code for each E_i, and finally assemble the overall derivative of E by summation.

Under the assumption that $E(\mathbf{u})$ is at least C^1 continuous, we can efficiently solve (5) with an appropriate choice of optimizer (see [Nocedal and Wright 2006]).

3.1 Example: Deformable solids

We now demonstrate how to formulate a motivating example within the mathematical framework of (5). We wish to simulate a deformable solid with the non-linear Neo-Hookean material using a linear tetrahedral finite element discretization and the Backward Euler integrator, subject to gravity. We let $\mathbf{u} = \mathbf{x}$ be the global vector of deformed vertex positions, and each element is associated with four vertices, forming a local vector $\hat{\mathbf{x}} = \mathbf{R}_e\mathbf{u} = \mathbf{R}_e\mathbf{x} \in \mathbb{R}^{12}$ containing the deformed vertex positions stacked in an element-local vector. From this we can compute the deformation gradient $\mathbf{F}_e = \mathbf{F}_e(\hat{\mathbf{x}})$ of the element [Sifakis and Barbic 2012].

The strain energy density for the Neo-Hookean model is given by

$$\psi^{\mathrm{NH}} = \frac{\mu}{2}(I_c - 3) + \mu\log(\det(\mathbf{F})) + \frac{\lambda}{2}\log^2(\det(\mathbf{F})), \tag{6}$$

where μ and λ are the Lamé parameters and $I_c = \mathrm{tr}(\mathbf{F}^T\mathbf{F})$ [Smith et al. 2018]. We can compute the strain energy for the element by integrating the strain energy density over its domain K_e

$$E_{\mathrm{NH},e}(\hat{\mathbf{x}}) = \int_{K_e} \psi^{NH}(\mathbf{F})\mathrm{d}\mathbf{X} = V_e\,\psi^{NH}(\mathbf{F}_e(\hat{\mathbf{x}})). \tag{7}$$

Here V_e denotes the volume of the element. Our total energy function for the minimization problem (5) becomes

$$E(\mathbf{x}) = E_{\mathrm{inertia}} + \sum_e E_{\mathrm{NH},e}(\mathbf{R}_e\mathbf{x}). \tag{8}$$

Since the gradient and Hessian are computed and assembled by our framework, only the energy functions $E_{\mathrm{inertia},e}$ and $E_{\mathrm{NH},e}$ need to be provided in symbolic form by the user.

4 Existing solutions

Our framework, is designed to facilitate the exploration of novel, complicated potentials and intricate interactions across multiple systems and discretizations. While SymX can of course implement relatively simple simulations and well-known potentials found in the literature, its principal advantage lies in supporting work beyond that.

With that objective in mind, we now examine existing approaches to differentiation, evaluation, and assembly in the context of our problem as defined in Eq. (5), and compare their suitability against the requirements of automation, performance, productivity, flexibility, and accessibility set forth in Section 1.

4.1 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.

4.2 Numerical differentiation

While numerical differentiation has seen impressive advances in robustness and can provide reliable derivative information, for example with the Complex Step method [Luo et al. 2019], it still faces the fundamental problem that it requires multiple evaluations of the energy value itself, at least one for each entry in the gradient and Hessian. In our testing on a linear tetrahedral element with a 12×12 Hessian, evaluating the value of the energy was significantly more expensive than 1/144th of the runtime needed for the whole Hessian matrix. Therefore, we consider numerical differentiation unsuitable for our application.

4.3 Automatic differentiation

AD is often the solution of choice for many applications due to its flexibility, easiness of integration in existing codebases 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 (Eq. (5)) which can 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 to differentiate with respect to all global (instead of local) degrees of freedom, which becomes unfeasible at large scales. Additionally, the limitations of general-purpose AD tools are further exacerbated when the topology of the problem changes, for example due to dynamic contacts or remeshing, necessitating the recalculation of global derivatives and/or problem sparsity. The lack of structural awareness of the problem also inhibits the possibility to perform per-element operations, such as projecting element Hessians to the cone of positive semi-definite matrices, which is a common practice in second-order minimization frameworks [Li et al. 2020; Teran et al. 2005].



Fig. 2. The drum of a tumble dryer rotates with eight pieces of cloth inside. 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.

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 7.

4.4 Symbolic off-the-shelf tools

Symbolic mathematical engines such as Mathematica [Wolfram Research 2023], Maple [Maplesoft 2023] or SymPy [Meurer et al. 2017] can be used to compute derivatives and generate corresponding code. 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.

To highlight why this is desirable, consider the tumble dryer simulation shown in Fig. 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.

Relying solely on external differentiation tools would require the user to generate the code for all the involved per-element energies, followed by manual integration into the simulation code. To incorporate the externally generated code, the user then has to write glue code for gathering the locally required values for each energy from

the global data arrays. In the dryer example for instance, this becomes very tedious since each one of the 138 functions has a unique signature and operates on a distinct set of inputs and outputs which requires a function-specific mapping from the simulation data and assembly to the global derivative data structures. Even after this initial setup, changes to the expressions might happen regularly in research projects which would require re-running the external tools and potentially updating the function handling in the simulation codebase. This process is error-prone and time-consuming and therefore does not meet the goals established in Section 1 for automation and productivity.

Some existing symbolic engines, such as Mathematica, provide low-level C interfaces to access their symbolic functionalities, which could be used to avoid relying on external scripts and to integrate the energy definitions directly in the simulation codebase. However, introducing external differentiation tools in the pipeline still requires implementing the declaration, evaluation and assembly components. 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 7 for SymPy differentiation benchmarks. Finally, coupling commercial engines (e.g., Maple or Mathematica) directly into the simulation codebase invalidates our goal of accessibility as defined in Section 1 since closed source licensed software prevents researchers from exchanging ideas or reproducing other works freely. In contrast, simulation-native open source solutions such as TinyAD or SymX offer a much more lightweight, fully-automated, single-codebase pipeline and have almost no setup and distribution barriers as only a C++ compiler is required.

4.5 Simulation systems and DSLs

While there is a plethora of relevant systems and Domain Specific Languages (DSL) as outlined in Section 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 dynamic branching and is outperformed by integrated solutions such as TinyAD [Schmidt et al. 2022]. The method by Herholz et al., on the other hand, presents in fact very 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. Further, their method has very long code generation and compilation times as they show in Table 3 of their work [Herholz et al. 2022].

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 great 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 for the evaluation and assembly of the Neo-Hookean potential energy in Section 7.3.

4.6 Conclusion

We observe that prevailing trends in frameworks for computer graphics [Schmidt et al. 2022] and related fields such as rendering [Jakob et al. 2022; Mara et al. 2021], machine learning [Abadi et al. 2016; Paszke et al. 2017] or mathematics [Alnæs et al. 2015; Rathgeber et al. 2016] show that modern tools have proliferated precisely because they are accessible, offer a very high degree of automation and safety and provide appealing flexibility and performance. The ongoing scientific research into better, problem-specific solutions for complex differentiation applications demonstrates that differentiation is in practice still an open problem, and that existing general purpose tools, while useful, do not offer a definitive solution for all differentiation needs. In this context, we identify a space for a framework to support researchers in developing and sharing simulation models in the context of Newton-type solvers, which is the motivation behind SymX.

5 SymX Framework

SymX is an integrated solution that seeks to fulfil all the requirements defined in Section 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 5.1. The symbolic engine is introduced in Section 5.2. Finally, we discuss the required matrix assembly in Section 5.3.

5.1 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. Fig. 3 shows how the minimization problem of the deformable solids example in Section 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. See Fig. 4 for a self-contained SymX example of the setup required to declare the energies defined in Fig. 3.

5.2 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`

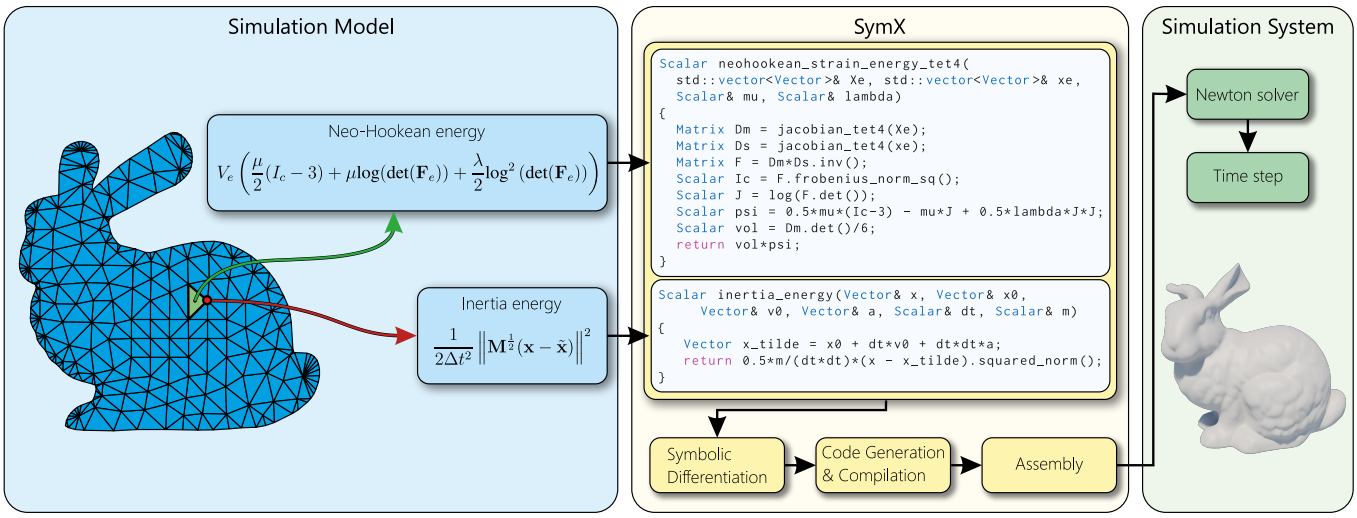


Fig. 3. Overview of a simulation step with the SymX framework. Left: The input is a discretized model and energy functions. In our application example we use a tet mesh and the inertia and Neo-Hookean strain energy functions. Center: The user has to implement a symbolic definition of these functions. 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.

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 is a special type of operation that is discussed in Section 5.2.6.

In the following we describe the components of our symbolic engine using the example presented in Section 3.1.

5.2.1 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 of our symbolic differentiation. 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 Fig. 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.

5.2.2 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 tetrahedra element, differentiation times improve from 1.88 ms to 0.44 ms thanks to caching intermediate derivatives.

5.2.3 Code Generation and Compilation. Once we have the symbolic expressions for a function and its derivatives, we need to

```
1  // Simulation data (uninitialized for brevity)
2  std::vector<std::array<double, 3>> x, x0, v0, a, x_rest;
3  std::vector<double> lumped_mass;
4  double time_step, mu, lambda;
5  std::vector<std::array<int, 4>> tets;
6  std::vector<int> nodes;
7
8  // Create global energy and define contributions
9  GlobalEnergy G;
10 DoF dof = G.add_dof_array(x);
11 G.add_energy("neo_hookean_tet4", tets,
12    [&](Energy& E, Element& tet)
13    {
14       // Create local symbols from the data arrays
15       std::vector<Vector> xe = E.make_dof_vectors(dof, x, tet);
16       std::vector<Vector> Xe = E.make_vectors(x_rest, tet);
17       Scalar m = E.make_scalar(mu);
18       Scalar l = E.make_scalar(lambda);
19
20       // Define energy
21       E.set(neohookean_strain_energy_tet4(Xe, xe, m, l));
22    });
23 G.add_energy("inertia", nodes,
24    [&](Energy& E, Element& node)
25    {
26       // Create local symbols from the data arrays
27       Vector xn = E.make_dof_vector(dof, x, node);
28       Vector x0n = E.make_vector(x0, node);
29       Vector v0n = E.make_vector(v0, node);
30       Vector an = E.make_vector(a, node);
31       Scalar mn = E.make_scalar(lumped_mass, node);
32       Scalar dt = E.make_scalar(time_step);
33
34       // Define energy
35       E.set(inertia_energy(xn, x0n, v0n, an, dt, mn));
36    });
37
38 // Compilation
39 G.compile("path/to/codegen/directory");
40
41 // Assemble global data structures
42 Assembled assembled = G.evaluate_E_grad_hess();
```

Fig. 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 Fig. 3.

evaluate them for all the elements. However, evaluating the expressions by 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). Fig. 5 partially shows the generated function to evaluate the energy value, gradient and Hessian of the Neo-Hookean energy defined in Fig. 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 some metadata with a signature hash id, and the number of inputs and outputs.

```
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;
}
```

Fig. 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 Fig. 4 lines 11 to 22. Only some representative lines from the top and the bottom of the function are shown.

5.2.4 Data-Symbol Mapping. A central concept in SymX is the mapping between simulation data and its corresponding symbols, as this key design principle allows it to evaluate the generated functions and to assemble the global structures autonomously. 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. This mapping works for both simulation data (e.g., positions, velocities, …) and connectivity information (e.g., tetrahedra, triangles, edges, …). Before the evaluation of a specific energy for all related elements, SymX requests updated views of all data arrays associated to it (including its connectivity array) using the lambdas. 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.

We now analyze the code example shown in Fig. 4. First the array of degrees of freedom must be declared (line 10) in order to identify the symbols the system will take derivatives with respect to. Lines 11 and 23 define the energy 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. From lighter to heavier in regards to runtime: 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, used in line 42, 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.

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 rest of this document it is assumed that the high-level data-symbol mapping is used and that evaluations and assembly are automated.

5.2.5 Dynamic topology. Contact interactions 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 smaller 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. This approach works because SymX has updated view access to the data arrays, including their sizes, which are defined and maintained by the user, as explained in the previous section.

Hence, the user only needs to update the list of contact pairs or mesh elements (if needed) before calling SymX. The resulting global derivatives will then reflect the latest state of the simulation, including any changes in the sparsity pattern or even the total number of degrees of freedom. Further details on dynamic topology assembly can be found in Section 5.3.

It is worth noting that SymX itself does not explicitly define concepts such as collision, contact, or even mesh. Everything is instead constructed from symbolic expressions associated with element lists, regardless of whether they represent contact pairs, FEM elements, rigid bodies, or something else entirely.

5.2.6 Extensions. In the following we introduce features of the system which are required for more complex simulations or to improve its performance.

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. In this work we make use of it, for example, in the implementation of the mollifiers needed for the IPC edge-edge contacts and the friction potentials. Additionally, thanks to branching we can implement expressions with functions like `min`, `max`, `abs` and `sign`, which allowed us to write the signed distance function to a cylinder, used in the scene shown in Fig. 9.

Conditional evaluations. All the branches spawned by `branch` will generate results that will be assembled. However, a special case of branching that needs dedicated treatment 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 user can define the energy function in combination with an activation condition:

$$E = \sum_{e \in \mathcal{E}} E_e, \quad E_e = \begin{cases} E_e^+ & \text{if } c_e > 0 \\ 0 & \text{if } c_e \leq 0, \end{cases} \tag{9}$$

where E_e^+ is the energy 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 we have to evaluate an energy density function multiplied by integration weights at a set of fixed integration points. In general, we can formulate this particular case abstractly as

$$E = \sum_{e \in \mathcal{E}} \sum_{k} \tilde{E}(\mathbf{R}_e \mathbf{u}; \mathcal{P}_e, \mathcal{P}_k) \tag{10}$$

for some energy contribution $\tilde{E}$, where $\mathcal{P}_k$ 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 expressions, 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 6.2.

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 7.5. 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. Our system offers optional numerical projection of element Hessian matrices to positive semi-definiteness before assembly, a common practice in second-order minimization methods to assist with convergence in 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 [Barbic 2012; Chao et al. 2010; McAdams et al. 2011] or closed-form derivatives that need intricate procedures to be obtained and cannot be obtained by direct scalar-based differentiation [Lin et al. 2022].

5.3 Assembly

In this section we describe the assembly for the general case of having multiple sets of degrees of freedom $\mathbf{u}_0, \dots \mathbf{u}_n$, 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

$$\begin{pmatrix} \frac{\partial^2 E}{\partial \mathbf{u}_1^2} & \frac{\partial^2 E}{\partial \mathbf{u}_1 \partial \mathbf{u}_2} & \cdots & \frac{\partial^2 E}{\partial \mathbf{u}_1 \partial \mathbf{u}_n} \\ \frac{\partial^2 E}{\partial \mathbf{u}_2 \partial \mathbf{u}_1} & \frac{\partial^2 E}{\partial \mathbf{u}_2^2} & \cdots & \frac{\partial^2 E}{\partial \mathbf{u}_2 \partial \mathbf{u}_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 E}{\partial \mathbf{u}_n \partial \mathbf{u}_1} & \frac{\partial^2 E}{\partial \mathbf{u}_n \partial \mathbf{u}_2} & \cdots & \frac{\partial^2 E}{\partial \mathbf{u}_n^2} \end{pmatrix} \cdot \begin{pmatrix} \Delta \mathbf{u}_1 \\ \Delta \mathbf{u}_2 \\ \vdots \\ \Delta \mathbf{u}_n \end{pmatrix} = - \begin{pmatrix} \frac{\partial E}{\partial \mathbf{u}_1} \\ \frac{\partial E}{\partial \mathbf{u}_2} \\ \vdots \\ \frac{\partial E}{\partial \mathbf{u}_n} \end{pmatrix}, \tag{11}$$

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 1 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 execution of the assembly 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 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.

Algorithm 1: Parallel global Hessian $\mathbf{H}$ evaluation and assembly. To avoid data races during parallel execution, the function `AddInPlace` and `AppendToBucketList` implement mutexes.

```
H ← 0;                          // Previous BCRS matrix zeroed
ClearAndResizeBucketList(B);
foreach energy E do
    foreach element e in parallel do
        d_e ← GatherData(user data, symbol-data maps);
        H_e ← hess_E(d_e);      // Call compiled function
        if project then
            ProjectToPD(H_e);
        foreach block-row i in H_e do
            I ← GlobalIndex(symbol-data maps, i);
            foreach block-column j in H_e do
                J ← GlobalIndex(symbol-data maps, j);
                if BlockExists(H, I, J) then
                    AddInPlace(H, I, J, H_e, i, j);
                else
                    AppendToBucketList(B, I, J, H_e, i, j);
    if HasSparsityChanged(H, B) then
        H ← Rebuild(H, B);
```

6 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 following application examples were implemented with Backward Euler time integration without loss of generality. As discussed in Section 3, other time integration methods can be formulated as an optimization problem, and could be implemented with SymX as well. 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.

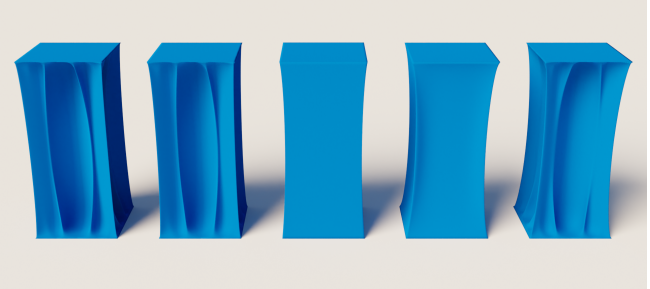


Fig. 6. Comparison of different material models in a simulation of a stretched deformable cube. From left to right: ARAP with a volume conserving term, fixed co-rotational, St. Venant-Kirchhoff, Neo-Hookean and Stable Neo-Hookean material. Note that we use *lagged* (constant per time step), rotation matrices for the ARAP and fixed co-rotational energies.

6.1 Non-Linear Material Models

To demonstrate how concise yet powerful SymX's symbolic representation is, we implemented five different material models which took just 46 lines of code in total, see Appendix B. These constitutive 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 expression for a given element type and let the framework work out the rest. Fig. 6 presents a comparison of the implemented 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 [Kugelstadt et al. 2018; Müller and Gross 2004], 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]).

6.2 High-order Lagrangian Finite Elements

To evaluate the energy of high-order Lagrangian finite elements, numerical integration is typically applied using quadrature rules. The total deformation energy of an element is then given by

$$E_e^{FEM} = \sum_{i=1}^{p} w_i \, \det(\mathbf{J}_0^e) \, \psi\left(\mathbf{F}(\boldsymbol{\xi}_i, \mathbf{X}^e, \mathbf{x}^e)\right), \tag{12}$$

where ψ is the strain energy density function, p is the number of integration points, and w_i represents the quadrature weight. The integration point $\boldsymbol{\xi}_i$ is defined in the coordinate system of the reference element, and $\mathbf{J}_0^e = \mathbf{J}_0^e(\boldsymbol{\xi}_i)$ is the Jacobian of the mapping from the reference element to the physical element in the undeformed configuration (see, e.g., Wriggers et al. [2008]). The following code shows the implementation of a generic FEM integrator (Eq. (12)) in SymX:

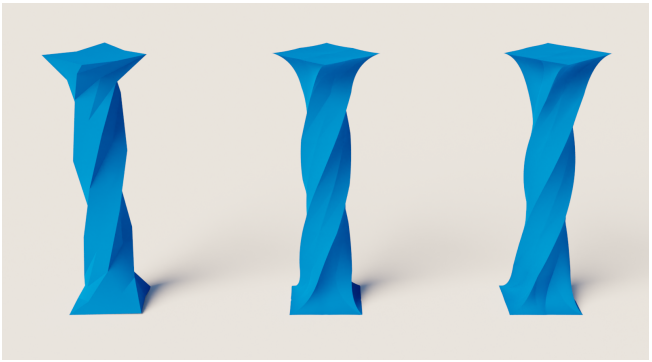


Fig. 7. Comparison of linear (left), quadratic (center) and cubic (right) finite elements in a simulation of a stretched and twisted cube.

```
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;
}
```

where `jac` and `psi` are generic element Jacobian and potential energy density functions. SymX includes common element Jacobians and strain energy density functions by default. Appendix A shows how to use SymX to compute Jacobians, including the example of three common FEM elements: linear and quadratic tetrahedra and bilinear hexahedron.

Since the quadrature points and weights are typically constant per element type, we can employ the fixed summation feature of SymX (see Section 5.2.6). As a result, only the evaluation of ψ at a generic integration point needs to be differentiated and compiled. This example showcases how our framework allows for complex concepts to be expressed very concisely while preserving generality, which significantly boost productivity, reduces the room for error and ease communication between researchers. Additionally, such high level implementations with SymX do not degrade simulation performance since the code that describes the expressions is only executed once to generate the optimized code that is actually evaluated at runtime. Fig. 7 shows a comparison of linear, quadratic, and cubic finite elements for the Stable Neo-Hookean material [Smith et al. 2018].

6.3 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



Fig. 8. Our pipeline seamlessly handles changes in discretization, number of degrees of freedom and sparsity pattern in a cloth simulation with adaptive mesh refinement.

strategy to demonstrate that SymX can handle changes in discretization topology (see Fig. 8).

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]

$$E_b(\mathbf{x}_e) = \frac{k_b}{2}\mathbf{x}_e^T\mathbf{Q}_e\mathbf{x}_e, \tag{13}$$

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 $\mathbf{Q}_e \in \mathbb{R}^{12\times12}$ is the internal edge quadratic form, which is constant during the simulation. Implementing this energy 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 potential energy:

$$E_{sl}(\mathbf{E}) = \sum_i^2 \begin{cases} k_{sl}A_e(\sigma_i(\mathbf{E}) - \sigma_l)^3 & \text{if} \quad \sigma_i(\mathbf{E}) > \sigma_l \\ 0 & \text{if} \quad \sigma_i(\mathbf{E}) \le \sigma_l, \end{cases} \tag{14}$$

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.

6.4 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. Recently, Li et al. [2020] introduced the Incremental Potential Contact (IPC) method which is a robust approach to handle contact with friction. In this section we show how the contact barrier and the friction potentials can be implemented in our framework.

First, we define a contact potential energy as

$$E_c(d) = -k_c(d - \hat{d})^2\ln(d/\hat{d}) \tag{15}$$

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 derive the following potential energy from the IPC friction model

$$E_f(y) = \mu f_n \begin{cases} -\frac{y^3}{3\hat{y}^2} + \frac{y^2}{\hat{y}} + \frac{\hat{y}}{3} & \text{if } y \le \hat{y} \\ y & \text{if } y > \hat{y}, \end{cases} \tag{16}$$

where $y = \|\mathbf{T}\Delta\mathbf{v}\|_2$ is the sliding contact velocity with the contact projection matrix $\mathbf{T} \in \mathbb{R}^{2\times3}$ and the relative velocity $\Delta\mathbf{v} = \mathbf{v}_a - \mathbf{v}_b$ between the contact points a and b. $\hat{y}$ is the slide/stick velocity threshold, f_n is the contact pressure and μ is the Coulomb's friction coefficient. This energy is implemented in SymX as

```
Scalar friction(Vector& va, Vector& vb, Matrix& T,
   Scalar& mu, Scalar& fn, Scalar& yh)
{
   Vector yt = T*(va - vb);
   Scalar y = yt.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 must use a stable norm function that forces the returned value to be zero when $y < \varepsilon$, which is 10^{-14}m/s in our

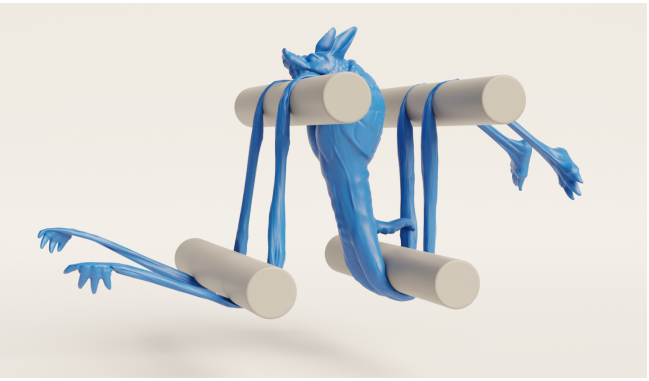


Fig. 9. Robust contact handling in a simulation of an armadillo which is extremely deformed by animated cylinders.

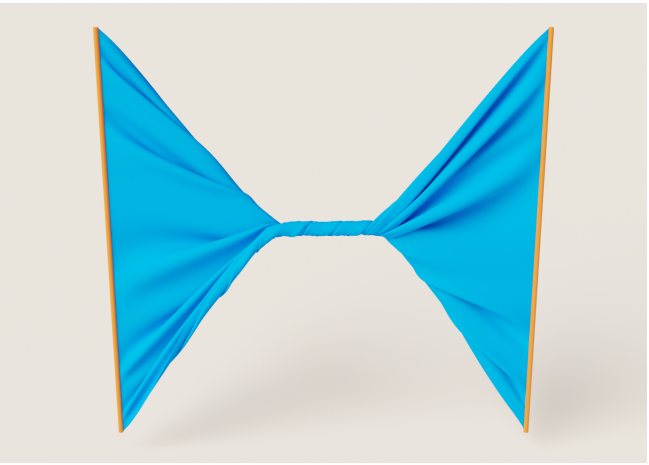


Fig. 10. A cloth is twisted which leads to a configuration with thousands self-collisions.

experiments, to avoid evaluating the function at a singularity which would trigger a division by zero in the derivatives. 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, however. Further discussion about this limitation can be found in Section 8.

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 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. As previously discussed in Section 5.2.5, SymX will evaluate and assemble every element of every energy contained in their respective connectivity array at the time of the evaluation call. Therefore, we only need to run the collision detection and update the list of pairs before requesting the global derivatives to SymX to get the correct assembly.

We use two experiments with extreme contact configurations (see Fig. 9, 10) 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].

6.5 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.

Drifting car. We simulate a car model (see Fig. 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 energies using the penalty method

$$E_C = \frac{1}{2} k_C C^2, \tag{17}$$

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

$$C^2_{bj}(\mathbf{x}_a, \mathbf{x}_b) = \|\mathbf{x}_a - \mathbf{x}_b\|_2^2 \tag{18}$$

$$C^2_{dl}(\mathbf{d}_a, \mathbf{d}_b) = \|\mathbf{d}_a - \mathbf{d}_b\|_2^2 \tag{19}$$

$$C^2_{sj}(\mathbf{x}_a, \mathbf{x}_b, \mathbf{d}_a) = \left\| \mathbf{x}_b - \mathbf{x}_a - \left((\mathbf{x}_b - \mathbf{x}_a)^T \mathbf{d}_a \right) \mathbf{d}_a \right\|_2^2. \tag{20}$$

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

$$\begin{aligned} E_{ds}(\mathbf{x}_a, \mathbf{x}_b, \mathbf{v}_a, \mathbf{v}_b) = &\frac{k_{sp}}{2} \left(\frac{\|\mathbf{x}_a - \mathbf{x}_b\|_2}{l_0} - 1 \right)^2 + \\ &\frac{\alpha_{dp}}{2 l_0} \left((\mathbf{v}_a - \mathbf{v}_b) \cdot \frac{\mathbf{x}_a - \mathbf{x}_b}{\|\mathbf{x}_a - \mathbf{x}_b\|_2} \right)^2, \end{aligned} \tag{21}$$

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 6.4. 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 paper 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.

7 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. In any case, 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.

7.1 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 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.

Table 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.

	linear	quadratic	cubic
Method	$t_{\text{eval.}}$ [µs]	$t_{\text{eval.}}$ [µs]	$t_{\text{eval.}}$ [µs]
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)

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, 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 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.

7.2 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 Fig. 7 with linear, quadratic and cubic tetrahedral elements sharing the same 137K degrees of freedom, and present the timings in Table 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, see Section 5.2.6.

The manually optimized solution is again the fastest in all cases and TinyAD is again the slowest, with SymX being up to 361x 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 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 (not included in Table 2), 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 out of the scope of this work 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.

Table 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.

	linear	quadratic	cubic
Method	t_{total} [ms]	t_{total} [ms]	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)

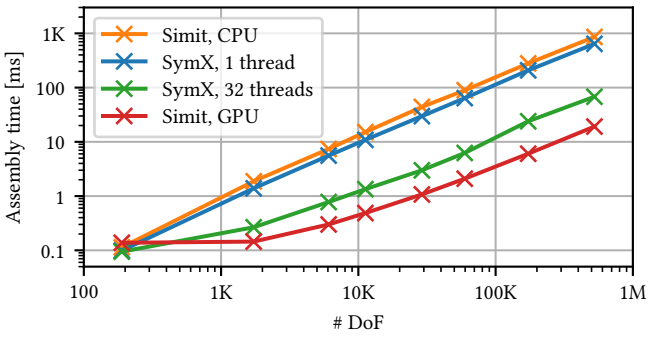


Fig. 11. Total evaluation and assembly time of the Hessian for the dynamic simulation of a cube with Neo-Hookean material and linear tetrahedra. Simit does not parallelize assembly on the CPU.

7.3 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 Fig. 11.

Note that while SymX supports parallel evaluation and assembly (Section 5.3), 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 times in comparison to Simit if we exclude simulations with fewer than 6075 degrees of freedom. Fig. 11 also includes runtimes

Table 3. Number of degrees of freedom and average evaluation and assembly timings per Newton iteration of the application examples from Sec. 6.

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

Table 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.

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

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 times faster than SymX's multithreaded execution, if we exclude 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 they would need to obtain (e.g. for the dryer scene of Fig. 2): collision detection, changing topologies, minimizer and 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.

7.4 Application Example Timings

In addition to these benchmarks, we also show timings of the application examples from Sec. 6 in Table 3.

7.5 Compilation Times

Finally, we also present the timings and memory requirements associated with differentiation and compilation for all the experiments in Table 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. In practice, a simulation software would keep the binaries for all the implemented potentials and only the relevant ones for a given simulation instance would be loaded. 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].

8 Limitations and Future Work

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 optimized 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.

The treatment of hard constraints, e.g. in the form of Lagrange Multipliers, falls beyond the scope of this work. Nevertheless, we anticipate no significant challenges in expanding SymX to handle these hard constraints and their derivatives as long as they can be symbolically represented.

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. 2018, 2019] to perform a more efficient projection. We believe that the techniques we present in

this paper 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 in their expression graph 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 kind 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 6.4. 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.

Finally, a natural next step for SymX to improve its performance is to provide support for GPU execution. Since SymX already implements read-only views of the data required for computing global derivatives, synchronizing data between the host and device would be straightforward. Extending the code generation process to produce GPU-compatible code for derivative evaluation can be accomplished analogously to the existing CPU code generation. Assembly and linear algebra operations could be performed using standard GPU libraries such as cuSPARSE [NVIDIA Corporation 2025].

9 Conclusions

In this work we presented SymX, a system to automate the differentiation and assembly in complex simulations based on optimization time integrators. The proposed system provides a set of symbolic types that allows engineers and researchers to succinctly define the different contributions to the global energy of the simulation. Thanks to the link between these symbols and the simulation data, the system can apply symbolic differentiation to the energies with respect to the degrees of freedom of the simulation and completely automate the evaluation and assembly process. Thanks to on-the-fly compilation of the derivatives code, our method has a performance comparable to code optimized by hand.

We demonstrated the capabilities of our method in an array of challenging simulations featuring state-of-the-art models and showed that the code required to express such simulation energies very closely resembles their original mathematical counterparts. In the view of the results obtained, we conclude that SymX can significantly support engineers by allowing them to quickly prototype fast and reliable simulation software with a minimal code footprint, that is also easy to understand and distribute. Changes in the expressions are immediately incorporated in the simulations which allow researchers to experiment with new models, or variations of existing ones, and to quickly reach results. The flexibility of our method also presents a path for an initial prototype to be gradually transitioned to a hybrid between symbolic and manual derivatives as the user sees fit. SymX is therefore a great candidate to provide flexible and powerful symbolic facilities to higher level simulation codebases that focus on other aspects, such as different types of material discretization or time integration. Finally, we are convinced that also other simulation methods like constraint-based approaches, or even applications in different fields like geometry processing, will benefit from our framework.

Acknowledgments

Fabian Löschner and Andreas Longva are funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) — 281466253; 411281008.

References

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. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Savannah, GA, USA) *(OSDI'16)*. USENIX Association, USA, 265–283.

Martin Alnæs, Jan Blechta, Johan Hake, August Johansson, Benjamin Kehlet, Anders Logg, Chris Richardson, Johannes Ring, Marie E Rognes, and Garth N Wells. 2015. 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. 2019. CasADi – A software framework for nonlinear optimization and optimal control. *Mathematical Programming Computation* 11, 1 (2019), 1–36. https://doi.org/10.1007/s12532-018-0139-4

Sheldon Andrews, Kenny Erleben, and Zachary Ferguson. 2022. Contact and Friction Simulation for Computer Graphics. In *ACM SIGGRAPH 2022 Courses* (Vancouver, British Columbia, Canada) *(SIGGRAPH '22)*. Association for Computing Machinery, New York, NY, USA, Article 3, 172 pages. https://doi.org/10.1145/3532720.3535640

Jernej Barbic. 2012. Exact corotational linear fem stiffness matrix. *USC, Los Angeles, CA, USA* (2012).

Jan Bender, Kenny Erleben, and Jeff Trinkle. 2014. Interactive Simulation of Rigid Body Dynamics in Computer Graphics. *Computer Graphics Forum* 33, 1 (2014), 246–270. https://doi.org/10.1111/cgf.12272

Miklós Bergou, Max Wardetzky, David Harmon, Denis Zorin, and Eitan Grinspun. 2006. A Quadratic Bending Model for Inextensible Surfaces. In *Proceedings of the Eurographics Symposium on Geometry Processing (SGP '06)*. Eurographics Association, 227–230. https://doi.org/10.2312/SGP/SGP06/227-230

Gilbert Louis Bernstein, Chinmayee Shah, Crystal Lemire, Zachary Devito, Matthew Fisher, Philip Levis, and Pat Hanrahan. 2016. Ebb: A DSL for Physical Simulation on CPUs and GPUs. *ACM Trans. Graph.* 35, 2, Article 21 (may 2016), 12 pages. https://doi.org/10.1145/2892632

J. Blinn. 1996. Consider the lowly 2 x 2 matrix. *IEEE Computer Graphics and Applications* 16, 2 (1996), 82–88. https://doi.org/10.1109/38.486688

Sofien Bouaziz, Sebastian Martin, Tiantian Liu, Ladislav Kavan, and Mark Pauly. 2014. Projective Dynamics: Fusing Constraint Projections for Fast Simulation. *ACM Trans. Graph.* 33, 4, Article 154 (jul 2014), 11 pages. https://doi.org/10.1145/2601097.2601116

George E. Brown, Matthew Overby, Zahra Forootaninia, and Rahul Narain. 2018. Accurate dissipative forces in optimization integrators. *ACM Transactions on Graphics* 37, 6 (dec 2018), 1–14. https://doi.org/10.1145/3272127.3275011

Isaac Chao, Ulrich Pinkall, Patrick Sanan, and Peter Schröder. 2010. A Simple Geometric Model for Elastic Deformations. *ACM Trans. Graph.* 29, 4, Article 38 (jul 2010), 6 pages. https://doi.org/10.1145/1778765.1778775

Yunuo Chen, Minchen Li, Lei Lan, Hao Su, Yin Yang, and Chenfanfu Jiang. 2022. A Unified Newton Barrier Method for Multibody Dynamics. *ACM Trans. Graph.* 41, 4, Article 66 (jul 2022), 14 pages. https://doi.org/10.1145/3528223.3530076

Deshana Desai, Etai Shuchatowitz, Zhongshi Jiang, Teseo Schneider, and Daniele Panozzo. 2022. ACORNS: An easy-to-use code generator for gradients and Hessians. *SoftwareX* 17 (2022), 100901. https://doi.org/10.1016/j.softx.2021.100901

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. 2011. 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* (Seattle, Washington) *(SC '11)*. Association for Computing Machinery, New York, NY, USA, Article 9, 12 pages. https://doi.org/10.1145/2063384.2063396

Zachary DeVito, Michael Mara, Michael Zollhöfer, Gilbert Bernstein, Jonathan Ragan-Kelley, Christian Theobalt, Pat Hanrahan, Matthew Fisher, and Matthias Niessner. 2017. Opt: A Domain Specific Language for Non-Linear Least Squares Optimization in Graphics and Imaging. *ACM Trans. Graph.* 36, 5, Article 171 (oct 2017), 27 pages. https://doi.org/10.1145/3132188

Dimitar Dinev, Tiantian Liu, and Ladislav Kavan. 2018. Stabilizing Integrators for Real-Time Physics. *ACM Transactions on Graphics* 37, 1 (Jan. 2018), 1–19. https://doi.org/10.1145/3153420

Zachary Ferguson, Minchen Li, Teseo Schneider, Francisca Gil-Ureta, Timothy Langlois, Chenfanfu Jiang, Denis Zorin, Danny M. Kaufman, and Daniele Panozzo. 2021. Intersection-free rigid body dynamics. *ACM Transactions on Graphics* 40, 4 (July 2021), 1–16. https://doi.org/10.1145/3450626.3459802

José Antonio Fernández-Fernández, Ryan Goldade, Ladislav Kavan, Jan Bender, and Philipp Herholz. 2025. 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, Article 58 (Aug. 2025). https://doi.org/10.1145/3747860

José Antonio Fernández-Fernández, Ralph Lange, Stefan Laible, Kai O. Arras, and Jan Bender. 2024. 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)*. 16888–16894. https://doi.org/10.1109/ICRA57147.2024.10610574

Theodore Gast, Craig Schroeder, Alexey Stomakhin, Chenfanfu Jiang, and Joseph Teran. 2015. Optimization Integrator for Large Time Steps. *IEEE Transactions on Visualization and Computer Graphics* 21, 10 (2015), 1103–1115. https://doi.org/10.1109/TVCG.2015.2459687

Andreas Griewank and Andrea Walther. 2008. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation* (second ed.). Society for Industrial and Applied Mathematics, USA.

Laurent Hascoet and Valérie Pascual. 2013. The Tapenade automatic differentiation tool: Principles, model, and specification. *ACM Trans. Math. Software* 39, 3 (April 2013), 1–43. https://doi.org/10.1145/2450153.2450158

F. Hecht. 2012. New development in FreeFem++. *J. Numer. Math.* 20, 3-4 (2012), 251–265. https://freefem.org/

Philipp Herholz, Tuur Stuyck, and Ladislav Kavan. 2024. A Mesh-based Simulation Framework using Automatic Code Generation. *ACM Transactions on Graphics* 43, 6 (Nov. 2024), 1–17. https://doi.org/10.1145/3687986

Philipp Herholz, Xuan Tang, Teseo Schneider, Shoaib Kamil, Daniele Panozzo, and Olga Sorkine-Hornung. 2022. Sparsity-Specific Code Optimization Using Expression Trees. *ACM Trans. Graph.* 41, 5, Article 175 (may 2022), 19 pages. https://doi.org/10.1145/3520484

Daniel Holz, Stefan Rhys Jeske, Fabian Löschner, Jan Bender, Yin Yang, and Sheldon Andrews. 2025. Multiphysics Simulation Methods in Computer Graphics. *Computer Graphics Forum* (April 2025). https://doi.org/10.1111/cgf.70082

Yuanming Hu, Tzu-Mao Li, Luke Anderson, Jonathan Ragan-Kelley, and Frédo Durand. 2019. Taichi: A Language for High-Performance Computation on Spatially Sparse Data Structures. *ACM Trans. Graph.* 38, 6, Article 201 (nov 2019), 16 pages. https://doi.org/10.1145/3355089.3356506

Wenzel Jakob, Sébastien Speierer, Nicolas Roussel, and Delio Vicini. 2022. Dr.Jit: A Just-in-Time Compiler for Differentiable Rendering. *ACM Trans. Graph.* 41, 4, Article 124 (jul 2022), 19 pages. https://doi.org/10.1145/3528223.3530099

Kai Jia. 2021. SANM: A Symbolic Asymptotic Numerical Solver with Applications in Mesh Deformation. *ACM Trans. Graph.* 40, 4, Article 79 (jul 2021), 16 pages. https://doi.org/10.1145/3450626.3459755

L. Kharevych, Weiwei Yang, Y. Tong, E. Kanso, J. E. Marsden, P. Schröder, and M. Desbrun. 2006. Geometric, Variational Integrators for Computer Animation. In *Proceedings of the 2006 ACM SIGGRAPH/Eurographics Symposium on Computer Animation* (Vienna, Austria) *(SCA '06)*. Eurographics Association, Goslar, DEU, 43–51. https://doi.org/10.2312/SCA/SCA06/043-051

Theodore Kim, Fernando De Goes, and Hayley Iben. 2019. Anisotropic Elasticity for Inversion-Safety and Element Rehabilitation. *ACM Trans. Graph.* 38, 4, Article 69 (jul 2019), 15 pages. https://doi.org/10.1145/3306346.3323014

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. 2016. Simit: A Language for Physical Simulation. *ACM Trans. Graph.* 35, 2, Article 20 (mar 2016), 21 pages. https://doi.org/10.1145/2866569

Tassilo Kugelstadt, Dan Koschier, and Jan Bender. 2018. Fast Corotated FEM using Operator Splitting. *Computer Graphics Forum* 37, 8 (2018), 12 pages. https://doi.org/10.1111/cgf.13520

Lei Lan, Danny M. Kaufman, Minchen Li, Chenfanfu Jiang, and Yin Yang. 2022. Affine body dynamics: fast, stable and intersection-free simulation of stiff materials. *ACM Transactions on Graphics* 41, 4 (July 2022), 1–14. https://doi.org/10.1145/3528223.3530064

Minchen Li, Zachary Ferguson, Teseo Schneider, Timothy Langlois, Denis Zorin, Daniele Panozzo, Chenfanfu Jiang, and Danny M. Kaufman. 2020. Incremental Potential Contact: Intersection-and Inversion-Free, Large-Deformation Dynamics. *ACM Trans. Graph.* 39, 4, Article 49 (aug 2020), 20 pages. https://doi.org/10.1145/3386569.3392425

Minchen Li, Ming Gao, Timothy Langlois, Chenfanfu Jiang, and Danny M. Kaufman. 2019. Decomposed optimization time integrator for large-step elastodynamics. *ACM Transactions on Graphics* 38, 4 (jul 2019), 1–10. https://doi.org/10.1145/3306346.3322951

Minchen Li, Danny M. Kaufman, and Chenfanfu Jiang. 2021. Codimensional Incremental Potential Contact. *ACM Trans. Graph.* 40, 4, Article 170 (jul 2021), 24 pages. https://doi.org/10.1145/3450626.3459767

Huancheng Lin, Floyd M. Chitalu, and Taku Komura. 2022. Isotropic ARAP Energy Using Cauchy-Green Invariants. *ACM Trans. Graph.* 41, 6, Article 275 (nov 2022), 14 pages. https://doi.org/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. 2022. 2.0 - MOOSE: Enabling massively parallel multiphysics simulation. *SoftwareX* 20 (2022), 101202. https://doi.org/10.1016/j.softx.2022.101202

Ran Luo, Weiwei Xu, Tianjia Shao, Hongyi Xu, and Yin Yang. 2019. Accelerated Complex-Step Finite Difference for Expedient Deformable Simulation. *ACM Trans. Graph.* 38, 6, Article 160 (nov 2019), 16 pages. https://doi.org/10.1145/3355089.3356493

Fabian Löschner, José Antonio Fernández-Fernández, Stefan Rhys Jeske, Andreas Longva, and Jan Bender. 2023. Micropolar Elasticity in Physically-Based Animation. *Proceedings of the ACM on Computer Graphics and Interactive Techniques* 6, 3 (Aug. 2023), 1–24. https://doi.org/10.1145/3606922

F. Löschner, J. A. Fernández-Fernández, S. R. Jeske, and J. Bender. 2024. Curved Three-Director Cosserat Shells with Strong Coupling. *Computer Graphics Forum* 43, 8 (Oct. 2024). https://doi.org/10.1111/cgf.15183

M. Macklin, K. Erleben, M. Müller, N. Chentanez, S. Jeschke, and T. Y. Kim. 2020. Primal/Dual Descent Methods for Dynamics. In *Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Computer Animation* (Virtual Event, Canada) *(SCA '20)*. Eurographics Association, Goslar, DEU, Article 9, 12 pages. https://doi.org/10.1111/cgf.14104

Maplesoft. 2023. *Maple*. Waterloo, Ontario. https://www.maplesoft.com/

Michael Mara, Felix Heide, Michael Zollhöfer, Matthias Nießner, and Pat Hanrahan. 2021. Thallo – Scheduling for High-Performance Large-Scale Non-Linear Least-Squares Solvers. *ACM Trans. Graph.* 40, 5, Article 184 (sep 2021), 14 pages. https://doi.org/10.1145/3453986

Marcin Maździarz. 2010. Unified Isoparametric 3D LagrangeFinite Elements. *CMES. Computer Modeling in Engineering & Sciences* 66 (09 2010). https://doi.org/10.3970/cmes.2010.066.001

Aleka McAdams, Yongning Zhu, Andrew Selle, Mark Empey, Rasmus Tamstorf, Joseph Teran, and Eftychios Sifakis. 2011. Efficient Elasticity for Character Skinning with Contact and Collisions. *ACM Trans. Graph.* 30, 4, Article 37 (jul 2011), 12 pages. https://doi.org/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, Fredrik Vats, Shivam andJohansson, Fabian Pedregosa, Matthew J. Curry, Andy R. Terrel, Stepan Roucka, Ashutosh Saboo, Isuru Fernando, Sumith Kulal, Robert Cimrman, and Anthony Scopatz. 2017. SymPy: symbolic computing in Python. *PeerJ Computer Science* 3 (Jan. 2017), e103. https://doi.org/10.7717/peerj-cs.103

William S. Moses and Valentin Churavy. 2020. Instead of rewriting foreign code for machine learning, automatically synthesize fast gradients. In *Proceedings of the 34th International Conference on Neural Information Processing Systems* (Vancouver,

BC, Canada) *(NIPS '20)*. Curran Associates Inc., Red Hook, NY, USA, Article 1046, 14 pages.
Matthias Müller and Markus Gross. 2004. Interactive Virtual Materials. In *Proceedings of Graphics Interface 2004* (London, Ontario, Canada) *(GI '04)*. Canadian Human-Computer Communications Society, Waterloo, CAN, 239–246.
Rahul Narain, Matthew Overby, and George E. Brown. 2016. ADMM ⊇ Projective Dynamics: Fast Simulation of General Constitutive Models. In *ACM SIGGRAPH/Eurographics Symposium on Computer Animation*. 1–8.
Jorge Nocedal and Stephen J. Wright. 2006. *Numerical Optimization* (2e ed.). Springer, New York, NY, USA.
J. F. Nolan. 1953. *Analytical Differentiation on a Digital Computer*. Master's thesis. Massachusetts Institute of Technology.
NVIDIA Corporation. 2025. cuSPARSE Library, CUDA Toolkit 13.0. https://docs.nvidia.com/cuda/cusparse/
M. Ortiz and L. Stainier. 1999. The variational formulation of viscoplastic constitutive updates. *Computer Methods in Applied Mechanics and Engineering* 171, 3 (1999), 419–444. https://doi.org/10.1016/S0045-7825(98)00219-9
Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic differentiation in PyTorch. (2017).
Florian Rathgeber, David A Ham, Lawrence Mitchell, Michael Lange, Fabio Luporini, Andrew TT McRae, Gheorghe-Teodor Bercea, Graham R Markall, and Paul HJ Kelly. 2016. Firedrake: automating the finite element method by composing abstractions. *ACM Transactions on Mathematical Software (TOMS)* 43, 3 (2016), 1–27.
Patrick David Sanan. 2014. *Geometric Elasticity for Graphics, Simulation, and Computation*. Ph. D. Dissertation. California Institute of Technology. https://doi.org/10.7907/DF7X-F354
P. Schmidt, J. Born, D. Bommes, M. Campen, and L. Kobbelt. 2022. TinyAD: Automatic Differentiation in Geometry Processing Made Simple. *Computer Graphics Forum* 41, 5 (2022), 113–124. https://doi.org/10.1111/cgf.14607
Craig Schroeder. 2019. Practical Course on Computing Derivatives in Code. In *ACM SIGGRAPH 2019 Courses* (Los Angeles, California) *(SIGGRAPH '19)*. Association for Computing Machinery, New York, NY, USA, Article 22, 22 pages. https://doi.org/10.1145/3305366.3328073
Eftychios Sifakis and Jernej Barbic. 2012. FEM Simulation of 3D Deformable Solids. In *ACM SIGGRAPH Courses*. 1–50. https://doi.org/10.1145/2343483.2343501
Breannan Smith, Fernando De Goes, and Theodore Kim. 2018. Stable Neo-Hookean Flesh Simulation. *ACM Transactions on Graphics* 37, 2, Article 12 (mar 2018), 15 pages. https://doi.org/10.1145/3180491
Breannan Smith, Fernando De Goes, and Theodore Kim. 2019. Analytic Eigensystems for Isotropic Distortion Energies. *ACM Trans. Graph.* 38, 1, Article 3 (feb 2019), 15 pages. https://doi.org/10.1145/3241041
Olga Sorkine and Marc Alexa. 2007. As-Rigid-As-Possible Surface Modeling. In *Eurographics Symposium on Geometry Processing*, Alexander Belyaev and Michael Garland (Eds.). The Eurographics Association. https://doi.org//10.2312/SGP/SGP07/109-116
Alexey Stomakhin, Russell Howes, Craig Schroeder, and Joseph M. Teran. 2012. Energetically Consistent Invertible Elasticity. In *Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Computer Animation* (Lausanne, Switzerland) *(SCA '12)*. Eurographics Association, Goslar, DEU, 25–32.
Joseph Teran, Eftychios Sifakis, Geoffrey Irving, and Ronald Fedkiw. 2005. Robust Quasistatic Finite Elements and Flesh Simulation. In *Proceedings of the 2005 ACM SIGGRAPH/Eurographics Symposium on Computer Animation* (Los Angeles, California) *(SCA '05)*. Association for Computing Machinery, New York, NY, USA, 181–190. https://doi.org/10.1145/1073368.1073394
L. Westhofen, J. A. Fernández-Fernández, S. R. Jeske, and J. Bender. 2024. Strongly Coupled Simulation of Magnetic Rigid Bodies. *Computer Graphics Forum* 43, 8 (Oct. 2024). https://doi.org/10.1111/cgf.15185
Inc. Wolfram Research. 2023. Mathematica, Version 13.2. https://www.wolfram.com/mathematica Champaign, IL, 2022.
Peter Wriggers. 2008. *Nonlinear finite element methods*. Springer Science & Business Media.
Chang Yu, Yi Xu, Ye Kuang, Yuanming Hu, and Tiantian Liu. 2022. MeshTaichi: A Compiler for Efficient Mesh-Based Operations. *ACM Trans. Graph.* 41, 6, Article 252 (nov 2022), 17 pages. https://doi.org/10.1145/3550454.3555430

A Jacobians

We illustrate now how to define Jacobian functions with SymX for Lagrangian FEM simulations. The Jacobian of an element is

$$\mathbf{J}_\mathbf{x} = \frac{\partial \mathbf{x}}{\partial \boldsymbol{\xi}} \tag{22}$$

where $\mathbf{x}$ are the coordinates in the current configuration and $\boldsymbol{\xi}$ the coordinates in the reference configuration. The Jacobian at rest configuration $\mathbf{J}_\mathbf{X}$ is defined analogously. Space is typically discretized as $\mathbf{x} \approx \mathbf{N}(\mathbf{x}i)\mathbf{x}_h$ where $\mathbf{N}$ are the interpolation (or shape) functions and $\mathbf{x}_h$ are the coordinates of the nodes of the discretization element.

The following is a generic function to compute Jacobians symbolically with SymX

```
Matrix fem_jacobian(const FEM_Element_Type& element_type,
    const std::vector<Vector>& xh, const Vector& xi)
{
    int N = xi.size();
    Vector v = interpolation(element_type, xh, xi);
    std::vector<Scalar> jac;
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            jac.push_back(diff(v[i], xi[j]));
        }
    }
    return Matrix(jac, { N, N });
}
```

For completeness, we include below the interpolation scheme of three common FEM element types: linear and quadratic tetrahedra and bilinear hexahedron [Maździarz 2010].

```
enum class FEM_Element_Type { Tet4, Tet10, Hex8 };

template<typename T>
T interpolation(const FEM_Element_Type& element_type,
    const std::vector<T>& v, const Vector& xi)
{
    if (element_type == FEM_Element_Type::Tet4) {
        Vector N = Vector({
            1.0 - xi[0] - xi[1] - xi[2],
            xi[0],
            xi[1],
            xi[2]});
        return dot(N, v);
    }
    else if (element_type == FEM_Element_Type::Tet10) {
        Scalar N0 = 1.0 - xi[0] - xi[1] - xi[2];
        Scalar N1 = xi[0];
        Scalar N2 = xi[1];
        Scalar N3 = xi[2];
        Vector N = Vector({
            N0 * (2.0 * N0 - 1.0),
            N1 * (2.0 * N1 - 1.0),
            N2 * (2.0 * N2 - 1.0),
            N3 * (2.0 * N3 - 1.0),
            4.0 * N0 * N1,
            4.0 * N1 * N2,
            4.0 * N2 * N0,
            4.0 * N0 * N3,
            4.0 * N1 * N3,
            4.0 * N2 * N3});
        return dot(N, v);
    }
    else if (element_type == FEM_Element_Type::Hex8) {
        Scalar NXm = 0.5 * (1.0 - xi[0]);
        Scalar NXp = 0.5 * (1.0 + xi[0]);
        Scalar NYm = 0.5 * (1.0 - xi[1]);
        Scalar NYp = 0.5 * (1.0 + xi[1]);
        Scalar NZm = 0.5 * (1.0 - xi[2]);
        Scalar NZp = 0.5 * (1.0 + xi[2]);
        Vector N = Vector({
            NXm * NYm * NZm,
            NXp * NYm * NZm,
            NXp * NYp * NZm,
            NXm * NYp * NZm,
            NXm * NYm * NZp,
            NXp * NYm * NZp,
            NXp * NYp * NZp,
            NXm * NYp * NZp });
        return dot(N, v);
    }
}
```

In Section 6.2 we show a generic FEM integrator written with SymX that can take any Jacobian function `jac` with the following signature

```
std::function<Matrix(std::vector<Vector>&,Vector&)> jac;
```

In order to use the example jacobian function provided in this appendix one needs to select an element before passing it to the integrator. For example

```
auto jac = [element_type](std::vector<T>& v, Vector& xi){
     fem_jacobian(element_type, v, xi);
  };
```

Similar selection would need be done for the constitutive models in Appendix B. Note that this flexible composability using C++ lambdas does not have performance implications at simulation time since the symbols are processed and compiled regardless of how the expressions are constructed.

B Constitutive Models

Here we show the implementation in SymX of the five constitutive models [Lin et al. 2022; Smith et al. 2018; Stomakhin et al. 2012] used in the simulation shown in Fig. 6.

```
Scalar constitutive_models_energy_density(Energy& energy,
   Matrix& F, Matrix& R,  /* R is assumed constant*/
   Scalar& lambda, Scalar& mu,
   ConstitutiveModels model)
{
   // Eq. 14 from [Smith et al. 2018]
   if (model == ConstitutiveModels::StableNeoHookean) {
      Scalar mu_ = 4/3*mu;
      Scalar lambda_ = lambda + 5/6*lambda;

      Matrix C = F.transpose()*F;
      Scalar detF = F.det();
      Scalar Ic = C.trace();
      Scalar alpha = 1 + mu_/lambda_ - mu_/(4*lambda_);
      return 0.5*mu_*(Ic - 3) +
         0.5*lambda_*(detF - alpha).powN(2) -
         0.5*mu_*log(Ic + 1);
   }

   // 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) - mu*logdetF +
         0.5*lambda*logdetF.powN(2);
   }

   // 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).powN(2);
   }

   // 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*(F.transpose()*R).trace() + 3) +
         0.5*lambda*(detF - 1).powN(2);
   }

   // 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);
   }
}
```