THESIS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

Automated Computational Modeling

JOHAN JANSSON

Department of Applied Mechanics
Chalmers University of Technology
Göteborg, Sweden 2006

Automated Computational Modeling
JOHAN JANSSON
ISBN 91-7291-809-8

Doktorsavhandlingar vid Chalmers tekniska högskola
Ny serie nr 2491
ISSN 0346-718x

Department of Applied Mechanics
Chalmers University of Technology
SE-412 96 Göteborg
Sweden
Telephone +46 (0)31-772 1000

Printed in Sweden
Chalmers Reproservice
Göteborg, Sweden 2006

AUTOMATED COMPUTATIONAL MODELING

JOHAN JANSSON

Abstract.

This thesis is part of the **FEniCS** project of *Automation of Computational Mathematical Modeling (ACMM)* as the modern manifestation of the basic principle of science: formulating mathematical equations (modeling) and solving equations (computation).

The vision of **FEniCS** is to set a new standard towards the goals of generality, efficiency, and simplicity, concerning mathematical methodology, implementation, and application. ACMM includes the key steps of Automation of (a) discretization of differential equations, (b) solution of discrete systems, (c) error control of discrete solutions, (d) optimization and (e) modeling. **FEniCS** is based on adaptive finite element methods (FEM) [1].

This thesis presents the following examples of *Automated Computational Modeling* as concrete realizations of ACMM with main focus on (a)-(c):

MG: Multi-adaptive Galerkin ODE-solver: This part concerns the automation of (a1) discretization in time by the MG implementation of the multi-adaptive ODE-solver mcG(q)/mdG(q) formulated in the thesis ([9]) by Anders Logg, based on Galerkin's method with continuous/discontinuous piecewise polynomial approximation in time of degree q with different time steps for different components, automatically determined by a posteriori error estimation. MG realizes automation of (a1) by Galerkin's method, (b) by fixed-point or Newton's method, and (c) by a posteriori error estimation using duality. MG is the first general multi-adaptive ODE-solver with automatic error control based on duality. MG has potentially a very vast range of applicability. MG may also be run in mono-adaptive form with the same time step for all components, eliminating the over-head required for multi-adaptivity. For bench-mark problems with different time scales, we demonstrate substantial performance gains with MG, as compared to mono-adaptive solvers. MG is joint work with Anders Logg.

Ko: Solid mechanics solver: This part includes the automation of (a2) discretization in space of a general continuum model for solid mechanics with elasto-visco-plastic materials and large displacements, rotations and deformations. When coupled with MG for time-discretization this gives the solid mechanics solver *Ko* realizing (a) and (b) with (c) in progress. Ko is based on an updated Lagrangian formulation where equilibrium and constitutive equations are expressed on the current deformed configuration. Ko is the first automated solid mechanics solver and the range of possible applications is very large. We show that the performance of Ko is comparable to that of a mass-spring solver, which is the industry standard for performance-intensive solid mechanics simulations [8]. Ko demonstrates the general capability and potential of **FEniCS**.

DOLFIN as a PSE: We present the **FEniCS** tool chain, and in particular **DOLFIN**, as a general and automated problem solving environment (PSE). **DOLFIN** realizes the overall concept of automated computational modeling by taking a PDE in mathematical notation as input and automatically discretizing and computing the solution by the FEM with full efficiency, including automated time discretization of time dependent PDE with the ODE solver. This is joint work with Johan Hoffman, Anders Logg and Garth Wells.

Automated Modeling: We present a case study of automated modeling (e) in a model problem with a fast and a slow time scale. By resolving the fast time scale for a short period of time an effective coefficient is determined by optimization, which allows simulation of the slow time scale over long time. This is joint work with Claes Johnson and Anders Logg.

Appended papers

The following papers are included in the thesis:

Paper I *A discrete mechanics model for deformable bodies* Computer-Aided Design 34 (2002)

Paper II *Algorithms for multi-adaptive time-stepping* Chalmers Finite Element Center Preprint Series no. 2004-13

Paper III *Simulation of mechanical systems with individual time steps* Chalmers Finite Element Center Preprint Series no. 2004-14

Paper IV *Computational Modeling of Dynamical Systems* Mathematical Models and Methods in Applied Sciences 15 (2005)

Paper V *DOLFIN: an automated problem solving environment*

Paper VI *Ko: a Fenics solid mechanics solver*

Acknowledgements

To my family Elsie-Britt, Erik and Lasse, and to my friends.

Special thanks to my supervisor Claes Johnson for encouragement, creativity and constructiveness, and to Anders Logg for great cooperation.

1. Introduction

This thesis is part of the **FEniCS** project [3] of *Automation of Computational Mathematical Modeling (ACMM)* as the modern manifestation of the basic principle of science:

(I) formulating mathematical equations (modeling),
(II) solving equations (computation).

The mathematical models of science and technology usually take the form of *systems of partial differential equations* such as

(i) Navier's equations: solid mechanics,
(ii) Navier–Stokes equations: fluid mechanics,
(iii) Maxwell's equations: electromagnetics,
(iv) Schrödinger's equation: quantum mechanics.

Each system of equation has the general form

$$A(\alpha, D, u, f) = 0 \quad \text{in } Q, \tag{1}$$

where A represents a system of partial differential equations over a doamin Q is space-time including boundary and initial conditions, α represents *coefficients*, D represents partial derivatives, f represents input e.g. in the form of applied forces and u is a function satisfying the system of equations. We view the system (1) as the *model* including the domain Q, the coefficients α and the input f as given *data*, which determine a *solution* u. Each of the set of equations (i)-(iv) describes in very concise form a basic scientific discipline with a wide variety of applications in different areas of technology.

The objective of the modeling (I) is to formulate the equation (1) including the specification of coefficients such as Lamé coefficients in solid mechanics, viscosity in fluid mechanics, permeabilities in electromagneticcs and potentials in quantum mechanics, as well as geometric data to specify Q. The objective of (II) is to compute the solution u for each given set of data.

The vision of **FEniCS** is Automation of Computational Mathematical Modeling (ACMM) towards the goals of generality, efficiency, and simplicity, concerning mathematical methodology, implementation, and application. Specifically, ACMM concerns automation of computational solution of general systems of differential equations including (i)-(iv), where automation signifies that after specification of model including data and error tolerance, an approximate solution within the error tolerance is automatically computed by a computer, ideally with minimal computational work.

The goal of ACMM is to provide solutions of general systems of differential equations on a lap-top with the same degree of automation as the computation of e.g. trigonometric functions using a pocket calculator. The comparison is natural since all the elementary functions including trigonometric functions, in fact are solutions to simple linear ordinary differential equations, and thus in a pocket calculator are determined by computational solution of the relevant equation when needed, instead of storing large tables of values. The difference is the scope with ACMM aiming at general nonlinear systems of partial differential equations, but the principle is the same.

ACMM includes the key steps of automation of:

(a) discretization of differential equations,
(b) solution of discrete systems,
(c) error control of computed solutions,
(d) optimization,
(e) modeling.

By *discretization* a given set of differential equations is translated into a discrete system of algebraic equations, which is solved using numerical algebra on a computer, to produce an approximation U of the solution u. The objective of *error control* is to assure that the difference $u - U$ is smaller than a given tolerance with respect to a given error measure. **FEniCS** uses the finite element method to automate discretization and duality-based a posteriori error estimation to automate error control. Automated optimization is realized by solving the set of differential equations expressing optimality using (a)-(c). Automation of modeling requires an Ansatz combined with (d). Automated modeling may concern computation of (effective) coefficients α in a model of the form (1) from best least-squares fit to measured or computed solutions u, often referred to as *inverse problems*. In principle, (d) and (e) can thus be reduced to (a)-(c), which accordingly represent the basic elements of ACMM.

The essential step of (a), which concerns both (a1) *time-discrization* and (a2) *space-discretization*, is automated computation of finite element stiffness matrices and assembly to a global stiffness matrix. This requires efficient evaluation of integrals of combinations of derivatives of finite element basis functions over finite elements. **FEniCS** achieves this in the Fenics Form Compiler (**FFC**) by factorization of the element stiffness matrix into a reference and geometry tensor. The reference tensor contains integrals over a reference element computed once, which for each element upon multiplication a geometry tensor gives the element stiffness matrix for each element. The input to **FFC** is then the equation (1) in standard mathematical notation and a given finite element mesh and finite elements, and the output is computer code (e.g. C++) specifying the discrete system.

The step (b) is automated in **FEniCS** using, with input from **FFC**, the parallel numerical linear algebra package PETSc.

The essential step of (c) is automated computation of (c1) discrete residuals and (c2) stability factors/weights by automated formulation and solution of a dual linearized problem. **FEniCS** currently achives these requirements partially, with a full implementation in sight.

The objective of this thesis is to contribute to the realization of the vision of **FEniCS** with the following concrete basic elements representing *Automated Computational Modeling*:

MG: Multi-adaptive Galerkin ODE-solver: This part concerns the automation of time-discretization (a1) by the MG implementation of the multi-adaptive ODE-solver mcG(q)/mdG(q) formulated in the thesis ([9]) by Anders Logg, based on continuous/discontinuous piecewise polynomial approximation in time of degree q with different time steps for different components, automatically determined by a posteriori error estimation. MG realizes automation of (a1) by Galerkin's method and (b) by fixed-point or Newton's method, and (c) by a posteriori error estimation

using duality with (c1) available and (c2) in progress. MG is the first general multi-adaptive ODE-solver with automatic error control based on duality. MG realizes automation of time discretization and thus has potentially a very vast range of applicability. MG may also be run in mono-adaptive form with the same time step for all components, eliminating the over-head required for multi-adaptivity. In this perspective MG can be viewed as a form of automated model reduction with large time steps for slow components and small time steps for fast components.

For bench-mark problems with different time scales, we demonstrate substantial performance gains with MG, as compared to mono-adaptive solvers. MG is joint work with Anders Logg.

Ko: solid mechanics solver: This part includes the automation of space-discretization (a2) of a general continuum model for solid mechanics including elastic, viscous and plastic materials and large displacements, rotations and deformations [10]. When coupled with MG for time-discretization this gives the solid mechanics solver *Ko* realizing (a) and (b) with (c) in progress. Ko is based on an updated Lagrangian formulation where equilibrium and constitutive equations are expressed on the current deformed configuration. Ko uses **FFC** for fast computation of the stiffness matrix in each time step. Ko is the first automated solid mechanics solver and the range of possible applications is very large. We show that the performance of Ko is comparable to that of a mass-spring solver, which is the industry standard for performance-intensive solid mechanics simulations. Ko has several advantages as compared to a mass-spring solver, such as automatic adaptive error control and specification of material properties, which in standard mass-spring models are performed ad hoc manually.

Ko may be viewed as a demonstration of the general capability and potential of **FEniCS**: The **FEniCS** code for Ko is less than 100 lines, and would require many thousands of lines of standard finite element code. Ko in particular demonstrates the ease of of coupling time-discretization with MG with space discretization with **FFC**. Another example is offered by the **FEniCS** Navier-Stokes solver by Johan Hoffman, opening to fluid-structure simulations with coupling to Ko.

DOLFIN as a PSE: We present the **FEniCS** tool chain, and in particular **DOLFIN**, as a general and automated problem solving environment (PSE). **DOLFIN** realizes the overall concept of automated computational modeling by taking a PDE in mathematical notation as input and automatically discretizing and computing the solution by the FEM with full efficiency, including automated time discretization of time dependent PDE with the ODE solver.

Automated modeling: We present an example of automated modeling (d) for a problem with a fast and a slow time scale. By resolving the fast time scale for a short period of time, an effective coefficient in a reduced model with only the slow time scale is determined, and the reduced problem is then solved with large time steps, with update of the effective coeffcient when needed. This allows the efficient solution over long time intervals without resolving the fast time scale, except over

small time intervals. The study has the character of a case study of a model case, with many possibilities of generalization usiung the general tools of **FEniCS**.

We now describe the main parts of the thesis in some more detail with focus on the basic aspects (a)-(c). We also indicate how the different appended papers fit into the general picture.

2. Multi-Adaptive ODE-Solver MG

A basic component of **FEniCS** is the multi-adaptive ODE-solver MG developed in cooperation with Anders Logg. MG automates the computational solution of general systems of ordinary differential equations (ODEs) of the form

$$\dot{u}(t) = f(t, u(t)) \quad \text{for } t > 0,\ u(0) = u^0,$$

where $f : R^{n+1} \to R^n$ is a given mapping, and u^0 a given initial condition. MG is based on continuous or discontinuous Galerkin methods in multiadaptive form with different time steps for different components adaptively determined by duality-based error control, referred to as mcG(q) and mdG(q) where q indicates the degree of the polynomial approximation in time.

Performance. We perform a benchmark experiment of solving a reaction-diffusion test problem with the multi-adaptive solver and a mono-adaptive solver. The benchmark is illustrated in figure 1 with the multi-adaptive step sequence in figure 2. Benchmark results can be seen in table 1.

N	$\|e(T)\|_\infty$	time	M	n	μ
1000	$1.8 \cdot 10^{-5}$	13.6 s	1922 (5)	4.0 (1.5)	95.3
2000	$1.7 \cdot 10^{-5}$	17.3 s	1923 (5)	4.0 (1.2)	140.5
4000	$1.6 \cdot 10^{-5}$	24.0 s	1920 (6)	4.0 (1.0)	185.0
8000	$1.7 \cdot 10^{-5}$	33.7 s	1918 (5)	4.0 (1.0)	218.8
16000	$1.7 \cdot 10^{-5}$	57.9 s	1919 (5)	4.0 (1.0)	234.0
N	$\|e(T)\|_\infty$	time	M	n	μ
1000	$2.3 \cdot 10^{-5}$	28.1 s	117089 (1)	4.0	1.0
2000	$2.2 \cdot 10^{-5}$	64.8 s	117091 (1)	4.0	1.0
4000	$2.2 \cdot 10^{-5}$	101.3 s	117090 (1)	4.0	1.0
8000	$2.2 \cdot 10^{-5}$	175.1 s	117089 (1)	4.0	1.0
16000	$2.2 \cdot 10^{-5}$	327.7 s	117089 (1)	4.0	1.0

Table 1. Benchmark results for mcG(1) (above) and cG(1) below for fixed tolerance tol $= 1.0 \cdot 10^{-6}$ and varying number of components (and size of domain). μ is an estimated efficiency index.

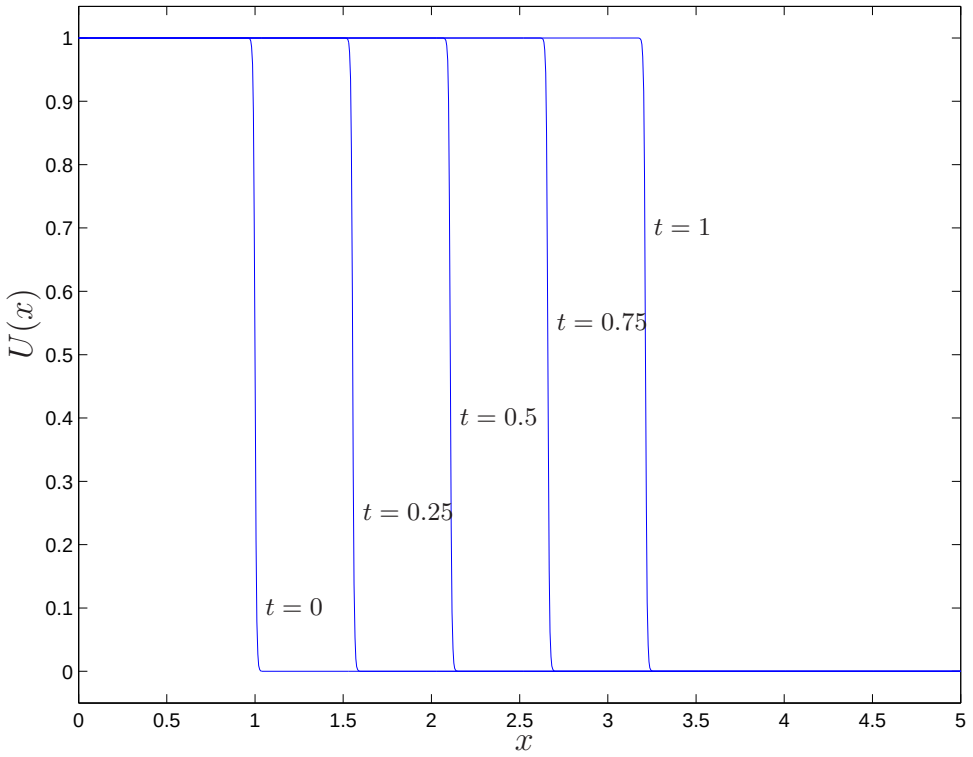


FIGURE 1. Propagation of the solution of a reaction–diffusion test problem.

Main Result. Consider an ODE with N_K slow components requiring a time step K and N_k fast components requiring a time step k for resolution to the given tolerance (or alternatively for stability reasons when using an explicit method). A standard mono-adaptive solver has to take time step k for all components. A multi-adaptive solver can take time step K for the slow components and time step k for the fast components. If N_K is much larger than N_k, this has a potential for a massive speedup, up to a factor $\frac{K}{k}$.

We describe the implementation of such a multi-adaptive solver and demonstrate dramatic performance gains due to multi-adaptivity, i.e. the solution is computed with the same error but much less work.

3. Solid Mechanics Solver

The world of solid mechanics is described by Navier's equations including an *equilibrium equation* expressing balance of forces according to Newton's 2nd law, a *constitutive model* connection stress to strain, or rates thereof and *boundary and initial conditions*. The equilibrium equation has a generic form, while the constitutive equation appears in many different forms expressing different material behavior including elastic, viscous, plastic and damage effects.

Our objective is to develop using the general tools of **FEniCS** an automated solver for solid mechanics simulations according to the following requirements:

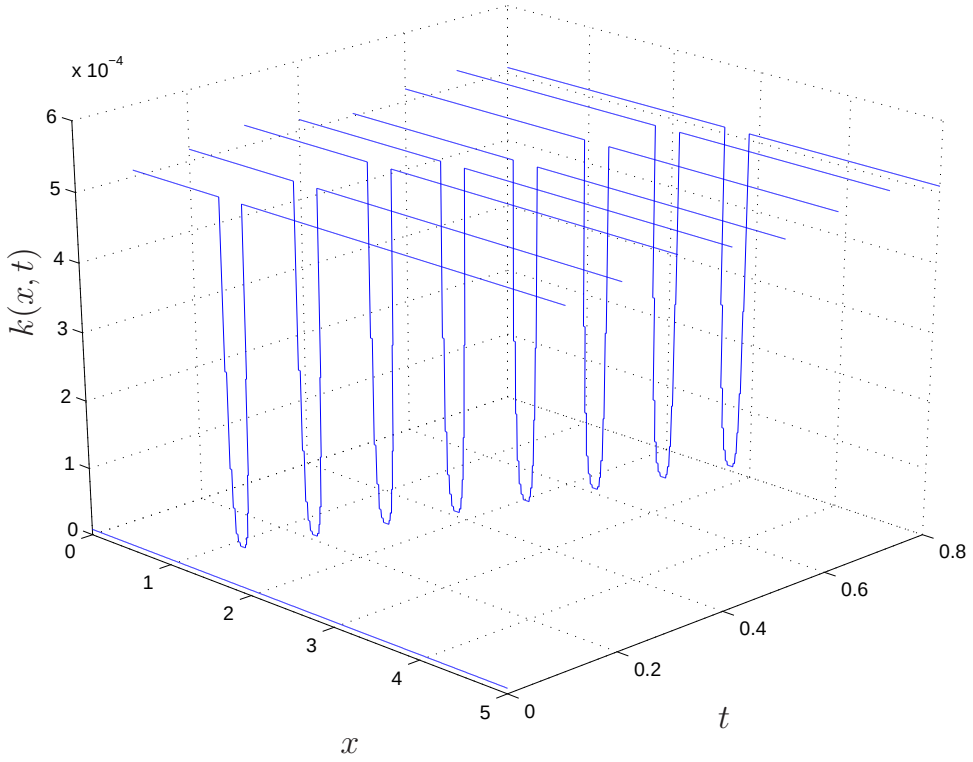


FIGURE 2. The multi-adaptive time steps as function of space at a sequence of time intervals for a reaction–diffusion test problem.

- large displacments, rotations and deformations
- elastic, viscous and plastic materials
- contact and friction boundary conditions
- performance allowing real-time simulation.

We may summarize the design specifications as those required in an advanced computer game based on realistic solid mechanics.

We restrict the constitutive model to a combination of elastic, viscous and plastic material behavior which can be described by Lamé coefficients, a coefficient of viscosity and a plastic limit, which we for simplicity take to be constant. Even with these limitations we can model a rich variety of materials, with variable coefficients presenting no additional difficulty beyond specification. We thus consider the model (1) to be given including data, and our task is to automate the computation of a corresponding approximate solutin U.

We use an *updated Lagrangian formulation* where the equilibrium equation and constitutive equation are expressed over the current configuration of a solid body subject to displacement, rotation and deformatin subject to some load. The constitiutive equation in general relates rates of stresses to rates of strain, but we also consider for purely elastic bodies a constitutive equation directly relating stress to strain. In the updated Lagrangian formulation with constitutive equation in rate form, the initial configuration of the body is not kept, and both equilbrium and constitutive equations are expressed over the current

configuration as well as boundary conditions. We refer to the **FEniCS** realization of this model as Ko.

Ko is realized using **FFC** for space discretization and MG for time discretization through the **DOLFIN** interface.

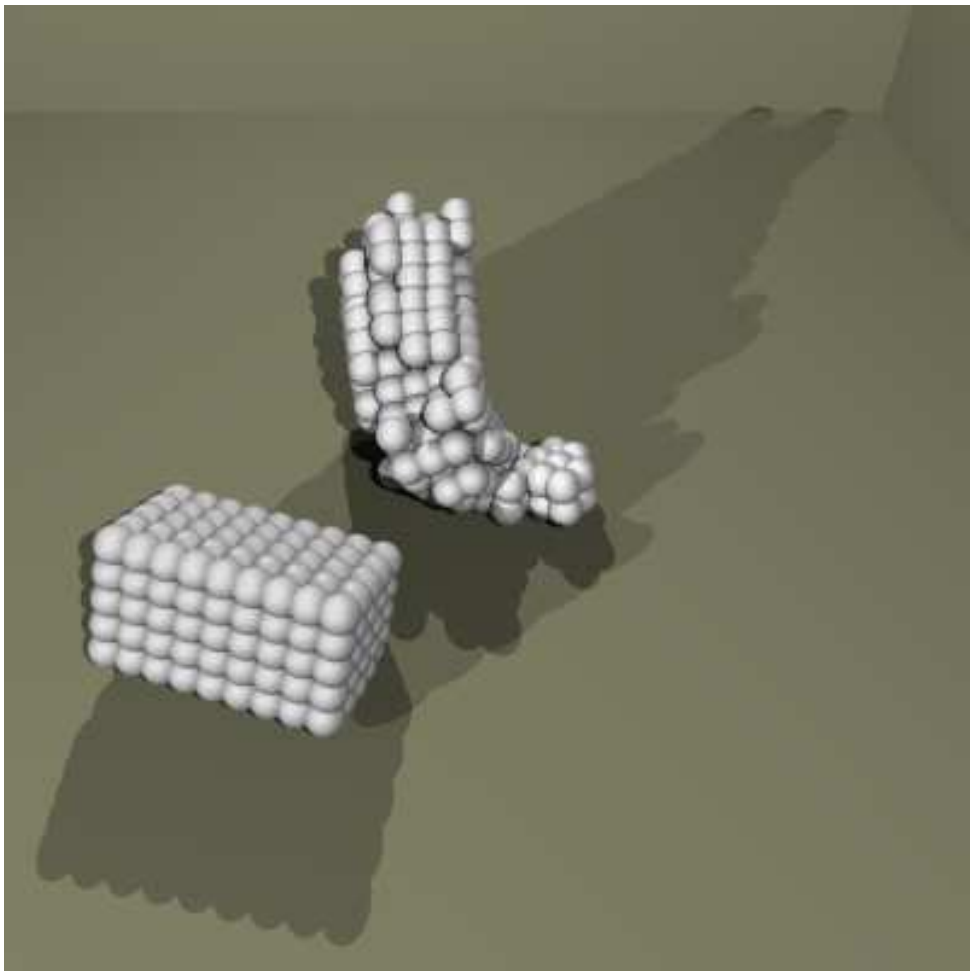


FIGURE 3. Example DOLFIN output of visco-elasto-plastic model solid mechanics model with contact, simulating a cow and a block being thrown in a room.

Performance. We compare the speed of Ko with that of mass-spring models which we have used in earlier work. We show that an implementation of a corresponding PDE model in Ko is only a factor 2-3 slower than a mass-spring implementation based on the same linear algebra data structures. This means PDE models in Ko could replace mass-spring implementations for performance-intensive applications.

Main Result. Ko represents a basic feature of **FEniCS**: We start from a mathematical model in concise form which is a basic general model of solid mechanics. We build the automated solid mechanics solver Ko using **FEniCS** where the essential part consists in

```
# Form representing the equilibrium equation of elasticity

name = "ElasticityDirect"
element1 = FiniteElement("Vector Lagrange", "tetrahedron", 1)
element2 = FiniteElement("Discontinuous vector Lagrange",
                         "tetrahedron", 0, 9)

q = TestFunction(element1) # Test function
dotv = TrialFunction(element1) # Trial function
f = Function(element1) # Body force
B = Function(element2) # Deformation measure

lmbda   = Constant() # Lame coefficient
mu      = Constant() # Lame coefficient

# Dimension
d = len(q)

# Manual tensor representation
def tomatrix(q):
    return [ [q[3 * j + i] for i in range(d)] for j in range(d) ]

Bmatrix = tomatrix(B)

def E(e, lmbda, mu):
    Ee = 2.0 * mult(mu, e) + mult(lmbda, mult(trace(e), Identity(d)))

    return Ee

ematrix = 0.5 * (Identity(d) - Bmatrix)

sigmamatrix = E(ematrix, lmbda, mu)

a = dot(dotv, q) * dx
L = (-dot(sigmamatrix, grad(q)) + dot(f, q)) * dx
```

FIGURE 4. Form for a total stress Euler-Almansi elasticity model.

specifying the variational form as well as data including coefficients and boundary/initial conditions. We demonstrate efficiency of Ko and thereby the power of **FEniCS**.

The Navier-Stokes solver of **FEniCS** developed by Johan Hoffman similarly automates the simulation of fluid flow including turbulent flow of fluids with small viscosity. Combining the Navier and the Navier–Stokes solvers we may simulate a rich world of fluid-structure interaction. The combined solver is now ready for implementation in **FEniCS**.

4. DOLFIN as a PSE

Traditionally problem solving has been equation-specific. An equation is selected, a method and solver is then derived or chosen specifically for that equation. For example, it is common to talk about a "Maxwell solver" or a "Navier-Stokes solver" which have been developed specifically for those equations.

When the equation is significantly changed, or a new equation is selected, the process needs to start from the beginning again. This implies much redundant manual work.

We present a free software tool called **DOLFIN** which combines generality with optimal efficiency. "Generality" means here that any equation can be input into **DOLFIN** essentially as it looks on paper. "Optimal" means here that **DOLFIN** is able to reach the same efficiency as a manually-developed solver for a specific equation.

DOLFIN is a component of the **FEniCS** tool-chain, where the role of **DOLFIN** is the problem solving environment, or programmer user interface for formulating and solving equations.

We illustrate the generality and efficiency of **DOLFIN** by presenting the following aspects:

Simple form language: We can input any equation into **DOLFIN** in mathematical notation, meaning that no extensive re-formatting or manual manipulation is required.

Assembly efficiency: Assembly is the forming of a discrete system (equation system for the degrees of freedom) given a discretization (finite element and mesh) of an equation. This is the key step for solving an equation. **DOLFIN** achieves generality and full efficiency by generating assembly code from a description of the equation.

High-level programming interface: Generality means that we should enforce a high level of abstraction, and this also applies to the programming interface. **DOLFIN** publishes a high-level programming interface in C++ and Python. The Python interface enables Just-In-Time (JIT) compilation of generated assembly code, which means that the code generation and compilation is transparent to the user of the interface.

PDE/ODE solver integration: DOLFIN provides capability for solving both initial value Ordinary Differential Equations (ODE) as well as Partial Differential Equations (PDE). We present a method which allows the general ODE solver in **DOLFIN** to be used for solving PDE by writing the PDE in the form $\dot{u} = f(t, u)$.

Applications: We present applications in incompressible fluid flow (Navier-Stokes' equations), see figure 5 for example output, and large deformation elasto-plasticity, see figure 4 for an example form and figure 3 for example output.

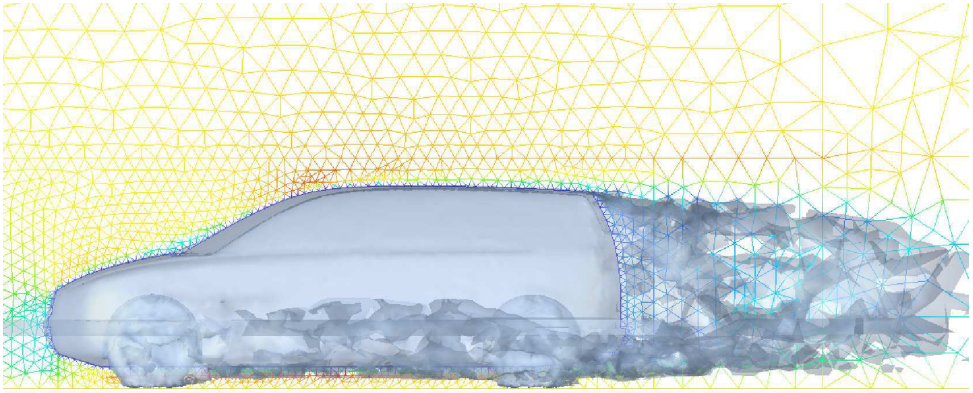


FIGURE 5. Example DOLFIN output of a turbulent incompressible Navier-Stokes model, simulating drag of air flow on a car. (Courtesy of Johan Hoffman) [4]

Main Result. We present automation advances which bring us closer to achieving the goal of solving an equation using **DOLFIN** (and indirectly **FEniCS** as a whole) using a minimal amount of manual work, while retaining the efficiency of a manually written solver.

DOLFIN (in the form of the Python interface) has very successfully been used in a PDE project course at Chalmers University of Technology. Students completed projects in streamline diffusion stabilization, Navier-Stokes' equations and acoustic equations (linearized Euler equations). The automated discretization **DOLFIN** enabled the students to focus on the forms instead of solver performance.

5. A Case Study of Automated Modeling

We again consider ODEs with multiple time scales. In this scenario the fast components are very fast compared to the slow components and time interval of the problem, and would be too costly to resolve directly. Instead, we construct a reduced model where we model the average effect of the very fast components on the rest of the system. This allows the system to be solved in a reasonable time. We show how the construction of the reduced model can be completely automated.

This is an example of (e) in the ACMM vision.

Main Result. We describe the process of automated construction of the reduced model and present an a posteriori error estimate for the modeling and discretization error.

We present two example problems with slow scales of order 1s and fast scales of order 10^{-6}s and 10^{-9}s on time intervals of $[0, 100]$. The automated modeling allows us to eliminate the very fast scales and only resolve the slow scales.

6. FEniCS

We present **FEniCS** [3], a free software [2] system for ACMM. The overall goal of ACMM is to build a computational machine which takes any PDE (in variational form) and a

tolerance for the error as input, and automatically computes a solution to the model which satisfies the tolerance.

FEniCS consists of the following components:

FIAT: FInite element Automatic Tabulator [5]. Automates the generation of finite elements. Provides representation of finite elements and evaluation of basis functions as well as general quadrature for integrating basis functions.

FFC: Fenics Form Compiler [6]. Automates the evaluation of variational forms. Provides assembly code generation and a form language for equation input.

FErari: Finite Element rearrangement to automatically reduce instructions [7]. Optimizes the evaluation of variational forms. Detects and exploits structure in element tensors.

DOLFIN: Dynamic Object oriented Library for FINite element computation. The programmer user interface for solving equations. Provides a high-level C++ and Python interface to:

- assembly
- variational form representation
- finite element representation
- function representation (typically a solution or coefficient)
- mesh representation
- linear algebra algorithms
- initial value multi- and mono-adaptive ODE solver
- file input/output

The assembly algorithm of a finite element method is typically of high complexity, and is not trivial to implement efficiently manually. A modification of the variational form or the choice of finite element normally means that large parts of the code need to be reimplemented. This is a waste of human resources and, due to the complexity of the algorithm, may easily introduce errors in the implementation.

The finite element assembly algorithm for a bilinear form $a(\cdot, \cdot)$ generating a matrix A can be formulated as follows: For each element K, add the local element matrix $A_{ij}^K = a_K(\hat{\phi}_i, \phi_j)$ to A, where $a_K(\cdot, \cdot)$ is the bilinear form restricted to the current element K.

FFC parses the form and, together with a description of the finite element, generates source code for evaluation of the local element matrix A^K. **FFC** (using FIAT) precomputes integrals on the reference element. **FFC** uses Ferari to exploit the structure of A^K to produce efficient code. This results in automatically generated source code which is as efficient as hand-written code. The generated assembly code is then linked into **DOLFIN** and can be accessed through the assembly interface.

7. Future

The potential of **FEniCS** has been demonstrated in the solid mechanics solver Ko and the Navier-Stokes solver.

We recall the key steps of ACMM, automation of:

(a) discretization of differential equations,

(b) solution of discrete systems,
(c) error control of computed solutions,
(d) optimization,
(e) modeling.

Today **FEniCS** realizes (a)-(b) fully, and (c) partially with a clear plan for full realization in sight. This opens new possibilities for efficient and reliable simulation of complex problems in many areas of science and technology. The **FEniCS** solid mechanics solver Ko and the Navier-Stokes solver demonstrate the potential.

The **FEniCS** plan for the immediate future includes Maxwell and Schrödinger solver and a variety of multi-physics solvers. **FEniCS** is a cooperative effort with a growing number of partners including Simula Research Laboratory and research groups at Delft University, Texas Tech, KTH, etc.

FEniCS is an open source project now reaching critical mass with some possibility of developing into "Linux for PDEs".

References

[1] K. Eriksson, D. Estep, P. Hansbo, and C. Johnson, *Computational differential equations*, 1996.
[2] Free Software Foundation, *GNU GPL*. `http://www.gnu.org/copyleft/gpl.html`.
[3] J. Hoffman, J. Jansson, C. Johnson, M. Knepley, R. C. Kirby, A. Logg, and L. R. Scott, *FEniCS*. `http://www.fenics.org/`.
[4] J. Hoffman and C. Johnson, *Applied Mathematics: Body and Soul*, vol. IV, Springer-Verlag, 2006. In press.
[5] R. C. Kirby, *FIAT: A new paradigm for computing finite element basis functions*, ACM Trans. Math. Software, 30 (2004), pp. 502–516.
[6] R. C. Kirby and A. Logg, *A compiler for variational forms*. submitted to ACM Trans. Math. Softw., 2005.
[7] R. C. Kirby, A. Logg, L. R. Scott, and A. R. Terrel, *Topological optimization of the evaluation of finite element matrices*. submitted to SIAM J. Sci. Comput., 2005.
[8] A. Liu, F. Tendick, K. Cleary, and C. Kaufmann, *A survey of surgical simulation: Applications, technology and edudcation*, Presence, 12 (2003).
[9] A. Logg, *Automation of Computational Mathematical Modeling*, PhD thesis, Chalmers University of Technology, Sweden, 2004.
[10] J. C. Simo and T. J. R. Hughes, *Computational Inelasticity*, Springer-Verlag, 2000.

Summary of Appended papers

Paper 1 **A discrete mechanics model for deformable bodies**

We describe an extended mass-spring model including contact. The example application is interactive physics-based geometric modeling for rapid prototyping in CAD. The mass-spring model is standard in computer graphics due to its simplicity and high performance. We use this model and its implementation as a comparison to PDE models and implementations presented in the later papers.

Paper 2 **Algorithms for multi-adaptive time-stepping**

We describe the main algorithms for multi-adaptivity: construction of time slabs and the solution of the resulting equation systems by fixed-point and Newton's

method. Further, the implementation of such a multi-adaptive solver is presented and we demonstrate dramatic performance gains due to multi-adaptivity, i.e. the solution is computed with the same error but much less work.

PAPER 3 **Simulation of mechanical systems with individual time steps**

The application of multi-adaptivity to mass-spring systems is covered in this paper. The aim is to show how to apply multi-adaptivity to practical applications.

PAPER 4 **Computational Modeling of Dynamical Systems**

This paper consists of a case study of automated modeling. The contribution of the thesis author is not major but the paper illustrates automated modeling - a key element of ACM and **FEniCS**.

PAPER 5 **DOLFIN: an automated problem solving environment**

We present the **FEniCS** tool chain, and in particular **DOLFIN**, as a general and automated problem solving environment (PSE). **DOLFIN** realizes the overall concept of automated computational modeling by taking a PDE in mathematical notation as input and automatically discretizing and computing the solution by the FEM with full efficiency, including automated time discretization of time dependent PDE with the ODE solver.

PAPER 6 **Ko: a Fenics solid mechanics solver**

In this paper we present a general continuum model for solid mechanics with elasto-visco-plastic materials and large displacements, rotations and deformations which model-wise is a much improved replacement for the mass-spring model. We describe how to implement the model using **FEniCS** and show that the implementation complexity and performance of the PDE model is comparable to the mass-spring model, while retaining the advantages of a PDE model: generality and spatial error control. The model is also automatically discretized in time using the multi-adaptive ODE solver in **FEniCS**.

ELSEVIER

Computer-Aided Design 34 (2002) 913–928

COMPUTER-AIDED DESIGN

www.elsevier.com/locate/cad

A discrete mechanics model for deformable bodies

J. Jansson*, J.S.M. Vergeest

Faculty of Design, Engineering and Production, Delft University of Technology, Delft, The Netherlands

Abstract

This paper describes the theory and implications of a discrete mechanics model for deformable bodies, incorporating behavior such as motion, collision, deformation, etc. The model is fundamentally based on inter-atomic interaction, and recursively reduces resolution by approximating collections of many high-resolution elements with fewer lower-resolution elements. The model can be viewed as an extended mass-spring model. We begin by examining the domain of conceptual design, and find there is a need for physics based simulation, both for interactive shape modeling and analysis. We then proceed with describing a theoretical base for our model, as well as pragmatic additions. Applications in both interactive physics based shape modeling and analysis are presented. The model is aimed at conceptual mechanical design, rapid prototyping, or similar areas where adherence to physical principles, generality and simplicity are more important than metric correctness.

Keywords: Mechanics model; Deformation; Collison; Deformable bodies; Geometric modeling; Conceptual design; Virtual claying

1. Introduction

The context of this paper is conceptual mechanical design. It is well known that the conceptual stage of the design process still is largely unsupported by computer tools. We believe that such support can greatly increase efficiency, by, for example, providing rapid verification of early design ideas, or support for quick description and modification of non-detailed 3D geometry, i.e. the equivalent of sketching.

There are many possible aspects of such support: natural interaction, vague description, virtual environments, physical simulation, rapid prototyping, etc. Our research group, the Integrated Concept Advancement (ICA) Group [17] is researching some of these areas. This paper will focus on physical simulation. We will describe a mechanics model and implementation that can be used for geometric modeling and physical analysis in the conceptual design phase.

For computer tools to actually support, instead of hinder this phase, several requirements must be met. There have been studies determining what these requirements are [8,32]. The relevant requirements posed for this specific domain are:

1. natural interaction methods—interaction with virtual objects should be experienced as interaction with real objects;
2. rapid feedback and evaluation—interaction should not be hindered by poor resolution and time lags.

Notably missing from the list of requirements is metric accuracy. This is the main difference compared to traditional CAD applications, we can sacrifice metric accuracy for other properties; interactivity for example. However, this does not give as much freedom as might seem. We still need to make sure that the phenomena we are trying to simulate are physically correct, especially if we want to perform physical analysis.

We have developed a mechanics model that is particle system based, and in which accuracy is dependent on the resolution of description. We reason recursively, the model defines lower-resolution elements which approximate a collection higher-resolution elements. This means that we can employ induction as a means of making sure the model is physically correct. If we can show that some ideal resolution is physically correct, and show that the operation of reducing resolution some given step does not remove this property, we can assume that low resolutions will also have this property. With this, we are not aiming for a formal proof of correctness, but want to show the philosophy of the model. Also, this kind of reasoning is very dependent on the definitions of the terms we are using. It is evident that each operation of reducing resolution removes some kind of correctness, we just want a guarantee that we are not removing any physical principles.

* Corresponding author.
E-mail address: j.jansson@io.tudelft.nl (J. Jansson).

0010-4485/02/$ - see front matter
PII: S0010-4485(01)00146-4

Assuming we have such a model that covers deformable bodies, we can consider various applications. The most direct application is analysis. During the early stage of design, it can be desirable to get indications of what kind of physical implications some given design directions will have, essentially a pruning of the design space. Ford Motor Company has expressed such a need in a paper [25]. Apparently there is a need for simulation data at the early design stage of new cars, and traditional simulation is too expensive and time-consuming to be applicable. Although the approach described in that paper is of a different type than what we propose [it proposes a Design of Experiment (DoE) approach], it clearly shows a need for such analysis. Even if our approach at this stage might not be suited for the car industry, we can assume that the same need exists in other related industries.

Another perhaps more interesting application is virtual environments. Given the coverage of the model (deformable bodies), and an interface that can present a user as a body in the environment, we can reformulate geometric modeling as a mechanics simulation problem instead of as a mathematical geometry problem. For example, if we can simulate bodies of some appropriate modeling material, such as clay or foam, and we can present the hands of the user as bodies in the environment, all the user has to do is manipulate the material with his/her hands, presumably a familiar process. Our job is then to make sure the properties of the simulation fulfil the requirements of such a manipulation process. This application requires less adherence to physical principles, and we can, for example, introduce artificial operations, which have no real physical base, to take some load off the model. This type of application is normally referred to as 'virtual claying'.

2. State of the art

The mechanics of deformable bodies is in no way a new research area. There exist numerous models aimed at various applications.

At least one attempt has been made to augment a rigid body model with a special module for deformable body collision [2]. It applies a two-phase model, where the first phase prevents inter-penetration, and the second phase calculates contact forces. Deformations are constrained to what can be represented by a global deformation function, which avoids the problem of calculating impulse propagation. It is not clear, however, how general this approach is. The authors also state that allowing complex deformation functions will lead to a heavy computational burden.

Particle models are often used where flexibility is needed with regard to the phenomena modeled. We use the term 'particle model' to distinguish what the computer graphics community calls a 'particle system' from the term used in physics. A particle model is a particle system with possible additional rules. Originally used to model smoke and fire phenomena, the models have developed to cover geometric modeling [29]. Strictly speaking, most of the other models mentioned could be denoted particle models.

Another well-established model is the mass-spring model. Provot has described a model used for cloth simulation [27], and Chen et al. a model aimed at general objects [5]. It describes bodies as sets of point-masses, and the materialistic properties as a graph of springs over these sets (essentially a particle model as well). While the model is very useful for describing deformation of a single body, it does not cover collision at all, since there exists no concept of volume. Computationally, it can give rise to stiff differential equations, for very stiff springs, for example, which requires finer time discretization, and thus more computation.

The primary tool for mechanics simulation in engineering analysis is called Finite Element Analysis (FEA) [1]. However, there is inconsistency in the literature about the definition and scope of this term. The term is derived from the term Finite Element Method (FEM) of analysis. It is the term FEM that is inconsistently used.

In Popov [26] (p. 104), the FEM is described as 'More recently a powerful numerical procedure has been developed, where a body is subdivided into a *discrete number* of finite elements, such as squares and cubes, and the analysis is carried out with a computer'. In Heath [13], it is described as: 'Finite element methods approximate the solution to a boundary value problem by a linear combination of basis functions ϕ_i, typically piecewise polynomials, which for historical reasons are called *elements*'. The former definition is commonly used in engineering discussions, while the second is used in mathematical and numerical method texts.

The difference between the (informal) definitions, and what is causing the confusion, is that the first definition is a general discretization of a body, while the second is a method for solving boundary value differential equation problems, by discretizing the solution function in a particular way. Additionally, neither definition tells us anything about which physics model (what assumptions, etc.) is used.

Originally, and still principally, FEA refers to statics [24], and this is the definition we will adopt. A typical statics problem results in a boundary value problem, which can then be solved using the FEM. A basic dynamics problem on the other hand, results in an initial value problem, for which the FEM does not apply. When referring to an analysis method, it is preferable to refer to the physics aspect of the analysis instead of the numerical aspect. For instance, the FEM can be used to solve heat transfer problems, which have no relation to solid mechanics. To group such differing analyses under the term FEA causes ambiguities.

Terzopoulos et al. have developed a Lagrangian mechanics model aimed at animation [30]. They use continuous bodies, and a combination of boundary value (finite difference) and initial value methods. They treat elastically deformable bodies, and also collisions, which they handle by creating a force field around each body.

In Terzopoulos and Fleischer [31], they extend this model with plasticity, and state an aim similar to ours: 'We envision users, aided by stereoscopic and haptic input–output devices, carving 'computer plasticine' and applying simulated forces to it in order to create free-form shapes interactively.' While we have not treated plasticity in our model, their treatment of plasticity is likely to be directly applicable in future development.

Kang and Kak [21] have developed an FEA system to create a geometric modeling system. The system presents the designer with an initial physical shape, represented by the FEM mesh. The user can then utilize a force-input interface, in their case a four-sensor plate, to manipulate nodes of the shape. The system could presumably be generalized to allow arbitrary input methods.

James and Pai use the Boundary Element Method (BEM) to create a virtual modeling environment [18]. This method is more suited to pure interactive deformation applications than the FEM due to only considering the boundary, and thus requiring less computation.

3. Mechanics model

3.1. Basic theory

We start building our theory at the atomic level. We know that any given body is made up of a large number of atoms, so if we can know the behavior of each atom, we will know the behavior of the body as a whole. An atom can be considered as a particle, a point mass. Between any given atom pair we have a central force, determined by the distance, and other properties of the atoms. This means we have a particle system, an entity that is quite simple to treat. If we have several bodies, we have several particle systems, which together can simply be treated as one particle system.

Now, we do not want to have an atom as the basic element in our model, to build any kind of useful bodies will require too many elements. However, we can make an approximation to overcome this. If we consider a solid body, it consists of many atoms close together, with neighboring atoms behaving in a similar way. If we were to treat every $2 \times 2 \times 2$ matrix of eight atoms in the body as a single element, we would end up with eight times less elements to treat. These new elements would in turn form a particle system, and through induction, we could keep reducing the resolution until we have a manageable number of elements. For this to be possible, we need to show that we can approximate a $2 \times 2 \times 2$ matrix of atoms, a particle system, as one single particle.

First of all, we need to formalize some of our statements. We have said that, in a solid body, 'neighboring atoms behave in a similar way'. Formally, what we mean by this is that two neighboring atoms have the same properties, and that the external force, the force due to all other atoms in the system, on two neighboring atoms can be approximated as being equal. We then take this further to apply to a $2 \times 2 \times 2$ matrix of atoms.

Particle system theory states that all forces that act on a particle in a particle system can be divided into two sets: internal forces, which stem from other particles in the system, and external forces, which stem from outside the system. This distinction is made because internal forces balance out, and do not affect the center of mass of the particle system. Thus, the center of mass motion can be determined strictly by taking external forces into consideration.

If all particles have the same external force applied to them, and all particles have the same properties, the result on the center of mass will be the same as if we treat the system as one particle, with the sum of the external forces applied to it, and with the sum of all the properties added to it. It is also clear that with this arrangement, all torques with regard to the system's center of mass cancel out, and the angular momentum of the system is conserved.

What we have done with this approximation is to remove resolution from the description of a body. Since an element of the body must, by our definition, be homogenous, we cannot have different forces acting on different parts of the element, as would be the case if the element was decomposed into its original parts. We will have to take this into consideration when we consider what kind of forces are acting between two elements. For example, interatomic forces have components that are only significant for a very small distance. If our elements are significantly larger than this distance, our approximation errors will be very large. We can, however, compensate for this by trying to find some sort of average force over the entire element. However, we have not examined this topic very closely yet, and our force models are still very simplified.

3.2. Model description

In the previous section, we describe how we can reduce the complexity of the atomic configuration of a body into larger and fewer 'elements'. We now need to describe how such elements interact, and more formally describe the components of the model.

We started off by examining how atoms are configured in a body, and we need to apply the same reasoning to determine how our elements interact. Since we have removed much of the resolution from the configuration of elements, we introduce an entity for interaction which allows us to retain some of the complexity. We say there is a 'connection' between two elements when they are interacting, and that such a connection has some state associated with it. We are also free to determine when an element pair will start interacting, and when it will stop. This can allow us to overcome some of the lack of geometric shape of an element.

With such an entity, we can define a force between a pair of elements which is not only dependent on some instantaneous distance or parameter of the elements, but also on some parameter of the connecion, which we can deduce through other means. For example, although the contact force between two bodies and the strain force inside a

body both physically stem from the same inter-atomic forces, the structure of the atoms inside a body might be different from that seen in the interaction between two bodies. We could describe that using our connections, thus creating a different force function between elements known to be fused to one another, and elements of different bodies.

Based on what we know of the micromechanics of inter-atomic behavior, and of some of the more macroscopic mechanics of bodies, we have formulated several forces we can use in our model.

3.2.1. Gravitation

First of all, we have a gravitational force. If necessary, we can describe a detailed gravitational force model where each element determines the force on every other element. However, normally it is sufficient to have a uniform gravitational force, where only one mass is the source of a gravitational field.

$$F_g = G\frac{m_1 m_2}{r_2} \tag{1}$$

where G is gravitational constant, m_1, m_2 are masses, and r is distance.

3.2.2. Elasticity and fracture

To determine inter-element forces, we start off by examining a graph of the inter-atomic forces (see Fig. 1 [23]). We can see that there exists a distance where the force is zero, and that the force becomes repulsive when decreasing the distance, and attractive when the distance is increased. If the distance is increased beyond a limit, the force decreases to insignificance. We can model this by using a simple Hooke formulation, and two threshold distances. One distance determines when two elements are close enough so the force is significant enough to be taken into consideration, and another determines when two elements have receded far enough from each other to no longer significantly interact.

$$F_c = -k(d - l) \tag{2}$$

where d is actual distance, k is a Hooke constant, and l is the nominal distance.We now have the core model. See Fig. 2 for a schema of two elements, a connection, and important quantities. We can now construct arbitrarily shaped bodies, in arbitrary initial states, and simulate the behavior. We will see, however, that such systems do not behave very well. In reality, most actions and interactions inside and between bodies involve non-conservative processes which damp the motion of such systems. As no such processes exist within our model, the systems will simply oscillate eternally, or more probably, if solved numerically, oscillate divergently and 'explode'.

Therefore, we need to identifyn and find a way to incorporate, such processes into our model. Unfortunately, such processes are not as simple as what we have seen so far, and need more heuristic and approximative methods.

There are three easily identifiable non-conservative

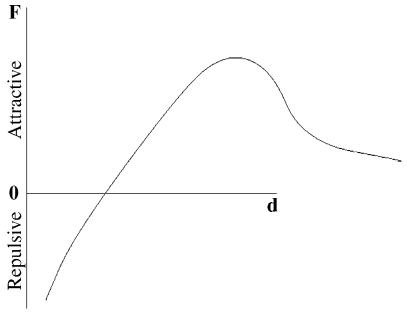


Fig. 1. Sketch of the inter-atomic force function.

processes we can observe: internal damping (compress and release a foam ball), sliding friction (run and fall down) and ambient viscous friction (throw a foam ball into the air at high speed). However, there does not exist any simple general models for these processes, the models that exist are only based on very special cases. Regardless, as long as they convey a reasonable approximation of the process, they are useful.

3.2.3. Internal damping

We damp the linear inter-element force with viscous friction, oriented along the elongation velocity vector:

$$F_d = -b\|\vec{e}\| \tag{3}$$

where b is the damping constant, and $\vec{e}$ is the elongation velocity.

3.2.4. Sliding friction

We use the standard sliding friction model, oriented along the velocity vector component normal to the normal force vector:

$$F_f = -\mu\|\vec{N}\| \tag{4}$$

where $\vec{N}$ denotes normal force, and μ the friction constant.

While the empirically found friction constants are only valid for interaction between two specific materials, we

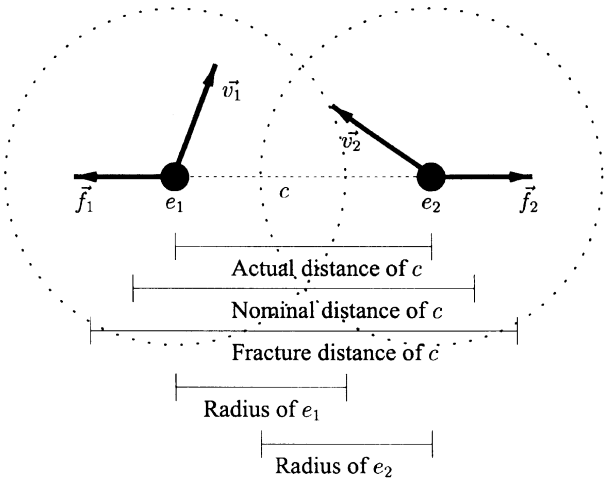


Fig. 2. Schema of two elements, a connection, and important quantities.

simplify this a bit, and define a friction constant for every element. When two elements interact, we average the constants for the friction force between the elements.

This force is separated from the other forces in that it is not central. We have to be careful when using it, because in conjunction with our approximation, it can lead to unexpected behavior. For example, in a body which is compressed and rotated, 'sliding' occurs between internal elements, producing a sliding friction force. While this behavior does not violate the model, it may not be what is expected.

3.2.5. Ambient viscous friction

We use the model for fluid resistance at high speed, oriented along the velocity vector:

$$F_v = -\pi r_e^2 \rho \|\vec{v}_e\|^2 \tag{5}$$

in which ρ is medium density.

3.2.5.1. Element. An element e is a set of parameters $\{\vec{p}, \vec{v}, b, m, r, k, t, \mu, C\}$.

$\vec{p}$	Position
$\vec{v}$	Velocity
b	Damping constant
m	Mass
r	Radius
t	Fracture distance
μ	Friction constant
C	Set of connections connected to the element

3.2.5.2. Connection. A connection c is a set of parameters $\{e_1, e_2, b, k, l, t, \mu\}$.

e_1, e_2	Elements comprising the connection
b	Damping constant
k	Hooke constant
l	Nominal distance
t	Fracture distance
μ	Friction constant

Connections are dynamically created and destroyed when elements start interacting and stop interacting, respectively. The parameters of a newly created connection are calculated from the parameters of the elements it connects. This is why the element primitive share some parameters with the connection primitive. Exactly how the new parameters should be calculated remains to be determined. Presumably, the radii of the colliding objects should also be taken into consideration. For now, however, we simply average the respective parameters for the new connection.

We are now satisfied with the components of the model. We can reasonably correctly simulate most phenomena observed in systems of deformable bodies. We will now proceed to show how we can numerically solve such systems defined within this model.

4. Implementation

4.1. Numerical solution

The nature of the inter-element forces in the physical model may provide for difficulties in mathematical treatment. Since we externally control the force functions, we control the connection entities through a state machine, as the system develops over time, formally we should include this state machine in our functions. However, piecewise in time, the state (the connections) remains the same, and we do not have to take this into consideration. Thus, we can describe the system as a series of differential equation systems, where the initial state of each system is determined by the state of the previous system, and by a state machine.

The mathematical formulation for each single system is quite simple. We have a standard particle system that we want to develop in time.

We can define the force on a single element e in the system:

$$\vec{F}_e = \vec{F}_C + \vec{F}_G + \bar{F}_L \tag{6}$$

where $\vec{F}_C$ is the sum of all the connection forces, $\vec{F}_G$ is the sum of all global forces, and $\vec{F}_L$ is the sum of all 'local' forces, i.e. forces depending only on the state of the element itself.

We then use Newton's second law, $F = mp''$, to produce a system of second-order ordinary differential equations:

$$\vec{p}_e'' = \frac{\vec{F}_e}{m_e} \tag{7}$$

To simplify solution, we want to transform our system into a new system of only first-order ordinary differential equations. We define two new vector functions:

$$g_1 = \vec{p}_e \tag{8}$$

$$g_2 = \vec{p}_e' \tag{9}$$

We now have a new system:

$$g_1' = g_2 \tag{10}$$

$$g_2' = \frac{\vec{F}_e}{m_e} \tag{11}$$

Thus, solving (11) produces g_2, which we can use in (10) to produce g_1, which is equal to $\vec{p}_e$, the solution to (7).

We can now solve this system using our preferred numerical method. We have to keep in mind that mathematical treatment may encounter difficulties, however, due to the usage of this state machine. As a start we choose Euler's method. It has proven to be practically usable, so even if we never manage to apply any other methods, we can still create a practical implementation.

The required step size is dependent on a number of factors. We are not so much interested in correctness as in

stability. We can reason that the Euler method (as well as most other numerical methods) works by sampling the state of the system, and then extrapolating current state to produce the next state. Since the extrapolation necessarily will bound state changes in our system, such as a collision, we have to make sure we do not extrapolate too far. This means that the step size is dependent on the velocity, position and radius of the elements in the system. Since our method is not adaptive, we must choose a step size in advance, which will correctly handle the most extreme event in the simulated sequence. Step size will also be dependent on the stiffness of our connections. This is, however, related to the previous attributes (velocity, etc.), as a connection only can provide an acceleration.

As we specify the initial state of the system when we describe what we want to simulate, we have all the information we need to start the Euler iteration. All we now need to do is to define $\vec{F}_e$ formally. We define E as the set of all elements in the system. The subscript e refers to the element we are calculating the force on. We also define an informal order in the set so we can iterate through it. We define each component separately.

4.1.1. Local force (ambient viscous friction is the only force)

$$\vec{F}_L = -\pi r_e^2 \rho \|\vec{v}_e\|^2 \frac{\vec{v}_e}{\|\vec{v}_e\|} \tag{12}$$

4.1.2. Global forces (gravitation is the only force)

$$\vec{F}_G = \sum_{i=0}^{|E|} G \frac{m_e m_i}{\|\vec{p}_e - \vec{p}_i\|^2} \frac{\vec{p}_e - \vec{p}_i}{\|\vec{p}_e - \vec{p}_i\|} \tag{13}$$

(Normally we only define one body as a gravitational source, to reduce computation, or a uniform gravitational field.)

4.1.3. Connection force

As the friction force depends on the other components of the inter-element force (which define the normal force of the friction equation), we need to further subdivide the inter-element force. For clarity, we simply use the logical components we have already defined:

$$\vec{F}_C = \vec{F}_b + \vec{F}_d + \vec{F}_f \tag{14}$$

where $\vec{F}_b$ is the original inter-element force definition, $\vec{F}_d$ is the damping force, and $\vec{F}_f$ is the friction force.

To simplify notation, we define the subscript e as we have done before, and a new subscript p as the opposite element in the connection. We iterate over the connection set C of the element:

$$\vec{F}_b = \sum_{i=0}^{|C_e|} -k_c(\|\vec{p}_e - \vec{p}_p\| - l_c) \frac{\vec{p}_e - \vec{p}_p}{\|\vec{p}_e - \vec{p}_p\|} \tag{15}$$

We define the relative velocity of the two elements in the connections as two components, one parallel to the connection, and one orthogonal:

$$\vec{v}_{\|} = \frac{(\vec{v}_e - \vec{v}_p)\cdot(\vec{p}_e - \vec{p}_p)}{\|\vec{p}_e - \vec{p}_p\|^2}(\vec{p}_e - \vec{p}_p) \tag{16}$$

$$\vec{v}_{\perp} = (\vec{v}_e - \vec{v}_p) - \vec{v}_{\|} \tag{17}$$

Then:

$$\vec{F}_d = \sum_{i=0}^{|C_e|} -b_c(\vec{v}_{\|}) \tag{18}$$

The sum of these two forces could be called 'contact force' in certain contexts. They form the normal force in the friction definition,

$$\vec{F}_N = \vec{F}_b + \vec{F}_d \tag{19}$$

We can now define the friction force:

$$\vec{F}_f = \sum_{i=0}^{|C_e|} -|u_e \vec{F}_N| \frac{\vec{v}_{\perp}}{\|\vec{v}_{\perp}\|} \tag{20}$$

4.2. Algorithms and performance

Before we discuss performance and efficient algorithms, we make some assumptions about the state of the system to remove the need to treat degenerate cases.

The state of the system consists of a set E of elements, and a set C of connections. The set of connections form a graph over the set of elements. For $|E| = n$, we have that the minimum of $|C|$ is 0 and the maximum is n^2. However, for the applications we have in mind, we make the assumption that there exists a structuring on the elements so that they are well-separated. This means that any given element has a number of connections that can be bounded by a constant independent of n. For instance, according to our original argumentation about elements, we formed elements from a 3D matrix of smaller elements. If we create connections between vertical, horizontal and diagonal neighbors in the matrix, we end up with $3 \cdot 3 \cdot 3 - 1 = 26$ neighbors, which is also the number of connections per element. Since each connection consists of two elements, we will have $13n$ connections for this example. During simulation, this may not be true locally, but for non-degenerate situations, we should be able to find a constant which can bound the number of connections.

4.2.1. Force calculations

The force calculations consist of simply performing the arithmetic as dictated by our force definitions. We have two kinds of force calculations, one that calculates a component of the force of a connection, and one that calculates a force component of a global force. Thus, for each connection force component, we have to make one calculation per

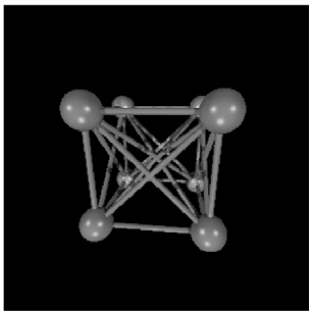


Fig. 3. Illustration of how four elements are interconnected (size of elements have been shrunk to prevent occlusion).

connection, and for each global force component, we have to make one calculation per element. According to our previous assumption of well-separatedness, both of these are $\Theta(n)$.

Numerical integration consists of calculating a new state of the system for every time step, as has been described. We have one equation per element, so this algorithm also is $\Theta(n)$.

4.2.2. State machine operations

The operations by the state machine so far only include determining when connections should be created and destroyed, and what properties they should have. This is a more complex computation. To determine when a connection should be destroyed, we simply have to perform a test for each connection, and if its elements are a distance t apart, we destroy it. Thus, this operation also is $\Theta(n)$. However, the opposite operation does not. Given a distance r for every element, we create a connection when two elements are within $2r$ from one another (and a connection does not already exist). The brute force algorithm compares every element with every other element, so this algorithm is $\Theta(n^2)$. We will see we can do better than that.

This problem can be formulated as the well-known collision detection problem. Given a set of shapes, find all pairs which intersect. Although we can never find an algorithm better than $O(n^2)$ in the general case, since there might exist $O(n^2)$ collision pairs, and we somehow need to find them, we can isolate such cases. There are several algorithms which are significantly more efficient than the brute force algorithm. One is based on dimension reduction [6], and is $O(n\log n + m)$, where m is the number of shape pairs which are 'very close'. We can also apply hierarchical space partitioning [20], which while difficult to prove, empirically performs as $O(n\log n)$.

4.2.3. Global performance

If we add our complexities together, we get $O(n\log n)$ (the most expensive complexity). We have performed empirical experiments that indicate this behavior [20]. For absolute performance, we refer to the same paper. We show that up to ca. 500 elements, with semi rigid material properties, we can achieve real time performance on standard PC hardware.

4.3. Creating bodies

Before we can do anything practical with the model, we need to specify the state of the model, i.e. the elements and connections and their properties. Our aim is to be able to take a standard solid CAD model as input to our system. If we look back at how we defined an element, we can see that it is quite straightforward to translate a given geometric description of a solid body into the physical representation of the model.

The geometric description of a solid body defines the volume of the body. The volume of a body is simply the union of all the atoms in the body. Since an element in our model simply is a spherical approximation of a large number of neighboring atoms, we can easily create a translation.

First of all, we need to decompose the geometric description into polyhedra, each which must be approximable by a sphere. These form our elements. We then determine the topology of the spheres from the topology of the polyhedra. From this topology, we can determine which spheres are connected. If we assume the body initially is in its rest shape, we specify the nominal distances of the connections so that there is no strain energy, concretely, we specify the nominal distances to be the actual distances between the elements. Since a geometric description has no physical attributes, we cannot determine any other parameters of the elements or connections from this description alone, but need extra information.

In our implementation, we can translate solid polygonal representations into the physical representation of the model (any solid representation should be translatable with this method). First of all, we convert the polygonal representation into a voxel representation by sampling the polygonal representation with an inside/outside function. We then generate an element for each voxel, with a diameter no more than twice the voxel width (so two neighboring elements can intersect, but no further). Connections are generated depending on the topology of the voxel matrix. We say that the neighbor of a voxel is any voxel which shares a vertex, and a connection is created for each neighbor. Fig. 3 illustrates how four voxels have created four elements (with shrunk diameters to prevent occlusion), with connections.

We can demonstrate a practical example. Say we have a boundary description of a part as described in Fig. 4. We can then generate a physical description using the previously described method to generate the description as shown in Fig. 4.

4.4. Interface

When practically using the model, it is normally not enough for the model to be isolated, we somehow need

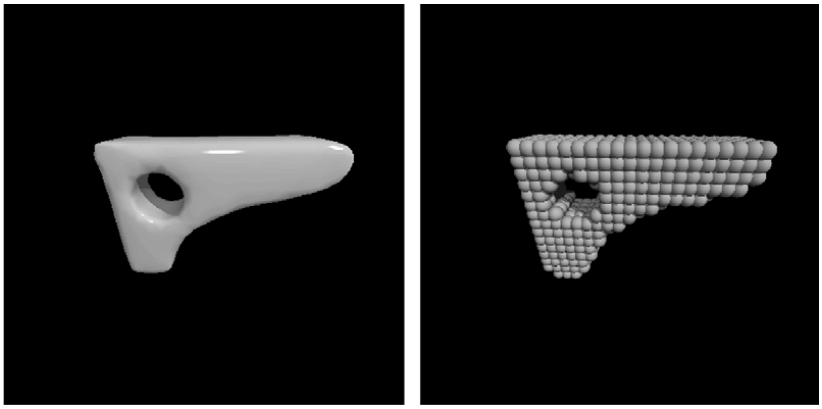


Fig. 4. Boundary and physical representation of a part (a support).

ways to interface with it, either interactively in real-time, or with other simulation systems. If the model publishes a standard interface, we can create a modular system, where models and devices easily can be replaced as needed.

4.4.1. Manipulation

We define two different ways of manipulating the state of the model: *physical* and *artificial*.

A physical manipulation is indirect, the manipulator needs to create entities in the model, and then use those to perform the manipulation. An example could be that the manipulator creates a tool as a configuration of elements and connections, and then applies that tool to some body in the environment to perform a deformation.

An artificial manipulation is direct, the manipulator changes the values in the state directly, thus bypassing the physical laws of the model. For instance, to control the previously discussed tool, a predefined path could provide values that are given the positions of the elements as time progresses.

4.4.2. Human interaction

With human interaction, we mean a real-time interaction which feels natural to a user, as if the user is interacting with something in the real environment. This can be supported to varying degrees, we will try to describe the relevant components.

At the most basic level, the user is a body that we want to represent in the model. The information that flows from the user to the model is position information of the body. The information which the model can provide back is position information of the bodies in the environment, but also force information where the user's body is interacting with bodies in the model.

This interaction has to be handled by actual physical devices. Position and orientation information can be sampled in many ways, through electromagnetic sensors or mechanical arms for example [3,4]. In our testing, we have used a simple mechanical arm (see Fig. 5) which can sample position and orientation of a rod. Visual feedback can be produced by a computer monitor, also in stereoscopic form, and with a computer graphics visualization of the state of the model. There exists devices for force feedback, or haptic feedback, but such devices are still not as established as the previous types. We have not had the opportunity to perform testing with such devices—this will be a future field to explore.

Interfacing the model with a haptic device should be fairly straightforward. We represent the haptic device as a body, whose position and orientation is provided by the device. When the body interacts with other bodies in the environment, there will be forces acting on the 'device body'. These forces can then be displayed by the haptic device, and will thus be translated into the real world, where they will accelerate the 'device body' in reality, and thus change its position. This creates a loop of feedback. As mentioned, we have unfortunately been unable to test this.

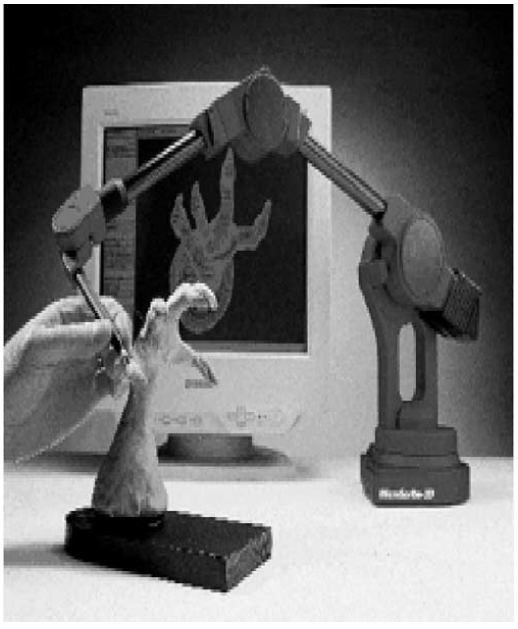


Fig. 5. We have used a MicroScribe 3D device to track position and orientation in real time.

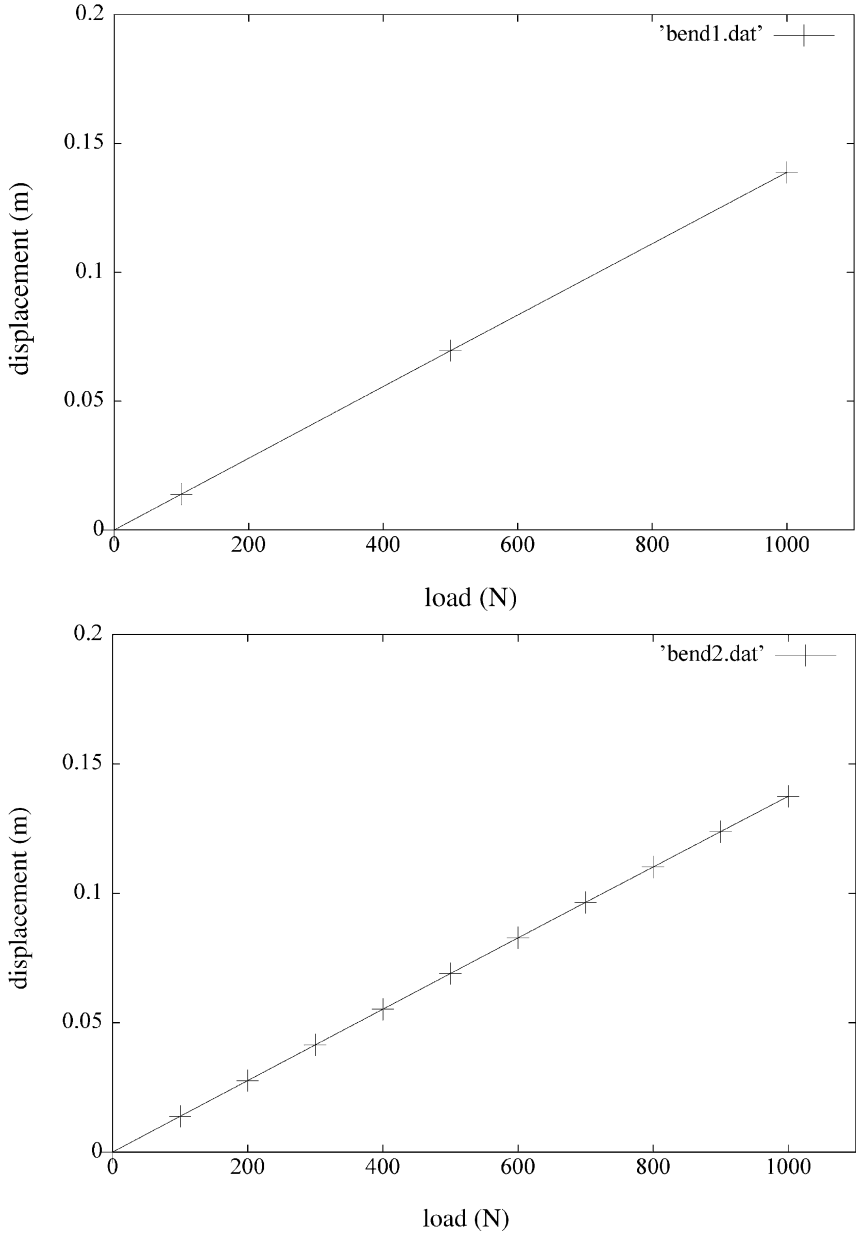


Fig. 6. Graphs describing the relation between the perpendicular loading of a beam and its displacement. The beam in the top graph is of twice the resolution than the bottom.

5. Preliminary verification

While we have a reasonably sound theoretical foundation, it is important to verify the model experimentally. We have performed some limited experiments that at least can give us an indication.

The experiment set up is a beam (see Fig. 7, right image), fixed at one end, with a static perpendicular load applied at the free end. The beam is $2 \times 0.7 \times 0.5$ m, the density is $0.58E3$ kg/m^3 and the elasticity modulus is $12.1E5$ N/m^2.

There is no gravity. The simulation is run until there is equilibrium (within a threshold), and the displacement of a point at the free end is measured. According to beam theory in mechanical engineering [26], the displacement grows linearly with the applied load, for small displacements. Thus, we apply a range of loads and analyze whether the relation is in fact linear. For now, we do not examine whether the magnitude of the displacement for a given load and material corresponds with beam theory, we are only interested in the relation.

In the graph at the bottom of Fig. 6, we have plotted the results of the experiment,. We can directly see the relation is linear.

We have also examined the impact of resolution on this particular experiment. We take the same beam, but double the resolution (see Fig. 7, left image), and examine the result. Before we can do that, we must use the same material in both beams, or the result will be meaningless. Ganovelli et al. [10] present a relation between the Hooke constant, Young's modulus, the spring length, and the volume of the

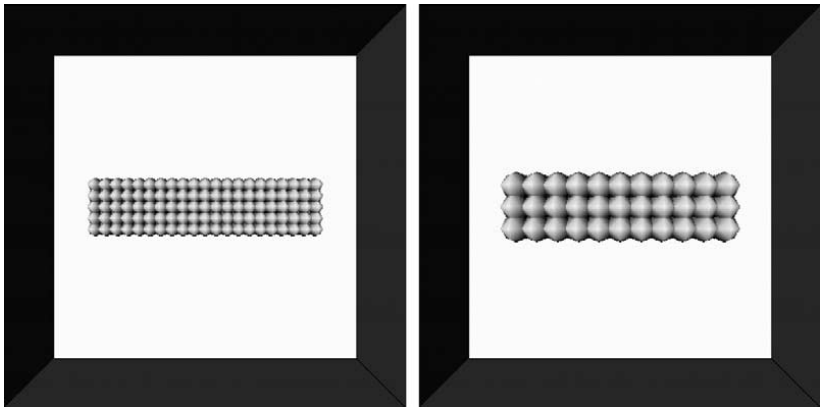


Fig. 7. A beam in two different resolutions.

tetrahedron which the spring mesh forms:

$$k_i = \frac{EV(\tau)}{l^2} \tag{21}$$

where τ is the tetrahedron considered, E is the Young's modulus, V is the volume, k_i is the Hooke's constant contributed by tetrahedron, and l is the spring length.

However, this relation has not provided consistent results for us. This will have to be examined more deeply in the future. Since at this point, we are only interested in the relation between two beams, we use a similar relation, which is still consistent with unit analysis:

$$k = El \tag{22}$$

where E is the Young's modulus, k is the Hooke's constant and l is the spring length.

We used this relation for the previous experiment, and now we perform the same experiment, but with twice the resolution. In the graph at the top of Fig. 6, we have plotted the results of the experiment. We can again directly see the relation is linear, and also that the graph is very nearly identical to the bottom graph. This means the behavior of the beam in this experiment is resolution-independent. As before, the absolute numbers are not important for this particular aspect to be proven. Due to the discretization process, the two resolutions do have differences, in mass and geometrical extent for example, but these differences are minor.

6. Applications

We now have a quite general model. The aim is now to determine how to apply this model, and in which applications it can be advantageous to use this model, compared to existing methods.

We can divide possible applications into two fields: *interactive* applications and *offline* applications. The main difference is that there exists a much tighter time constraint on interactive applications.

6.1. Geometric modeling

Geometric modeling is a typical interactive application. During shape design, geometric modeling is used both as a creative tool (sketching, claying), as well as for the final description of the shape. Normally, these two processes are not integrated. We will describe a geometric modeling application that integrates the two processes.

Geometric modeling can be done through a physical interface. We define a number of bodies as manipulators, which the user controls to manipulate other bodies in the environment. This provides a completely natural interface, and requires no knowledge of the model. However, due to limitations of the model, such a system is not yet fully practical. For example, since the model only supports elasticity and fracture, and no plasticity, it is not possible to perform a permanent non-fracturing deformation on a body. Therefore, we need to introduce a number of artificial operations, which can provide a replacement for full plasticity, as well as other properties.

We also have to make the distinction between deformation and creation/annihilation. The model supports deformation but not creation/annihilation of bodies or of elements of bodies. However, this could be resolved in a similar manner. We could simply couple the model with a system able to create/annihilate geometry, and then perform a translation procedure between the two systems. In our tests, we have used standard and custom geometric modeling packages to create basic shapes, which are then translated into a physical representation, and imported into the environment. We have not attempted annihilation as of yet.

6.1.1. Modeling operations

6.1.1.1. Physical deformation. The main operation is simply interacting with the physical model through contact forces, generated by the interaction between the manipulators and the bodies in the environment (see Fig. 8). With this operation, we can also perform fracturing, though such an operation might be implemented more efficiently as an artificial operation. Depending on how we interpret the state of

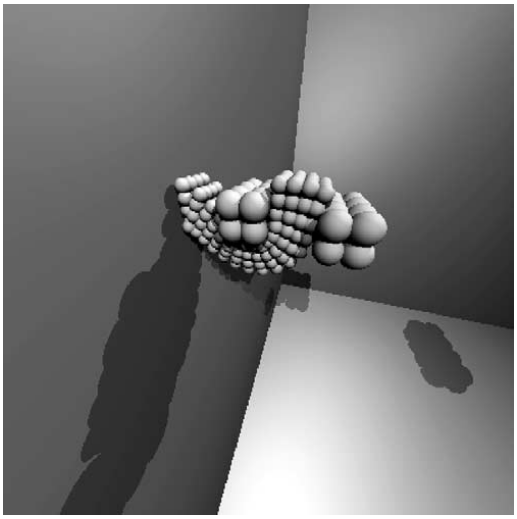


Fig. 8. The manipulators are applied for a bending operation.

the model, we could also perform topology changes with this operation (aside from fracturing, which can directly perform topology changes).

6.1.1.2. Renormalization (artificial plasticity). Since we have no plasticity, deformations are essentially useless for prolonged modeling since they are not permanent. We can alleviate this by introducing an artificial operation called 'renormalization'. When this operation is performed on a certain body, all connections in the body assume the current distance as the nominal distance. Macroscopically, this means the body takes on the deformed shape as the rest shape.

6.1.1.3. Direct attribute modification. A generalization of renormalization is 'direct attribute modification'. Given that we can select a part of a body, or a body, and determine which elements comprise the selection, we can artificially modify the attributes of the elements or of the connections between the elements. We could, for example, make part of a body stiffer, so that deformation of the entire body has less impact on that specific part. Another possibility could be increasing the fracture distance of elements of parts of a body, thus making it more 'sticky'. This could be used to glue parts of bodies together, or, applied to a manipulator, could increase flexibility of manipulation. However, the only such operation which has been implemented and tested is this 'renormalization'. We have also tested the 'glue on manipulator' concept, but only as part of defining the actual manipulator.

6.1.2. Deformation mapping

While such a geometric description is enough, when a b-rep (boundary representation) description already exists, it can be useful to use that representation directly, and then map the deformation of the physical representation to the b-rep. We have previously described how we can generate a physical representation from a b-rep. If we store the b-rep with the physical representation, we can use the b-rep for visualization and post processing, while using the physical representation for simulation (Fig. 9).

There exists methods which can map deformation of a 'cage' to a b-rep shape inside the cage [28]. We can apply a similar method, but locally for each vertex of the b-rep, and let the cage be defined by neighboring elements. Normally, interpolation is used to determine the deformation inside the cage, however, since our cages are very local to the vertices, we do not use interpolation (Fig. 10).

We denote the space where the elements are expressed as 'simulation space'. For each vertex in the b-rep, we find four close elements that we can generate an orthogonal base from, using the Gram–Schmidt method, for example. One element forms origo, one element forms the primary axis, while the other two are used to determine the orientations of the two secondary axes. We then transform the vertex into this new base, and store this representation. If we now transform the vertex back into simulation space, we will get back the original vertex. However, if the simulation has led to a deformation of the original positions of the elements, the vertex will also be deformed according to the deformation of the space determined by the elements.

We can view this using a Voronoi diagram formulation. If we generate the Voronoi diagram of the elements, we will end up with cells, where each vertex of the b-rep exists in a cell. If the elements are deformed, the cells are also deformed. This way, we can determine the discrete space deformation from the deformation of the elements. However, this model is only valid for one specific Voronoi diagram, so deformations that give rise to new diagrams may produce erroneous results. Depending on the tolerance of the application, such errors may or may not be acceptable. In our testing, such errors have been acceptable when only visualization is required.

6.1.3. Geometric modeling example

See Fig. 11, combined with the previous Fig. 8, for illustrations of how we can perform operations using a natural interface to produce arbitrary deformations.

6.2. Analysis

As mentioned in Section 1, it can be useful to perform simple analysis in the conceptual stage to determine fruitful design directions. It is obvious that if the later stage analysis could be performed at the conceptual stage, it would be done. However, this is not practical. What we have instead is a cheaper variant, which produces less accurate results. However, it is better to have some results which can be indicative, and later can be verified more thoroughly, than to have none at all, and perhaps have to go through a redesign after layer analysis.

6.2.1. Analysis example

Our example considers the analysis of a design of a

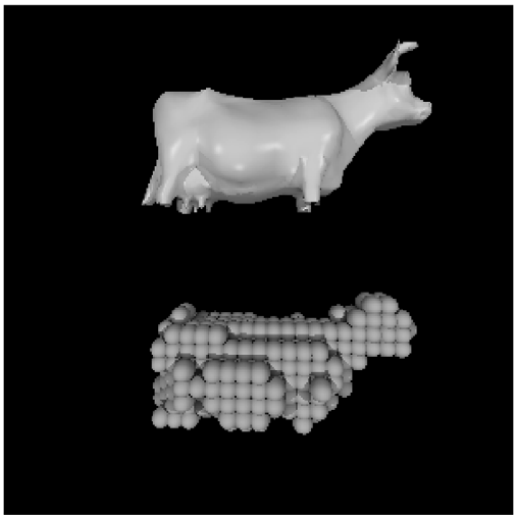


Fig. 9. A b-rep of a cow shape is translated to a physical representation.

support part. We fix two supports by the back surfaces, and then let them support a thick beam. We then drop a heavy cylinder on the beam, and examine the behavior of one of the supports during impact. The supports and beam are semi-rigid, perhaps comparable to a wood material, or a polymer.

Fig. 12 shows the system during the simulation. Fig. 13 shows a close-up of only one of the supports during simulation, with everything else in the system removed from the visualization. Since we cannot yet map materials from the real world into the model, such analysis is not yet practically usable for most cases. However, if the material properties can be made to match reasonably, through empirical testing for example, it can be used to compare behaviors of different designs. The support, beam and cylinder were modeled using a traditional solid modeling system (Rhinoceros). The translation of the geometry required an insignificant time, however, some manual input had to be made (element sizes and material parameters). The total computation time for the simulation in the example was less than 600 s. The hardware used was a dual processor (Intel Celeron, 450 MHz) PC-AT system.

6.3. Integration with existing methods

After conceptual shape modeling and analysis has been performed, and some possible designs have been produced, we need to transfer this information to detail design systems, and perform more accurate analysis. However, this transfer is theoretically trivial, since we at least implicitly have a discrete boundary representation of the geometry of the bodies, and a discrete representation of the physical quantities of the bodies.

More difficult, and interesting, is direct integration with existing methods. For example, we might have parts of the design which are at the conceptual phase, and parts which are already detailed. An important future aim could be to be able to perform simulation with both a traditional Finite Element Analysis model and a model such as this directly

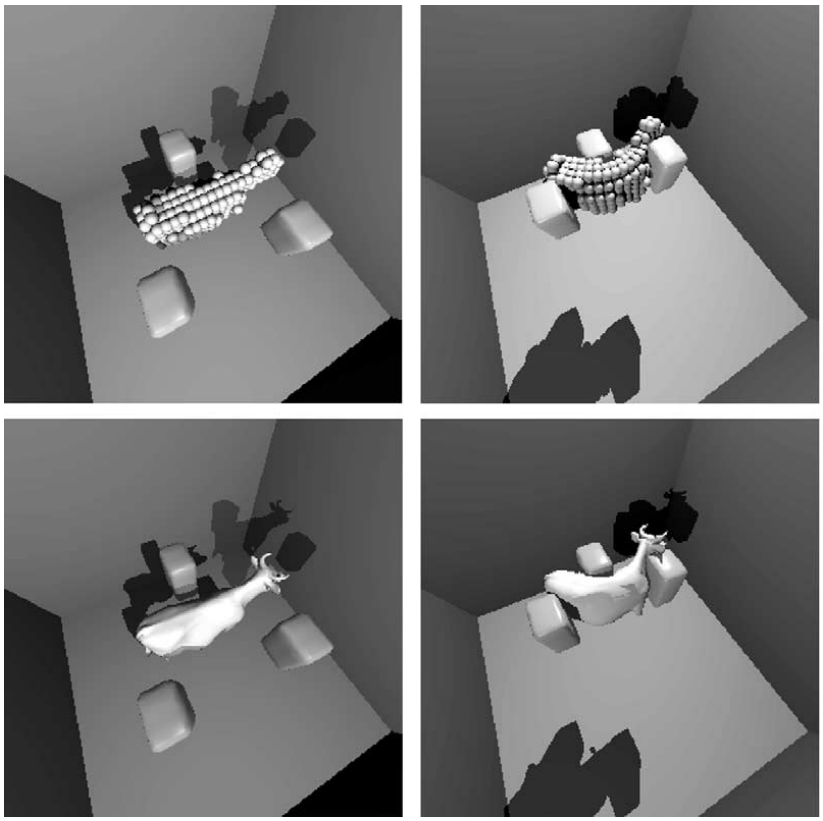


Fig. 10. The deformation of the physical representation can be mapped onto the b-rep.

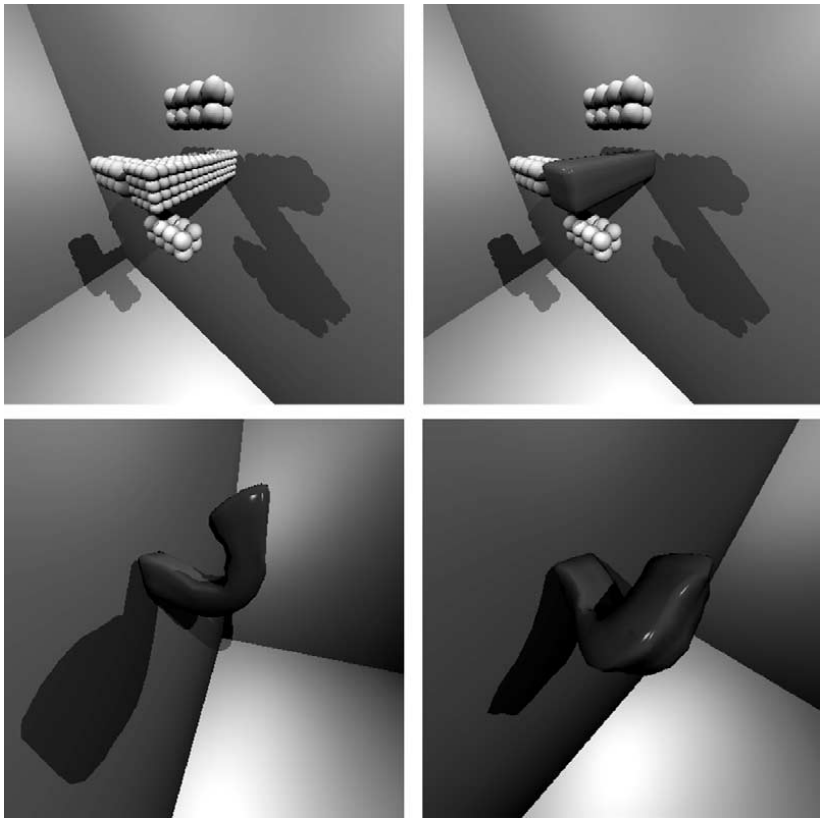


Fig. 11. We start with a bar, which we want to perform bending operations on. By applying tools, we can perform bending operations on the bar. We are, however, limited by the simplicity of our manipulators, and the interface. We use our artificial plasticity to first bend the bar upwards, then 'freeze' that shape as the nominal shape. We then bend the deformed bar sideways.

interfaced, so any two bodies from different representations could interact.

The presented model is meant to be used in conceptual design support tools. This means that it has no direct influence on the structure of the design process, it only enhances certain stages. Any design process normally consists of first creating a concept according to specifications, and then analyzing whether it actually meets those specifications.

This model can be used at the concept shape creation stage, as a kind of 'virtual claying', or more natural geometry manipulation than typical direct geometrical entity manipulation. At the analysis stage, it can be used to give an indication of feasibility, or a rough estimate of the behavior. An analysis example could be a concept for a new type of washing machine. This type of simulation could then indicate how much angular velocity of the drum is needed to create the desired friction or movement of the cloth, and also what vibrations and deflections the drum causes on the structure of the machine.

7. Conclusions and future research

We have described the theory and implementation of a discrete mechanics model for deformable bodies. The model attempts to preserve physical principles by starting at the atomic level, and then recursively approximating groups of basic elements into fewer larger elements. We have practically demonstrated that the model incorporates behaviors such as motion, collision and deformation, and theoretically shown behaviors such as fracture and fusing. We have presented two main applications in the conceptual design domain for this model: interactive shape modeling/geometric modeling (virtual claying) and rapid analysis. To support the claim that the model is suited for rapid evaluation, we have presented an algorithm analysis, and shown that the most expensive algorithm is collision-detection, and that algorithms exist which are provably $O(n\log n + m)$, where n is the number of shapes considered, and m the number of shape pairs which are 'very close'. Fundamentally, our system can be represented as a mass-spring system, and thus shares many of the weaknesses and strengths of such a system. For example, bodies made of very rigid materials require a very fine time discretization, with long computation time as a result.

Possible future research directions are:

1. Plasticity. Currently, we incorporate elasticity and viscosity as material phenomena. While we have an artificial model of plasticity, a physics based plasticity model could have large benefits. There has been work done in this area that may be directly applicable to this model [31].
2. Experimental verification. Before the model can be

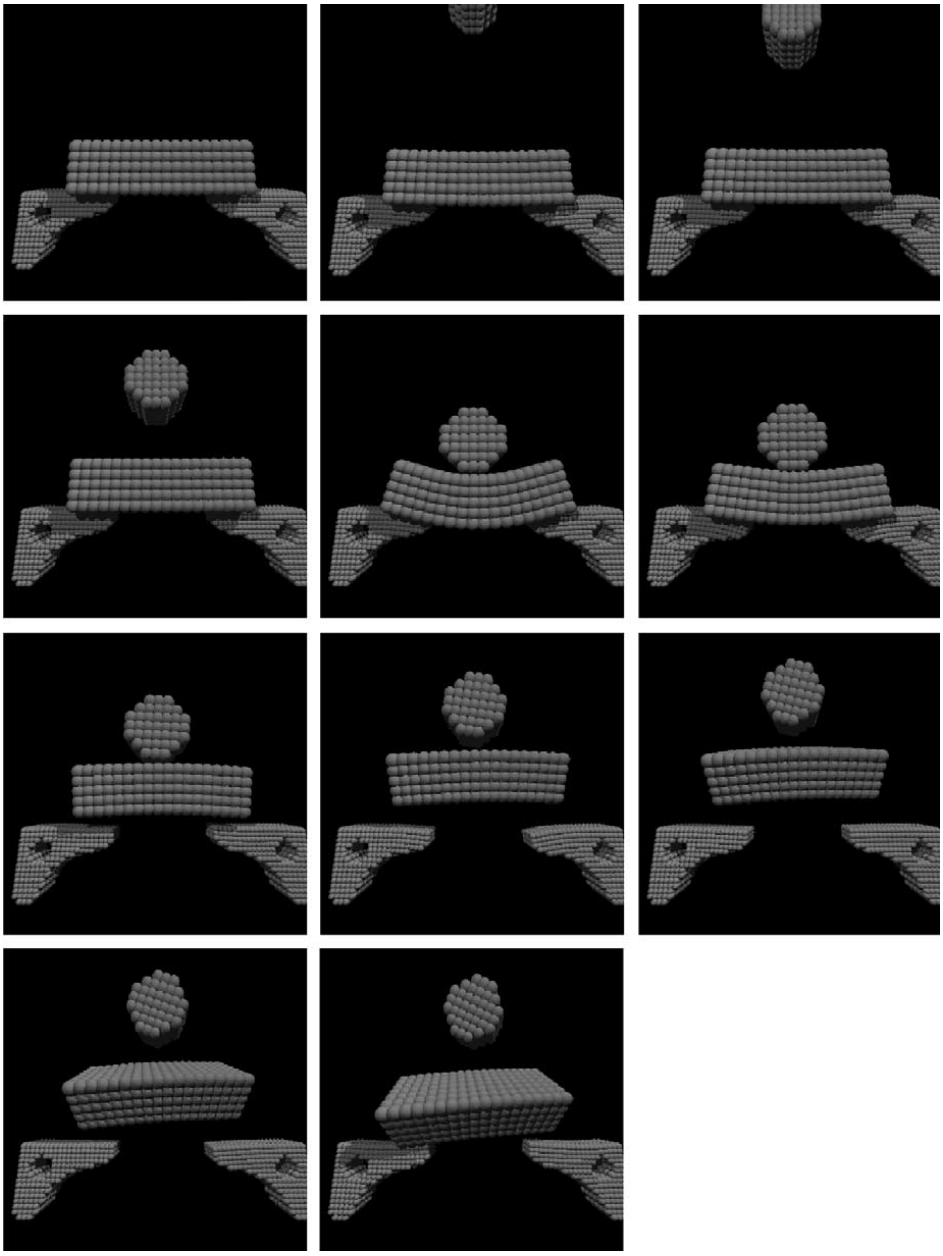


Fig. 12. The beam and supports are initially at rest, while the cylinder is falling rapidly. The cylinder impacts on the beam, and the beam and supports flex quite significantly. The supports and beam flex back, and launch the cylinder upwards again, the beam is also launched. The total computation time for this simulation was less than 600 s.

practically used, we need to perform experimental verification of the modeled phenomena.

3. Material mapping. Related to experimental verification, we need to be able to map real-world materials to parameters in the model, so we can translate a description of a real-world system of bodies into the model.
4. Dynamic resolution change. Our theory is based on a recursive resolution reduction. If we can show that a given collection of low-resolution elements sufficiently approximates a given collection of higher-resolution elements, we can dynamically replace the two representations at will. While this is true in our theory, we have not shown how it practically can be done, for example, how the parameters of elements are dependent on resolution.
5. Interfacing with rigid-body representations. If we can represent bodies using the standard rigid-body formulation, we can at least partly overcome the computational expense of simulating rigid materials. This should be fairly straightforward, but requires testing, and perhaps examination of additional possibilities of such a hybrid model.

Acknowledgements

This research has been performed as part of the Integrated

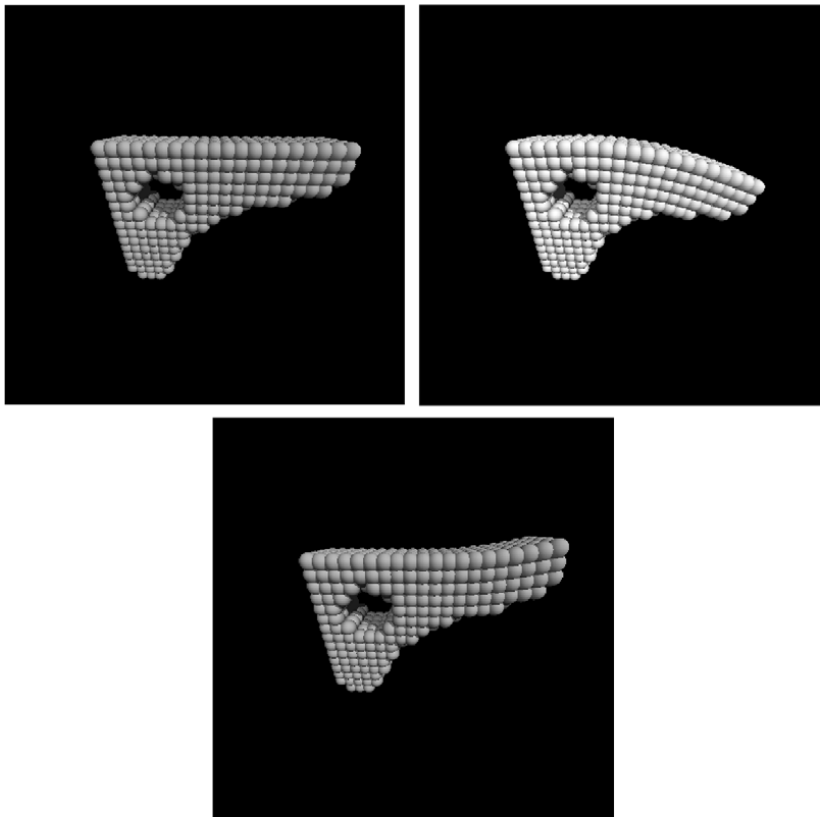


Fig. 13. A close-up of one of the supports. In the first image the support is at rest. In the second image the impact is at its extreme. In the third image the support has flexed back. Aside from a large deflection, we see no anomalies in the support.

Concept Advancement (ICA) project, at the Delft University of Technology.

References

[1] Andersson K, Sellgren U. Modeling and simulation of physical behavior of complex products. Proceedings of Produktmodeller '98, Linköping, 10 November, 1998.
[2] Baraff D, Witkin A. Dynamic simulation of non-penetrating flexible bodies. Computer Graphics (Proc. SIGGRAPH), vol. 26. 1992. p. 303–8.
[3] Berkley J, Weghorts S, Gladstone H. Real-time finite element modeling with haptic support. In: Proceedings of the ASME Design Engineering Technical Conferences, 1999.
[4] Bullinger H, Breining R, Bauer W. Virtual prototyping—state of the art in product design. Proceedings of the Twenty-Sixth International Conference on Computers and Industrial Engineering, Melbourne, 1999. p. 103–7.
[5] Chen et al. Physically-based animation of volumetric objects. Proceeding of IEEE Computer Animation '98, 1998, p. 154–60.
[6] Cohen J, Lin M, Manocha D, Ponamgi M. I-COLLIDE: an interactive and exact collision detection system for large-scale environments. Proceedings of ACM Interactive 3D Graphics Conference, 1995. p. 189–96.
[8] Deisinger J, Blach R, Wesche G, Breining R, Simon A. Towards immersive modeling—challenges and recommendations: a workshop analyzing the needs of designers. In: Proceedings of Sixth Eurographics Workshop on Virtual Environments, 2000.
[10] Ganovelli F, Cignoni P, Montani C, Scopigno R. A multiresolution model for soft objects supporting interactive cuts and lacerations. Eurographics 2000;19(3).
[13] Heath M. Scientific computing. McGraw-Hill, 1997.
[17] ICA group web site. http://www.io.tudelft.nl/research/ica/.
[18] James D, Pai D. Accurate real time deformable objects. SIGGRAPH 99, 1999.
[20] Jansson J, Horváth I, Vergeest JSM. Implementation and analysis of a mechanics simulation module for use inn a conceptual design system. In: Proceedings ASME Design Engineering Technical Conferences, 2000.
[21] Kang H, Kak A. Deforming virtual objects interactively in accordance with an elastic model. Computer-Aided Design 1996.
[23] Kleppner D, Kolenkow R. An introduction to mechanics, McGraw-Hill, 1978. p. 91.
[24] Pedersen P. Elasticity—anisotropy—laminates, 2000. http://www.fam.dtu.dk/html/pp.html.
[25] Ping G, Nanxin W. DOE study in building surrogate models for complex systems. In: Proceedings ASME Design Engineering Technical Conferences, 2000.
[26] Popov E. Engineering mechanics of solids. Prentice-Hall, 1999.
[27] Provot X. Deformation constraints in a mass-spring model to describe rigid cloth behavior. Proceedings Graphics Interface '95 1995:147–54.
[28] Sederberg T, Parry S. Free-form deformation of solid primitives. Computer Graphics 1986:151–60.
[29] Szeliski R, Tonnesen D. Surface modeling with oriented particle systems. Computer Graphics 1992;26(2).
[30] Terzopoulos D, Platt J, Barr A, Fleischer K. Elastically deformable models. SIGGRAPH '87, 1987, p. 205–14.
[31] Terzopoulos D, Fleischer K. Modeling inelastic deformation: viscoelasticity, plasticity, fracture. SIGGRAPH '88, 1988, p. 269–78.
[32] Wiegers T, Horváth I, Vergeest JSM, Opiyo EZ, Kuczogi G. Requirements for highly interactive system interfaces to support conceptual design. CIRP99, 1999.

Further reading

Cormen H, Leiserson C, Rivest R. Introduction to algorithms. MIT Press, 1998.

Fuchs H, Kedem ZM, Naylor BF. On visible surface generation by a priori tree structures. SIGGRAPH 80, 1980, p. 124–33.

Goldstein H. Classical mechanics. Reading, MA: Addison-Wesley, 1950.

Grandin Jr H. Fundamentals of the finite element methods. Macmillan, 1986.

Hollerbach JM. Cohen E, Thompson W, Freier R, Johnson D, Nahvi A, Nelson D, Thompson TV. Haptic interfacing for virtual prototyping of mechanical CAD designs. In: Proceedings of the ASME Design Engineering Technical Conferences, 1997.

Horváth I, Kuczogi G, Staub G. Spatial behavioural simulation of mechanical objects. Proceedings of TMCE '98 1998:221–3.

Hunter PJ, Pullan AJ. FEM/BEM notes, 1997. http://www.esc.auckland.ac.nz/Academic/Texts/FEM-BEM-notes.html.

Jansson J, Vergeest JSM. A general mechanics model for systems of deformable solids. In: Proceedings International Symposium on Tools and Methods for Concurrent Engineering, 2000.

Keller H, Stolz H, Ziegler A, Bräunl T, Virtual mechanics—simulations and animation of rigid body systems. Computer Science Report No. 8/93, 1993. http://www.ee.uwa.edu.au/~braunl/aero/ftp/docu.english.ps.gz.

Algorithms and Data Structures for Multi-Adaptive Time-Stepping

Johan Jansson
Chalmers University of Technology
and
Anders Logg
Simula Research Laboratory

Multi-adaptive Galerkin methods are extensions of the standard continuous and discontinuous Galerkin methods for the numerical solution of initial value problems for ordinary or partial differential equations. In particular, the multi-adaptive methods allow individual and adaptive time steps to be used for different components or in different regions of space. We present algorithms for efficient multi-adaptive time-stepping, including the recursive construction of time slabs, adaptive time step selection and automatic generation of the dual problem. We also present data structures for efficient storage and interpolation of the multi-adaptive solution. The efficiency of the proposed algorithms and data structures is demonstrated for a series of benchmark problems.

Categories and Subject Descriptors: G.1.7 [**Ordinary Differential Equations**]: —*Error analysis, Initial value problems*; G.1.8 [**Partial Differential Equations**]: —*Finite Element Methods*; G.4 [**Mathematical Software**]: —*Algorithm design and analysis, Efficiency*

General Terms: Algorithms, Performance

Additional Key Words and Phrases: Multi-adaptivity, individual time steps, local time steps, ODE, continuous Galerkin, discontinuous Galerkin, mcgq, mdgq, C++, implementation, algorithms

1. INTRODUCTION

We have earlier in a sequence of papers [35; 36; 38] introduced the multi-adaptive Galerkin methods mcG(q) and mdG(q) for the approximate (numerical) solution of ODEs of the form

$$\begin{aligned} \dot{u}(t) &= f(u(t), t), \quad t \in (0, T], \\ u(0) &= u_0, \end{aligned} \tag{1}$$

where $u : [0, T] \to \mathbb{R}^N$ is the solution to be computed, $u_0 \in \mathbb{R}^N$ a given initial value, $T > 0$ a given final time, and $f : \mathbb{R}^N \times (0, T] \to \mathbb{R}^N$ a given function that is

Johan Jansson, Department of Computational Technology, Chalmers University of Technology, SE–412 96 Göteborg, Sweden. *Email*: johanjan@math.chalmers.se.
Anders Logg, Simula Research Laboratory, P.O. Box 134 NO-1325 Lysaker, Norway. *Email:* logg@simula.no.

Lipschitz-continuous in u and bounded.

The multi-adaptive Galerkin method mcG(q) and mdG(q) extend the standard mono-adaptive continuous and discontinuous Galerkin methods cG(q) and dG(q), studied earlier in detail in [29; 28; 30; 4; 13; 31; 8; 9; 7; 10; 11; 12; 6; 15; 16; 17; 19; 18], by allowing individual time step sequences $k_i = k_i(t)$ for the different components $U_i = U_i(t)$, $i = 1, 2, \ldots, N$, of the approximate solution $U \approx u$ of the initial value problem (1). For related work on local time-stepping, see also [26; 27; 39; 2; 1; 40; 20; 3; 34; 41].

In the current paper, we discuss important aspects of the implementation of multi-adaptive Galerkin methods. While earlier results on multi-adaptive time-stepping presented in [35; 36; 38] include the basic formulation of the methods, a priori and a posteriori error estimates, together with a proof-of-concept implementation and results for a number of model problems, the current paper addresses the important issue of efficiently implementing the multi-adaptive methods with minimal overhead as compared to standard mono-adaptive solvers. For many problems, in particular when the propagation of the solution is local in space and time, the potential speedup of multi-adaptivity is large, but the actual speedup may be far from the ideal speedup if the overhead of the more complex implementation is significant.

1.1 Implementation

The algorithms presented in this paper are implemented by the multi-adaptive ODE-solver available in DOLFIN [24; 25], the C++ interface of the new open-source software project FEniCS [23; 5] for the automation of Computational Mathematical Modeling (CMM). The multi-adaptive solver in DOLFIN is based on the original implementation Tanganyika, presented in [36], but has been completely rewritten for DOLFIN.

The multi-adaptive solver is actively developed by the authors, with the intention of providing the next standard for the solution of initial value problems. This will be made possible through the combination of an efficient forward integrator, automatic and reliable error control, full integration with the automatic discretization of PDEs through FFC [37; 32; 33] and a simple and intuitive user interface.

1.2 Obtaining the software

DOLFIN is licensed under the GNU General Public License [21], which means that anyone is free to use or modify the software, provided these rights are preserved. The complete source code of DOLFIN, including numerous example programs, is available at the DOLFIN web page [24].

1.3 Notation

The following notation is used throughout this paper: Each component $U_i(t)$, $i = 1, \ldots, N$, of the approximate m(c/d)G(q) solution $U(t)$ of (1) is a piecewise polynomial on a partition of $(0, T]$ into M_i sub intervals. Sub interval j for component i is denoted by $I_{ij} = (t_{i,j-1}, t_{ij}]$, and the length of the sub interval is given by the local *time step* $k_{ij} = t_{ij} - t_{i,j-1}$. We shall sometimes refer to I_{ij} as an *element*. This is illustrated in Figure 1. On each sub interval I_{ij}, $U_i|_{I_{ij}}$ is a polynomial of degree q_{ij}.

Furthermore, we shall assume that the interval $(0, T]$ is partitioned into blocks between certain synchronized time levels $0 = T_0 < T_1 < \ldots < T_M = T$. We refer to the set of intervals $\mathcal{T}_n$ between two synchronized time levels T_{n-1} and T_n as a *time slab*:

$$\mathcal{T}_n = \{I_{ij} : T_{n-1} \leq t_{i,j-1} < t_{ij} \leq T_n\}.$$

We denote the length of a time slab by $K_n = T_n - T_{n-1}$.

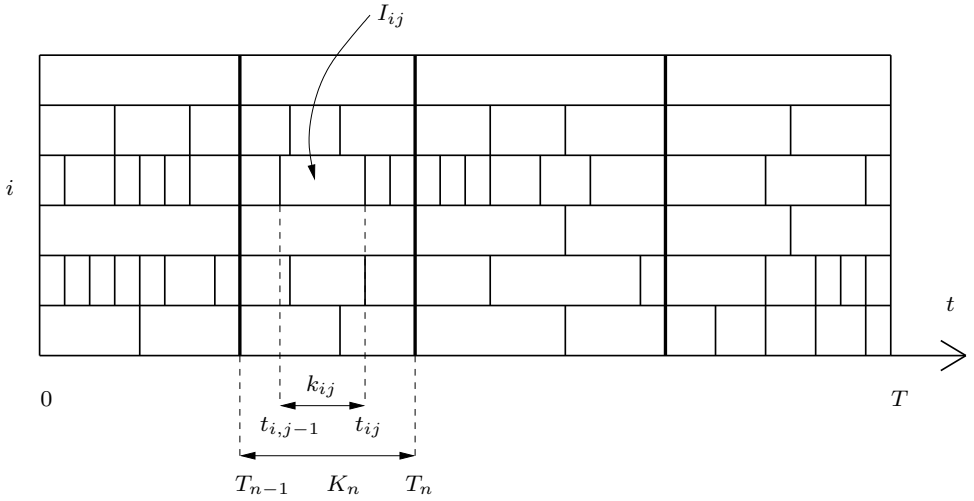


Fig. 1. Individual partitions of the interval $(0, T]$ for different components. Elements between common synchronized time levels are organized in time slabs. In this example, we have $N = 6$ and $M = 4$.

1.4 Outline of the paper

We first give an introduction to multi-adaptive time-stepping in Section 2. We then present the key algorithms used by the multi-adaptive ODE solver of DOLFIN in Section 3, followed by a discussion of data structures for efficient representation and interpolation of multi-adaptive solutions in Section 4. In Section 5, we then present a number of numerical examples, including benchmark problems that demonstrate the efficiency of the proposed algorithms and data structures.

2. MULTI-ADAPTIVE TIME-STEPPING

In this section, we give a quick introduction to multi-adaptive time-stepping, including the formulation of the methods, error estimates and adaptivity. For a more detailed account, we refer the reader to [35; 36; 38].

2.1 Formulation of the methods

Just as the standard cG(q) and dG(q) methods, the multi-adaptive mcG(q) and mdG(q) methods are obtained by multiplying the system of equations (1) with a suitable test function v, to obtain a variational problem of the form: Find $U \in V$

with $U(0) = u_0$, such that

$$\int_0^T (v, \dot{U}) \, \mathrm{d}t = \int_0^T (v, f(U, \cdot)) \, \mathrm{d}t \quad \forall v \in \hat{V}, \tag{2}$$

where $(\cdot, \cdot)$ denotes the standard inner product in $\mathbb{R}^N$ and $(\hat{V}, V)$ is a suitable pair of discrete functions spaces, the *test* and *trial* spaces respectively.

For the standard cG(q) method, the trial space V consists of the space of continuous piecewise polynomial vector-valued functions of degree $q = q(t)$ on a partition $0 = t_0 < t_1 < \cdots < t_M = T$ and the test space $\hat{V}$ consist of the space of (possibly discontinuous) piecewise polynomial vector-valued functions of degree $q - 1$ on the same partition. The multi-adaptive mcG(q) method extends the standard cG(q) method by extending the test and trial spaces to piecewise polynomial spaces on individual partitions of the time interval according to Figure 1. Thus, each component $U_i = U_i(t)$ is continuous and piecewise polynomial on the individual partition $0 = t_{i0} < t_{i1} < \cdots < t_{iM_i} = T$ for $i = 1, 2, \ldots, N$.

For the standard dG(q) method, the test and trial spaces are equal and consist of the space of (possibly discontinuous) piecewise polynomial vector-valued functions of degree $q = q(t)$ on a partition $0 = t_0 < t_1 < \cdots < t_M = T$, which extends naturally to the multi-adaptive mdG(q) method by allowing each component of the test and trial functions to be piecewise polynomial on individual partitions of the time interval as above. Note that for both the dG(q) method and the mdG(q) method, the integral $\int_{0,T}(v, \dot{U}) \, \mathrm{d}t$ in (2) must be treated appropriately at the points of discontinuity, see [35].

Both in the case of the mcG(q) and mdG(q) methods, the variational problem (2) gives rise to a system of discrete equations by expanding the solution U in a suitable basis on each local interval I_{ij},

$$U_i|_{I_{ij}} = \sum_{m=0}^{q_{ij}} \xi_{ijm} \phi_{ijm}, \tag{3}$$

where $\{\xi_{ijm}\}_{m=0}^{q_{ij}}$ are the *degrees of freedom* for U_i on I_{ij} and $\{\phi_{ijm}\}_{m=0}^{q_{ij}}$ is a suitable basis for $P^{q_{ij}}(I_{ij})$. For any particular choice of quadrature, the resulting system of discrete equations takes the form of an implicit Runge-Kutta method on each local interval I_{ij}. In the case of the mcG(q) method, the discrete equations are given by

$$\xi_{ijm} = \xi_{ij0} + k_{ij} \sum_{n=0}^{q_{ij}} w_{mn}^{[q_{ij}]} \; f_i(U(\tau_{ij}^{-1}(s_n^{[q_{ij}]})), \tau_{ij}^{-1}(s_n^{[q_{ij}]})), \tag{4}$$

for $m = 1, \ldots, q_{ij}$, where $\{w_{mn}^{[q_{ij}]}\}_{m=1,n=0}^{q_{ij}}$ are weights, τ_{ij} maps I_{ij} to $(0, 1]$: $\tau_{ij}(t) = (t - t_{i,j-1})/(t_{ij} - t_{i,j-1})$, and $\{s_n^{[q_{ij}]}\}_{n=0}^{q_{ij}}$ are quadrature points defined on $[0, 1]$. The discrete equations for the mdG(q) method are similar in structure. See Section 3.5 below for a discussion of suitable quadrature rules and basis functions.

2.2 Error estimates and adaptivity

In [35], a posteriori error estimates are proved for the multi-adaptive mcG(q) and mdG(q) methods. For the mcG(q) method, the estimate takes the form

$$|M(e)| \leq E \equiv \sum_{i=1}^{N} S_i^{[q_i]}(T) \max_{[0,T]} \{C_i k_i^{q_i} |R_i|\}\,, \tag{5}$$

with a similar estimate for the mdG(q) method. Here, $M : \mathbb{R}^N \to \mathbb{R}$ denotes some given functional of the global error $e = U - u$ to be estimated, $R = \dot{U} - f(U, \cdot)$ denotes the *residual* of the computed solution, C denotes a set of interpolation constants (which may be different for each local interval) and $S_i(T)$ denotes a *stability factor* that measures the rate of propagation of errors for component U_i (the influence of errors in component U_i on the size of the error in the given functional). Comparing to standard Runge-Kutta methods for the solution of initial value problems, the stability factor is the missing link between the "local error" and the global error. Note that alternatively, the stability information may be kept as a local time-dependent *stability weight* for more fine-grained control of the contributions to the global error. The stability factors are obtained from solving the *dual problem* of (1) for the given functional M, see [6; 35].

The individual time steps may then be chosen so as to equidistribute the error onto the different components,

$$C_{ij} k_{ij}^{q_{ij}} \max_{I_{ij}} |R_i| = \mathrm{TOL}/(N S_i), \tag{6}$$

in an iterative fashion according to the following basic adaptive algorithm:

(i) Solve the primal problem with time steps based on (6);
(ii) Solve the dual problem and compute the stability factors;
(iii) Compute an error bound E based on (5);
(iv) If $E \leq \mathrm{TOL}$ then stop; if not go back to (i).

3. ALGORITHMS

We present below a collection of the key algorithms for multi-adaptive time-stepping. The algorithms are given in pseudo-code and where appropriate we give remarks on how the algorithms have been implemented in C++ for DOLFIN. In most cases, we present simplified versions of the algorithms with focus on the most essential steps.

3.1 General algorithm

The general multi-adaptive time-stepping algorithm is Algorithm 1. Starting at $t = 0$, the algorithm creates a sequence of time slabs until the given end time T is reached. The end time T is given as an argument to CreateTimeSlab, which creates a time slab covering an interval $[T_{n-1}, T_n]$ such that $T_n \leq T$. CreateTimeSlab returns the end time T_n of the created time slab and the integration continues until $T_n = T$. For each time slab, the system of discrete equations is solved iteratively, using direct fixed-point iteration or a preconditioned Newton's method, until the discrete equations given by the mcG(q) or mdG(q) method have converged.

Algorithm 1 $U = \text{Integrate(ODE)}$

$t \leftarrow 0$
while $t < T$
    {time slab, t} $\leftarrow$ CreateTimeSlab($\{1, \ldots, N\}$, t, T)
    SolveTimeSlab(time slab)
end while

The basic forward integrator, Algorithm 1, can be used as the main component of an adaptive algorithm with automated error control of the computed solution as outlined in Section 2. In each iteration, the *primal problem* (1) is solved using Algorithm 1. An ODE of the form (1) representing the *dual problem* is then created and solved using Algorithm 1. It is important to note that both the primal and the dual problems are solved using the same algorithm, but with different time steps and, possibly, different tolerances, methods, and orders. When the solution of the dual problem has been computed, the stability factors $\{S_i(T)\}_{i=1}^N$ and the error estimate can be computed.

3.2 Recursive construction of time slabs

In each step of Algorithm 1, a new time slab is created between two synchronized time levels T_{n-1} and T_n. The time slab is organized recursively as follows. The root time slab covering the interval $[T_{n-1}, T_n]$ contains a non-empty list of elements, which we refer to as an *element group*, and a possibly empty list of time slabs, which in turn may contain nested groups of elements and time slabs. Each such element group together with the corresponding nested set of element groups is referred to as a *sub slab*. This is illustrated in Figure 2.

To create a time slab, we first compute the desired time steps for all components as given by the a posteriori error estimate (5). We discuss in detail the time step selection below in Section 3.3. A threshold θK is then computed based on the maximum time step K and a fixed parameter $\theta \in (0, 1)$ controlling the density of the time slab. The components are partitioned into two sets based on the threshold, see Figure 3. For each component in the group with large time steps, an element is created and added to the element group of the time slab. The remaining components with small time steps are processed by a recursive application of this algorithm for the construction of time slabs.

We organize the recursive construction of time slabs as described by Algorithms 2, 3, 5, and 4. The recursive construction simplifies the implementation; each recursively nested sub slab can be considered as a sub system of the ODE. Note that the group of recursively nested sub slabs for components in group I_1 is created before the element group containing elements for components in group I_0. The tree of time slabs is thus created recursively *breadth-first*, which means in particular that the element for the component with the largest time step is created first.

Algorithm 3 for the partition of components can be implemented efficiently using the function `std::partition()`, which is part of the Standard C++ Library.

Algorithm 2 {time slab, T_n} = CreateTimeSlab(components, T_{n-1}, T)

$\{I_0,\, I_1,\, K\} \leftarrow$ Partition(components)
if $T_{n-1} + K < T$
    $T_n \leftarrow T_{n-1} + K$
else
    $T_n \leftarrow T$
end if
element group $\leftarrow$ CreateElements(I_1, T_{n-1}, T_n)
time slabs $\leftarrow$ CreateTimeSlabs(I_0, T_{n-1}, T_n)
time slab $\leftarrow$ {element group, time slabs}

Algorithm 3 $\{I_0,\, I_1,\, K\}$ = Partition(components)

$I_0 \leftarrow \emptyset$
$I_1 \leftarrow \emptyset$
$K \leftarrow$ *maximum time step within* components
for each component
    $k \leftarrow$ *time step of* component
    if $k < \theta K$
        $I_0 \leftarrow I_0 \cup$ {component}
    else
        $I_1 \leftarrow I_1 \cup$ {component}
    endif
end for
$\underline{K} \leftarrow$ *minimum time step within* I_1
$K \leftarrow \underline{K}$

Algorithm 4 elements = CreateElements(components, T_{n-1}, T_n)

elements $\leftarrow \emptyset$
for each component
    create element *for* component *on* $[T_{n-1}, T_n]$
    elements $\leftarrow$ elements $\cup$ element
end for

Algorithm 5 time slabs = CreateTimeSlabs(components, T_{n-1}, T_n)

time slabs $\leftarrow \emptyset$
$t \leftarrow T_{n-1}$
while $t < T$
    {time slab, t} $\leftarrow$ CreateTimeSlab(components, t, T_n)
    time slabs $\leftarrow$ time slabs $\cup$ time slab
end while

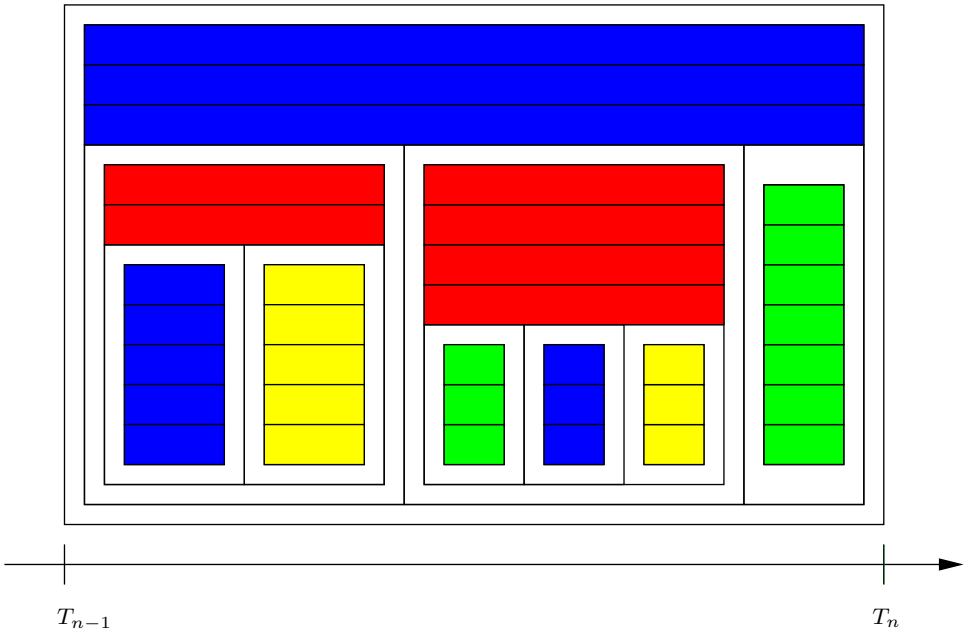


Fig. 2. The recursive organization of the time slab. Each time slab contains an element group and a list of recursively nested time slabs. The root time slab in the figure contains one element group of three elements and three sub slabs. The first of these sub slabs contains an element group of two elements and two nested sub slabs, and so on. The root time slab recursively contains a total of nine element groups and 35 elements.

3.3 Multi-adaptive time step selection

The individual and adaptive time steps k_{ij} are determined during the recursive construction of time slabs based on an a posteriori error estimate as discussed in Section 2. Thus, according to (6), each local time step k_{ij} should be chosen to satisfy

$$k_{ij} = \left(\frac{\mathrm{TOL}}{C_{ij} N S_i \max_{I_{ij}} |R_i|} \right)^{1/q_{ij}} . \tag{7}$$

where TOL is a given tolerance.

However, the time steps can not be based directly on (7), since that leads to unwanted oscillations in the size of the time steps. If $r_{i,j-1} = \max_{I_{i,j-1}} |R_i|$ is small, then k_{ij} will be large, and as a result r_{ij} will also be large. Consequently, $k_{i,j+1}$ and $r_{i,j+1}$ will be small, and so on. To avoid these oscillations, we adjust the time step k_{ij} according to Algorithm 6, which determines the new time step as a weighted harmonic mean value of the previous time step and the time step given by (7). Alternatively, DOLFIN provides time step control based on the (PID) controllers presented in [22; 42], including H0211 and H211PI. However, the simple controller of Algorithm 6 performs well compared to the more sophisticated controllers in [22; 42]. A suitable value for the weight w in Algorithm 6 is $w = 5$.

The initial time steps $k_{11} = k_{21} = \cdots = k_{N1} = K_1$ are chosen equal for all components and are determined iteratively for the first time slab. The size K_1

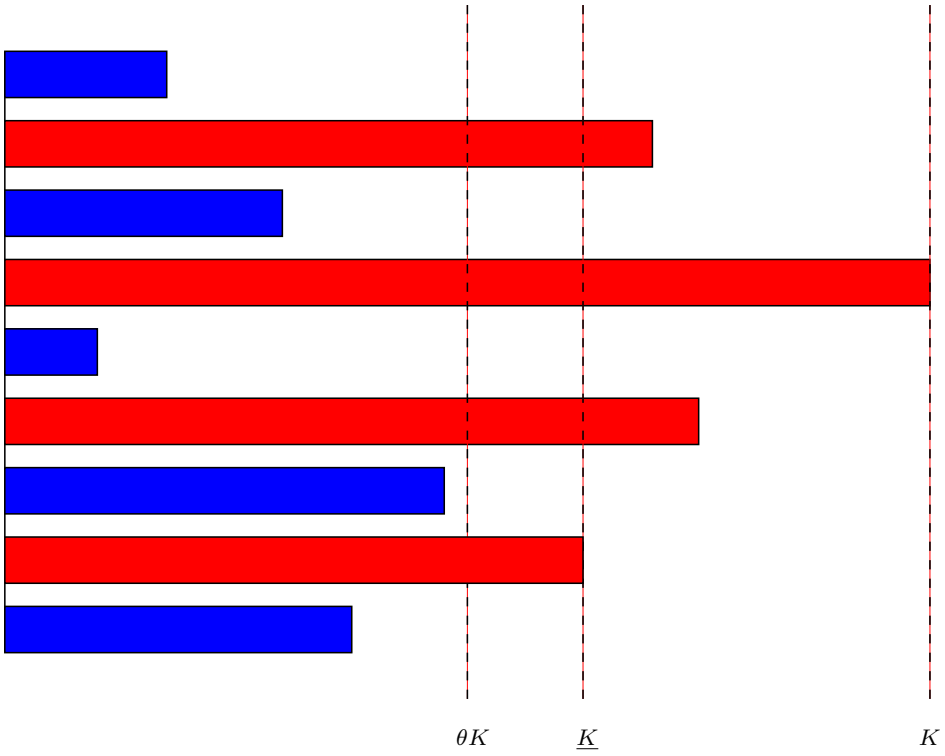


Fig. 3. The partition of components into groups of small and large time steps for $\theta = 1/2$.

Algorithm 6 $k = \text{Controller}(k_{\text{new}}, k_{\text{old}}, k_{\text{max}})$

$k \leftarrow (1 + w)k_{\text{old}}k_{\text{new}}/(k_{\text{old}} + wk_{\text{new}})$
$k \leftarrow \min(k, k_{\text{max}})$

of the first time slab is first initialized to some default value, possibly based on the length T of the time interval, and then adjusted until the local residuals are sufficiently small for all components.

3.4 Interpolation of the solution

To update the degrees of freedom on an element according to (4), the appropriate component f_i of the right-hand side of (1) needs to be evaluated at the set of quadrature points. In order for f_i to be evaluated, each component $U_{i'}$ of the computed solution U on which f_i depends has to be evaluated at the quadrature points. We let $\mathcal{S}_i \subseteq \{1, \ldots, N\}$ denote the *sparsity pattern* of component U_i, that is, the set of components on which f_i depends,

$$\mathcal{S}_i = \{i' \in \{1, \ldots, N\} : \partial f_i / \partial u_{i'} \neq 0\}. \tag{8}$$

Thus, to evaluate f_i at a given quadrature point t, only the components $\{U_{i'}\}_{i' \in S_i}$ need to be evaluated at t, as in Algorithm 7. This is of particular importance for problems of sparse structure and enables efficient multi-adaptive time integration

of time-dependent PDEs, as demonstrated below in Section 5. The sparsity pattern $\mathcal{S}_i$ is automatically detected by the solver. Alternatively, the sparsity pattern can be specified by a (sparse) matrix.

Algorithm 7 $y = \text{EvaluateRightHandSide}(i,\, t)$

for $i' \in \mathcal{S}_i$
    $x(i') \leftarrow U_{i'}(t)$
end for
$y \leftarrow f_i(x, t)$

3.5 Implementation of general elements

The system of discrete equations given by the variational problem (2) for the degrees freedom $\{\xi_{ijm}\}$ on each element I_{ij} takes the form

$$\xi_{ijm} = \xi_{ij0} + \int_{I_{ij}} w_m^{[q_{ij}]}(\tau_{ij}(t)) f_i(U(t), t)\,\mathrm{d}t, \quad m = 1, \ldots, q_{ij}, \tag{9}$$

for the mcG(q) method, where $\tau_{ij}(t) = (t - t_{i,j-1})/(t_{ij} - t_{i,j-1})$ and where $\{w_m^{[q_{ij}]}\}_{m=1}^{q_{ij}} \subset P^{[q_{ij}-1]}([0,1])$ are polynomial weight functions. For the mdG(q) method, the system of equations on each element has a similar form, with $m = 0, \ldots, q_{ij}$. As pointed out above in Section 2, the discrete equations take the form (4) for any particular choice of quadrature used to evaluate the integral in (9).

Thus, the weight functions $\{w_m^{[q_{ij}]}\}_{m=1}^{q_{ij}}$ need to be evaluated at a set of quadrature points $\{s_n\} \subset [0,1]$. In DOLFIN, these values are computed and tabulated each time a new type of element is created. If the same method is used for all components throughout the computation, then this computation is carried out only once.

For the mcG(q) method, Lobatto quadrature with $n = q + 1$ quadrature points is used. The $n \geq 2$ Lobatto quadrature points are defined on $[-1, 1]$ as the two end-points together with the roots of the derivative P'_{n-1} of the $(n-1)$th-order Legendre polynomial. The quadrature points are computed in DOLFIN using Newton's method to find the roots of P'_{n-1} on $[-1, 1]$, and are then rescaled to the interval $[0, 1]$.

Similarly, Radau quadrature with $n = q + 1$ quadrature points is used for the mdG(q) method. The $n \geq 1$ Radau points are defined on $[-1, 1]$ as the roots of $Q_n = P_{n-1} + P_n$, where P_{n-1} and P_n are Legendre polynomials. Note that the left end-point is always a quadrature point. As for the mcG(q) method, Newton's method is used to find the roots of Q_n on $[-1, 1]$. The quadrature points are then rescaled to $[0, 1]$, with time reversed to include the right end-point.

Since Lobatto quadrature with n quadrature points is exact for polynomials of degree $p \leq 2n - 3$ and Radau quadrature with n quadrature points is exact for polynomials of degree $p \leq 2n - 2$, both quadrature rules are exact for polynomials of degree $n - 1$ for $n \geq 2$ and $n \geq 1$, respectively. With both quadrature rules, the integral of the Legendre polynomial P_p on $[-1, 1]$ should thus be zero for $p = 0, \ldots, n - 1$. This defines a linear system, which is solved to obtain the quadrature weights.

After the quadrature points $\{s_n\}_{n=0}^{q_{ij}}$ have been determined, the polynomial weight functions $\{w_m^{[q_{ij}]}\}_{m=1}^{q_{ij}}$ are computed as described in [35] (again by solving a linear system) and then evaluated at the quadrature points. Multiplying these values with the quadrature weights, we rewrite (9) in the form (4).

4. DATA STRUCTURES

The solution on a standard mono-adaptive time slab, that is, a time slab constructed with equal time steps for all components, is typically stored as an array of values at the right end-point of the time slab, or as a list of arrays (possibly stored as one contiguous array) for a higher order method with several stages. A different data structure is needed to store the multi-adaptive solution on a time slab, such as the one in Figure 1. Such a data structure should ideally store the solution with minimal overhead compared to the cost of storing only the array of degrees of freedom for the solution on the time slab. In addition, it should also allow for efficient interpolation of the solution, that is, accessing the values of the solution for all components at any given time within the time slab. We present below a data structure that allows efficient storage of the entire solution on a time slab with little overhead, and at the same time allows efficient interpolation with $\mathcal{O}(1)$ access to any given value during the iterative solution of the system of discrete equations.

4.1 Representing the solution

The multi-adaptive solution on a time-slab can be efficiently represented using a data structures consisting of eight arrays as shown in Table I. For simplicity, we assume that all elements in a time slab are constructed for the same choice of method, mcG(q) or mdG(q), for a given fixed q.

The recursive construction of time slabs as discussed in Section 3.2 generates a sequence of *sub slabs*, each containing a list of *elements* (an element group). For each sub slab, we store the value of the time t at the left end-point and at the right end-point in the two arrays `sa` and `sb`. Thus, for sub slab number s covering the interval (a_s, b_s), we have

$$\begin{aligned} a_s &= \texttt{sa[}s\texttt{]}, \\ b_s &= \texttt{sb[}s\texttt{]}. \end{aligned} \tag{10}$$

Furthermore, for all elements in the (root) time slab, we store the degrees of freedom in the order they are created in the array `jx`. Thus, if each element has q degrees of freedom, as in the case of the multi-adaptive mcG(q) method, then the length of the array `jx` is q times the number of elements. In particular, if all components use the same time steps, then the length of the array `jx` is qN.

4.2 Interpolating the solution

For each element, we store the corresponding component index i in the array `ei` in order to be able to evaluate the correct component f_i of the right-hand side f of (1) when iterating over all elements in the time slab to update the degrees of freedom. When updating the values on an element according to (4), it is also necessary to know the left and right end-points of the elements. Thus, we store an array `es` that maps the number of a given element to the number of the corresponding sub slab

containing the element. As a consequence, the left end-point a_e and right end-point b_e for a given element e are given by

$$\begin{aligned} a_e &= \texttt{sa}[\texttt{es}[e]], \\ b_e &= \texttt{sb}[\texttt{es}[e]]. \end{aligned} \tag{11}$$

Array	Type	Description
`sa`	`double`	left end-points for sub slabs
`sb`	`double`	right end-points for sub slabs
`jx`	`double`	values for degrees of freedom
`ei`	`int`	component indices for elements
`es`	`int`	time slabs containing elements
`ee`	`int`	previous elements for elements
`ed`	`int`	first dependencies for elements
`de`	`int`	elements for dependencies

Table I. Data structures for efficient representation of a multi-adaptive time slab.

Updating the values on an element according to (4) also requires knowledge of the value at the left end-point, which is given as the end-time value on the previous element in the time slab for the same component (or the end-time value from the previous time slab). This information is available in the array `ee`, which stores for each element the number of the previous element (or -1 if there is no previous element).

The discrete system of equations on each time slab is solved by iterating over the elements in the time slab and updating the values on each element, either in a direct fixed-point iteration or a Newton's method. We must then for any given element e corresponding to some component $i = \texttt{ei}[e]$ evaluate the right-hand side f_i at each quadrature point t within the element. This requires the values of the solution $U(t)$ at t for all components contained in the sparsity pattern $\mathcal{S}_i$ for component i according to Algorithm 7. As a consequence of Algorithm 2 for the recursive construction of time slabs, elements for components that use large time steps are constructed before elements for components that use small time steps. Since all elements of the time slab are traversed in the same order during the iterative solution of the system of discrete equations, elements corresponding to large time steps have recently been visited and cover any element that corresponds to a smaller time step. The last visited element for each component is stored in an auxiliary array `elast` of size N. Thus, if $i' \in S_i$ and component i' has recently been visited, then it is straight-forward to find the latest element for component i' that covers the current element for component i and interpolate $U_{i'}$ at time t. It is also straight-forward to interpolate the values for any components that are present in the same element group as the current element.

However, when updating the values on an element e corresponding to some component $i = \texttt{ei}[e]$ depending on some other component $i' \in \mathcal{S}_i$ which uses smaller time steps, one must find for each quadrature point t on the element e the element e' for component i' containing t, which is non-trivial. The element e' can be found by searching through all elements for component i' in the time slab, but this quickly

becomes inefficient. Instead, we store for each element e a list of dependencies to elements with smaller time steps in the two arrays `ed` and `de`. These two arrays store a sparse integer matrix of dependencies to elements with smaller time steps for all elements in the time slab. Thus, for any given element e, the list of elements with smaller time steps that need to be interpolated are given by

$$\{\texttt{de}[\texttt{ed}[e]], \texttt{de}[\texttt{ed}[e]+1], \ldots, \texttt{de}[\texttt{ed}[e+1]]\}. \tag{12}$$

5. NUMERICAL EXAMPLES AND BENCHMARK RESULTS

In this section, we present a number of examples to illustrate various aspects of multi-adaptive time-stepping. All three examples are time-dependent PDEs that we discretize in space using the cG(1) finite element method to obtain a system of ODEs, sometimes referred to as the method of lines approach. In each case, we lump and invert the mass matrix so as to obtain a system of the form (1).

The first of the three examples demonstrates qualitatively the basic principles of multi-adaptive time-stepping, with small time steps only in regions where the local residual is large.

The second and third examples demonstrate quantitatively the benefits of multi-adaptive time-stepping compared to standard methods. In the first of these two benchmark problems, the individual time steps are chosen automatically based on an a posteriori error estimate and in the second, the time steps are fixed in time and vary in space according to the CFL condition so that $k \sim h$ locally.

5.1 The bistable equation

As a first example, we solve the bistable equation on the unit cube,

$$\begin{aligned} \dot{u} - \epsilon\Delta u &= u(1-u^2) \quad \text{in } \Omega \times (0,T], \\ \partial_n u &= 0 \quad \text{on } \partial\Omega \times (0,T], \\ u(\cdot,0) &= u_0 \quad \text{in } \Omega, \end{aligned} \tag{13}$$

with $\Omega = (0,1) \times (0,1) \times (0,1)$, $\epsilon = 0.0001$, final time $T = 100$, and with random initial data $u_0 = u_0(x)$ distributed uniformly on $[-1,1]$.

The bistable equation has been studied extensively before [17; 14] and has interesting stability properties. In particular, it has two stable steady-state solutions, $u = 1$ and $u = -1$, and one unstable steady-state solution, $u = 0$. From (13), it is clear that the solution increases in regions where it is positive and decreases in regions where it is negative. Because of the diffusion, neighboring regions will compete until finally the solution has reached one of the two stable steady states. Since this action is local on the interface between positive and negative regions, the bistable equation is an ideal example for multi-adaptive time-stepping.

The solution was computed on a uniformly refined tetrahedral mesh with mesh size $h = 1/64$. This mesh consists of $1,572,864$ tetrahedrons and has $N = 274,625$ vertices. In Figure 4, we plot the initial value used for the computation together with the solution at final time $t = 100$. We also plot the solution and the multi-adaptive time steps at time $t = 10$ in Figure 5, and note that the time steps are small in regions where there is strong competition between the two stable steady-state solutions, in particular in regions with where the curvature of the interface is small.

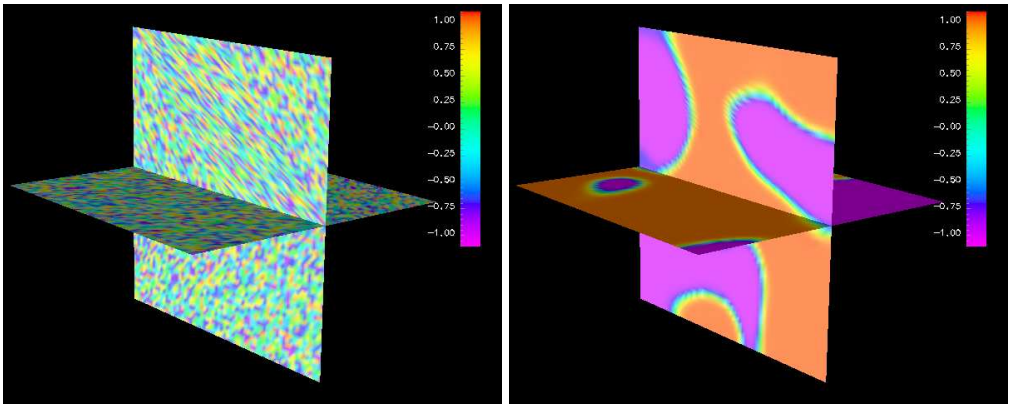


Fig. 4. Initial data (left) and final solution at time $t = 100$ (right) for the bistable equation (13).

5.2 A nonlinear reaction-diffusion equation

As a second example, we solve the following nonlinear reaction-diffusion equation, taken from [41]:

$$\begin{aligned} \dot{u} - \epsilon u'' &= \gamma u^2(1-u) \quad \text{in } \Omega \times (0,T], \\ \partial_n u &= 0 \quad \text{on } \partial\Omega \times (0,T], \\ u(\cdot,0) &= u_0 \quad \text{in } \Omega, \end{aligned} \tag{14}$$

with $\Omega = (0, L)$, $\epsilon = 0.01$, $\gamma = 1000$ and final time $T = 1$.

The equation is discretized in space with the standard cG(1) method using a uniform mesh with 1000 mesh points. The initial data is chosen according to

$$u_0(x) = \frac{1}{1 + \exp(\lambda(x-1))}. \tag{15}$$

The resulting solution is a reaction front, sweeping across the domain from left to right, as demonstrated in Figure 6. The multi-adaptive time steps are automatically selected to be small in and around the reaction front and sweep the domain at the same velocity as the reaction front, as demonstrated in Figure 7.

To study the performance of the multi-adaptive solver, we compute the solution for a range of tolerances with $L = 1$ and compare the resulting error and CPU time with a standard mono-adaptive solver that uses equal (adaptive) time steps for all components. To make the comparison fair, we compare the multi-adaptive mcG(q) method with the mono-adaptive cG(q) method for $q = 1$. Both methods are implemented for general order q in the same programming language (C++) within a common framework (DOLFIN), but the mono-adaptive method takes full advantage of the fact that the time steps equal for all components. In particular, the mono-adaptive solver may use much simpler data structures (a plain C array) to store the solution on each time slab and there is no overhead for interpolation of the solution. This is a more difficult benchmark than only comparing the number degrees of freedom (local steps) as in [41] or comparing the CPU time against the same multi-adaptive solver when it is forced to use identical time steps for all components as in [35], since it also takes into account the overhead of the more

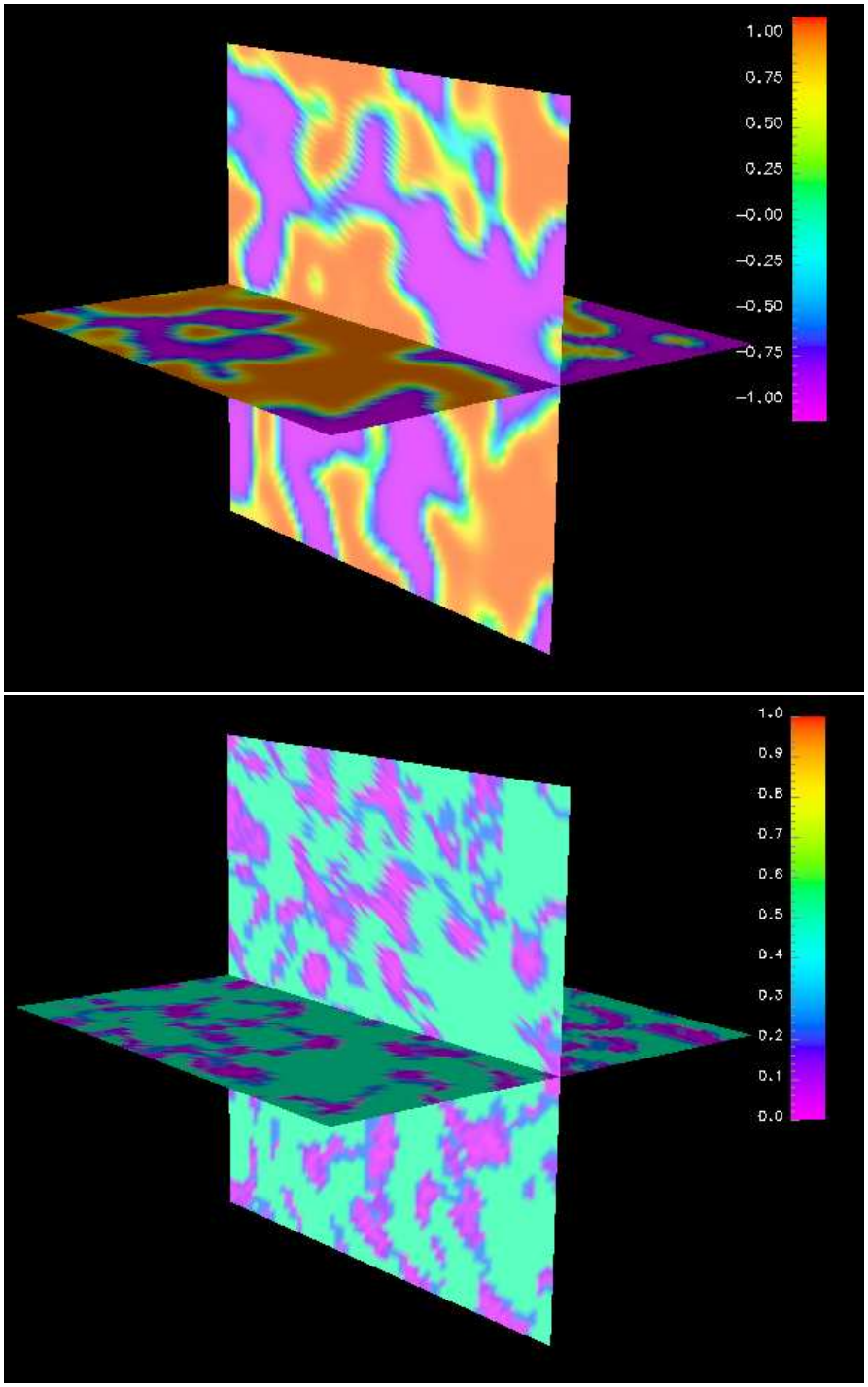


Fig. 5. Solution (above) and multi-adaptive time steps (below) at time $t = 10$ for the bistable equation (13).

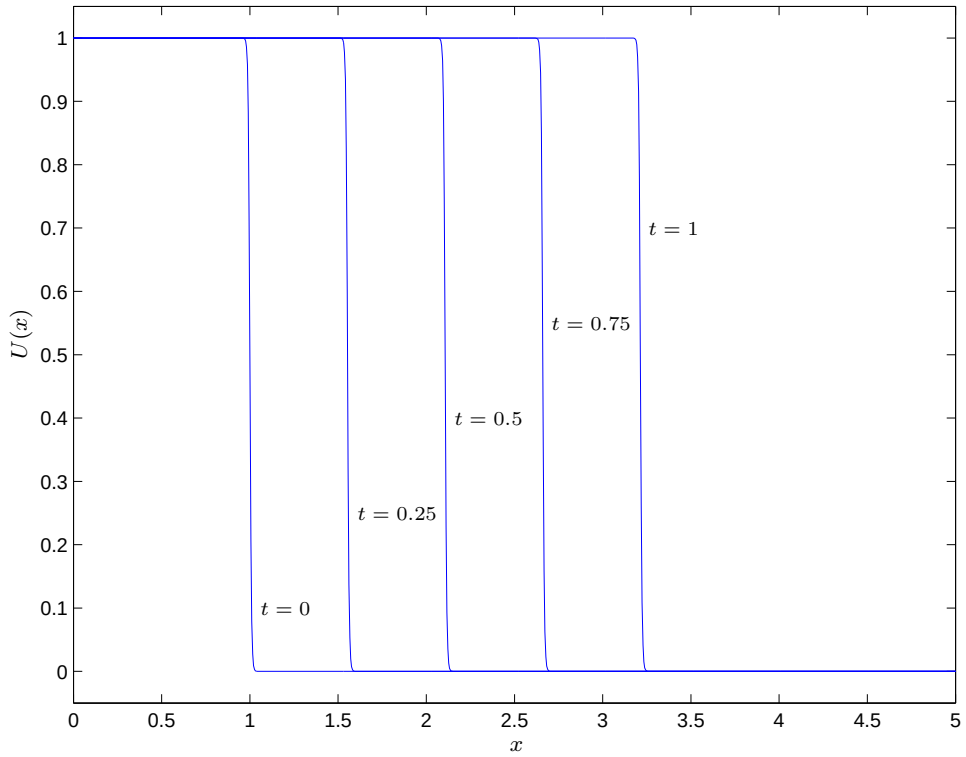


Fig. 6. Propagation of the solution of the reaction–diffusion problem (14).

complicated algorithms and data structures necessary for the implementation of multi-adaptive time-stepping.

Note that we don't solve the dual problem to compute stability factor (or stability weights) which is necessary to obtain a reliable error estimate. Thus, the tolerance controls only the size of the error modulo the stability factor, which is unknown.

In addition, we also compare the two methods for varying size L of the domain Ω, keeping the same initial conditions but scaling the number of mesh points according to the length of the domain, $N = 1000L/5$. As the size of the domain increases, we expect the relative efficiency of the multi-adaptive method to increase, since the number of inactive components increases relative to the number of components located within the reaction front.

In Figure 8, we plot the CPU time as function of the tolerance and number of components (size of domain) for the mcG(1) and cG(1) methods. We also summarize the results in Table II and Table III. As expected, the speedup expressed as the multi-adaptive efficiency index μ, that is, the ideal speedup if the cost per degree of freedom were the same for the multi- and mono-adaptive methods, is large in all test cases, around a factor 100. The speedup in terms of the total number of time slabs is also large. Note that in Table II, the total number of time slabs M remains practically constant as the tolerance and the error are decreased. The decreased tolerance instead results in finer local resolution of the reaction front, which is evident from the increasing multi-adaptive efficiency index. At the same time,

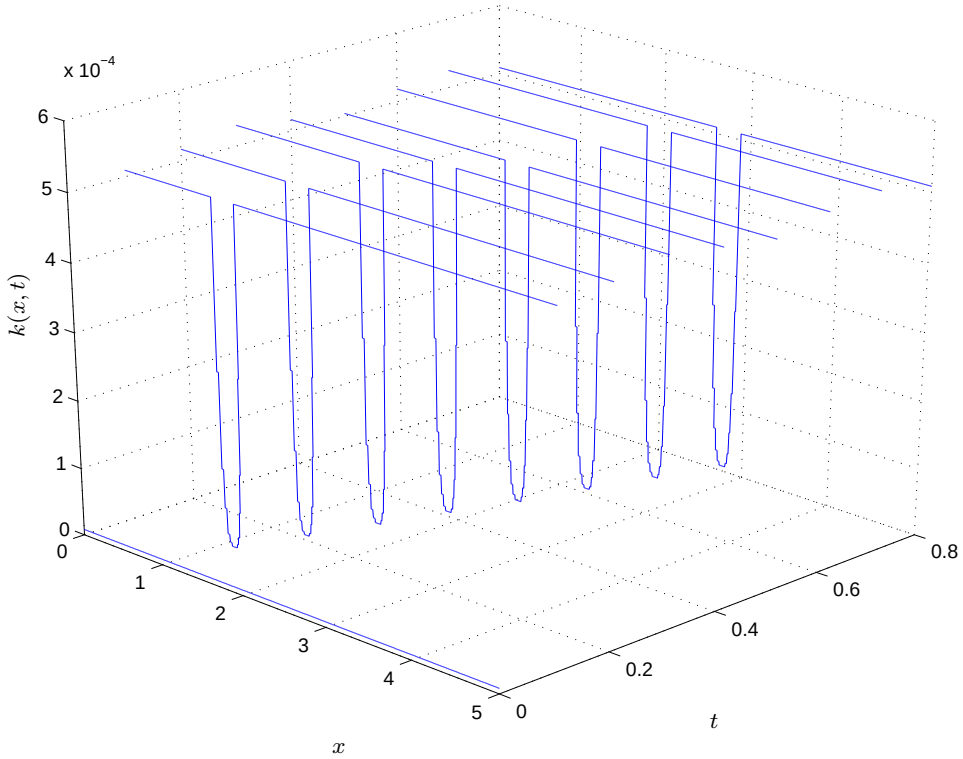


Fig. 7. The multi-adaptive time steps as function of space at a sequence of time intervals for the test problem (14).

the mono-adaptive method needs to decrease the time step for all components and so the relative efficiency of the multi-adaptive method increases as the tolerance decreases. See also Figure 9 for a comparison of the multi-adaptive time steps at two different tolerances.

The situation is slightly different in Table III, where the tolerance is kept constant but the size of the domain and number of components vary. Here, the number of time slabs remains practically constant for both methods, but the multi-adaptive efficiency index increases as the size of the domain increases, since the reaction front then becomes more and more localized relative to the size of the domain. As a result, the efficiency index of the multi-adaptive method increases as the size of the domain is increased.

In all test cases, the multi-adaptive method is more efficient than the standard mono-adaptive method also when the CPU time (wall-clock time) is chosen as a metric for the comparison. In the first set of test cases with varying tolerance, the actual speedup is about a factor 2.0 whereas in the second test case with varying size of the domain, the speedup increases from about a factor 2.0 to a factor 5.7 for the range of test cases. These are significant speedups, although far from the ideal speedup which is given by the multi-adaptive efficiency index.

There are mainly two reasons that make it difficult to attain full speedup. The first reason is that as the size of the time slab increases, the number of iterations n

needed to solve the system of discrete equations increases. In Table III, the number of iterations, including local iterations on individual elements as part of a global iteration on the time slab, is about a factor 1.5 larger for the multi-adaptive method. However, the main overhead lies in the more straightforward implementation of the mono-adaptive method compared to the more complicated data structures needed to store and interpolate the multi-adaptive solution. For constant time step and equal time step for all components, this overhead is roughly a factor 5 for the test problem, but the overhead increases to about a factor 100 when the time slab is locally refined. It thus remains important to further reduce the overhead of the implementation in order to increase the range of problems where the multi-adaptive methods give a positive speedup.

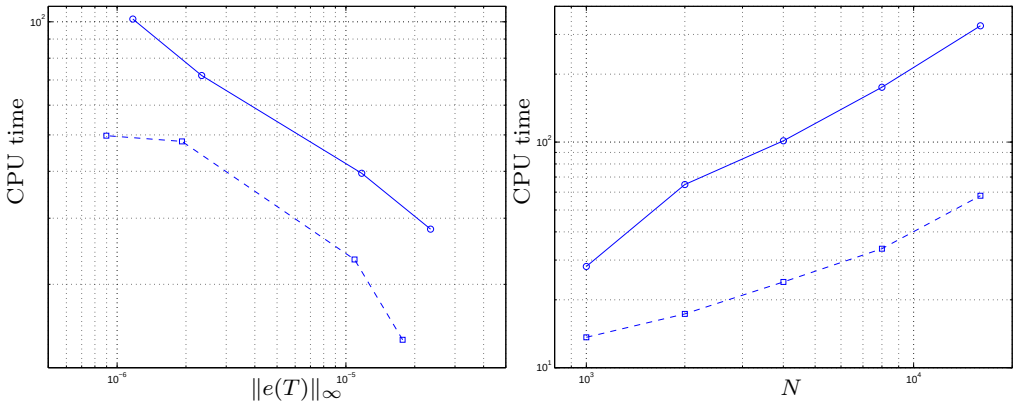


Fig. 8. CPU time as function of the local tolerance tol $\sim$ TOL (left) and number of components N (right) for mcG(1) (dashed line) and cG(1) (solid line).

tol	$\|e(T)\|_\infty$	time	M	n	μ
$1.0 \cdot 10^{-6}$	$1.8 \cdot 10^{-5}$	14.2 s	1922 (5)	3.990 (1.498)	95.3
$5.0 \cdot 10^{-7}$	$1.1 \cdot 10^{-5}$	23.3 s	1912 (9)	4.822 (1.544)	138.2
$1.0 \cdot 10^{-7}$	$1.9 \cdot 10^{-6}$	48.1 s	1929 (7)	4.905 (1.594)	142.6
$5.0 \cdot 10^{-8}$	$9.0 \cdot 10^{-7}$	49.8 s	1917 (7)	4.131 (1.680)	172.4
tol	$\|e(T)\|_\infty$	time	M	n	μ
$1 \cdot 10^{-6}$	$2.3 \cdot 10^{-5}$	28.1 s	117089 (1)	4.0	1.0
$5 \cdot 10^{-7}$	$1.2 \cdot 10^{-5}$	39.5 s	165586 (1)	4.0	1.0
$1 \cdot 10^{-7}$	$2.3 \cdot 10^{-6}$	71.9 s	370254 (1)	3.0	1.0
$5 \cdot 10^{-8}$	$1.2 \cdot 10^{-6}$	101.7 s	523615 (1)	3.0	1.0

Table II. Benchmark results for mcG(1) (above) and cG(1) below for varying tolerance and fixed number of components $N = 1000$.

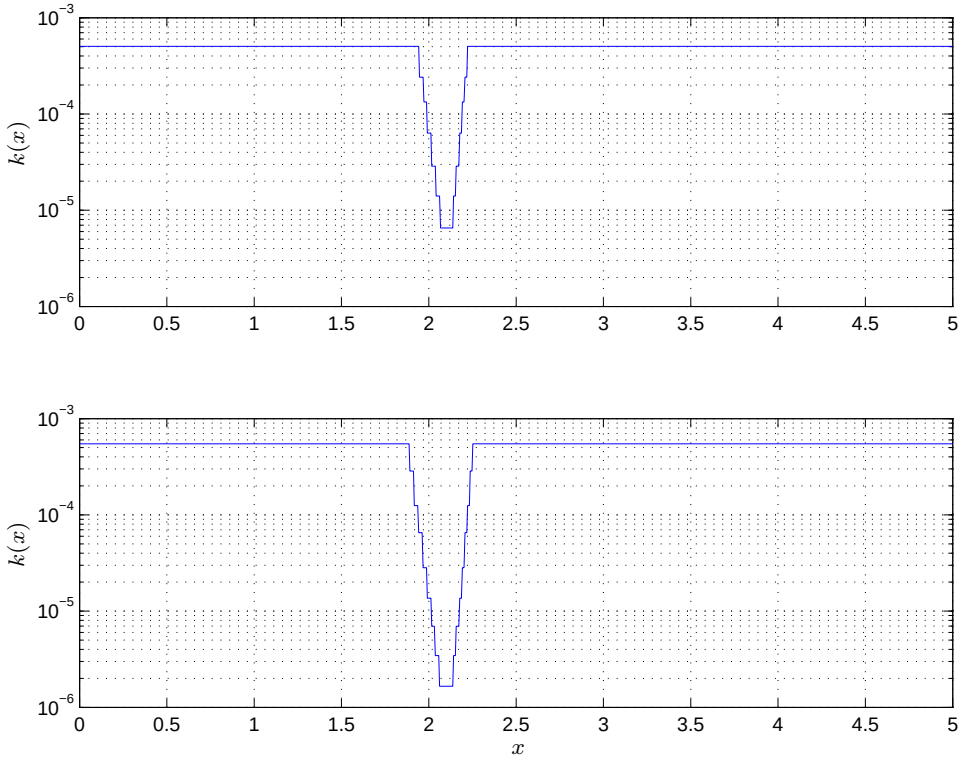


Fig. 9. Multi-adaptive time steps at $t = 0.5$ for two different tolerances.

N	$\|e(T)\|_\infty$	time	M	n	μ
1000	$1.8 \cdot 10^{-5}$	13.6 s	1922 (5)	4.0 (1.5)	95.3
2000	$1.7 \cdot 10^{-5}$	17.3 s	1923 (5)	4.0 (1.2)	140.5
4000	$1.6 \cdot 10^{-5}$	24.0 s	1920 (6)	4.0 (1.0)	185.0
8000	$1.7 \cdot 10^{-5}$	33.7 s	1918 (5)	4.0 (1.0)	218.8
16000	$1.7 \cdot 10^{-5}$	57.9 s	1919 (5)	4.0 (1.0)	234.0
N	$\|e(T)\|_\infty$	time	M	n	μ
1000	$2.3 \cdot 10^{-5}$	28.1 s	117089 (1)	4.0	1.0
2000	$2.2 \cdot 10^{-5}$	64.8 s	117091 (1)	4.0	1.0
4000	$2.2 \cdot 10^{-5}$	101.3 s	117090 (1)	4.0	1.0
8000	$2.2 \cdot 10^{-5}$	175.1 s	117089 (1)	4.0	1.0
16000	$2.2 \cdot 10^{-5}$	327.7 s	117089 (1)	4.0	1.0

Table III. Benchmark results for mcG(1) (above) and cG(1) below for fixed tolerance tol $= 1.0 \cdot 10^{-6}$ and varying number of components (and size of domain).

5.3 The wave equation

As a final example, we consider the wave equation,

$$\begin{aligned} \ddot{u} - \Delta u &= 0 \quad \text{in } \Omega \times (0, T], \\ \partial_n u &= 0 \quad \text{on } \partial\Omega \times (0, T], \\ u(\cdot, 0) &= u_0 \quad \text{in } \Omega, \end{aligned} \tag{16}$$

on a two-dimensional domain Ω consisting of two square sub domains of length 0.5 separated by a thin wall with a narrow slit of size 0.0001×0.0001 at its center. The initial condition is chosen as a plane wave traversing the domain from right to left. In Figure 10, we plot the initial data together with the (fixed) multi-adaptive time steps. The resulting solution is shown in Figure 11 at a sequence of time intervals.

The geometry of the domain Ω forces the discretization to be very fine close to the narrow slit. As a result, the CFL condition puts a limit on the size of the time step, roughly given by

$$k \leq h_{\min} = \min_{x \in \Omega} h(x), \tag{17}$$

where $h = h(x)$ is the local mesh size. With a larger time step, an explicit method will be unstable or, correspondingly, direct fixed-point iteration on the systems of discrete equations on each time slab will not converge without suitable stabilization.

On the other hand, with a multi-adaptive method, the time step may be chosen to satisfy the CFL condition only locally, that is,

$$k(x) \leq h(x), \quad x \in \Omega, \tag{18}$$

and as a result, the number of local steps may decrease significantly (depending on the properties of the mesh) and as a result the total work may decrease (depending on the size of the overhead for the multi-adaptive method). The speedup for the multi-adaptive mcG(1) method was a factor 4.2.

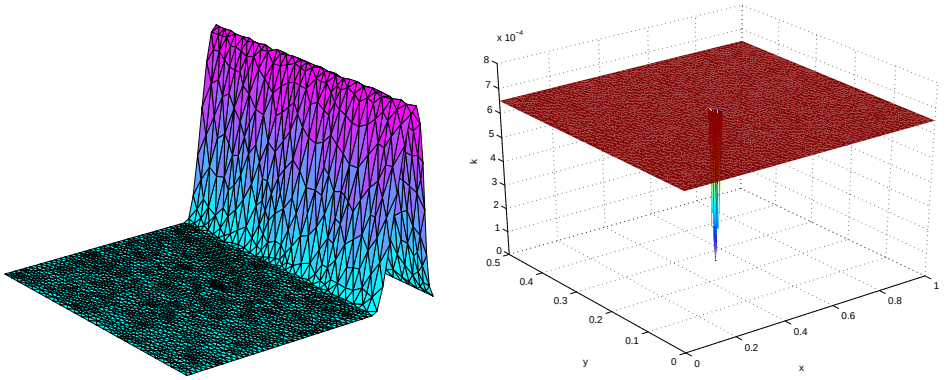


Fig. 10. Initial data (left) and multi-adaptive time steps (right) for the solution of the wave equation.

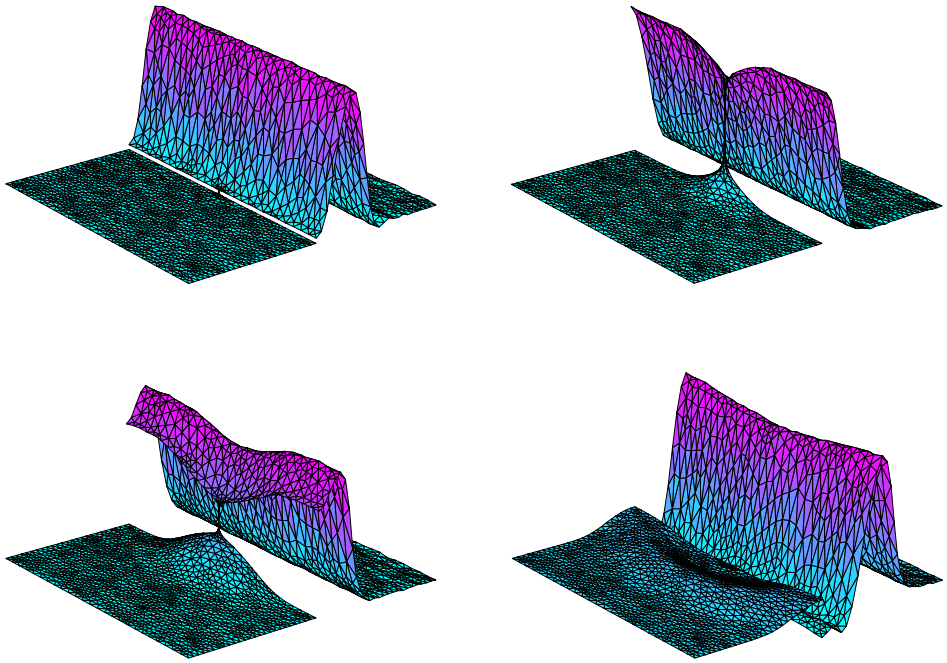


Fig. 11. The solution of the wave equation at times $t = 0.25$, $t = 0.4$, $t = 0.45$ and $t = 0.6$.

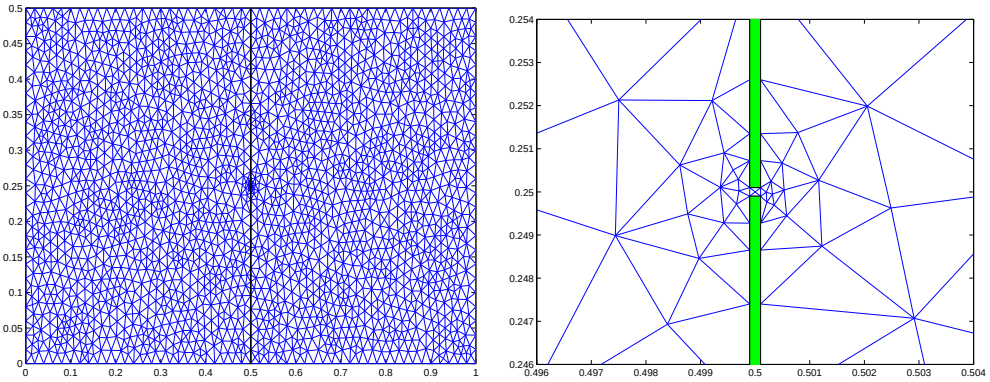


Fig. 12. The mesh used for the solution of the wave equation on a domain intersected by a thin wall with a narrow slit (left) and details of the mesh close to the slit (right).

REFERENCES

S. G. Alexander and C. B. Agnor, *n-body simulations of late stage planetary formation with a simple fragmentation model*, ICARUS, 132 (1998), pp. 113–124.

R. Dav, J. Dubinski, and L. Hernquist, *Parallel treeSPH*, New Astronomy, 2 (1997), pp. 277–297.

C. Dawson and R. C. Kirby, *High resolution schemes for conservation laws with locally varying time steps*, SIAM J. Sci. Comput., 22, No. 6 (2001), pp. 2256–2281.

M. Delfour, W. Hager, and F. Trochu, *Discontinuous Galerkin methods for ordinary differential equations*, Math. Comp., 36 (1981), pp. 455–473.

T. Dupont, J. Hoffman, C. Johnson, R. C. Kirby, M. G. Larson, A. Logg, and L. R. Scott, *The FEniCS project*, Tech. Rep. 2003–21, Chalmers Finite Element Center Preprint Series, 2003.

K. Eriksson, D. Estep, P. Hansbo, and C. Johnson, *Introduction to adaptive methods for differential equations*, Acta Numerica, 4 (1995), pp. 105–158.

K. Eriksson and C. Johnson, *Adaptive finite element methods for parabolic problems III: Time steps variable in space*, in preparation.

———, *Adaptive finite element methods for parabolic problems I: A linear model problem*, SIAM J. Numer. Anal., 28, No. 1 (1991), pp. 43–77.

———, *Adaptive finite element methods for parabolic problems II: Optimal order error estimates in $l_\infty l_2$ and $l_\infty l_\infty$*, SIAM J. Numer. Anal., 32 (1995), pp. 706–740.

———, *Adaptive finite element methods for parabolic problems IV: Nonlinear problems*, SIAM J. Numer. Anal., 32 (1995), pp. 1729–1749.

———, *Adaptive finite element methods for parabolic problems V: Long-time integration*, SIAM J. Numer. Anal., 32 (1995), pp. 1750–1763.

K. Eriksson, C. Johnson, and S. Larsson, *Adaptive finite element methods for parabolic problems VI: Analytic semigroups*, SIAM J. Numer. Anal., 35 (1998), pp. 1315–1325.

K. Eriksson, C. Johnson, and V. Thome, *Time discretization of parabolic problems by the discontinuous Galerkin method*, RAIRO MAN, 19 (1985), pp. 611–643.

D. Estep, *An analysis of numerical approximations of metastable solutions of the bistable equation*, Nonlinearity, 7 (1994), pp. 1445–1462.

———, *A posteriori error bounds and global error control for approximations of ordinary differential equations*, SIAM J. Numer. Anal., 32 (1995), pp. 1–48.

D. Estep and D. French, *Global error control for the continuous Galerkin finite element method for ordinary differential equations*, M^2AN, 28 (1994), pp. 815–852.

D. Estep, M. Larson, and R. Williams, *Estimating the error of numerical solutions of systems of nonlinear reaction–diffusion equations*, Memoirs of the American Mathematical Society, 696 (2000), pp. 1–109.

D. Estep and A. Stuart, *The dynamical behavior of the discontinuous Galerkin method and related difference schemes*, Math. Comp., 71 (2002), pp. 1075–1103.

D. Estep and R. Williams, *Accurate parallel integration of large sparse systems of differential equations*, Math. Models. Meth. Appl. Sci., 6 (1996), pp. 535–568.

J. E. Flaherty, R. M. Loy, M. S. Shephard, B. K. Szymanski, J. D. Teresco, and L. H. Ziantz, *Adaptive local refinement with octree load balancing for the parallel solution of three-dimensional conservation laws*, Journal of Parallel and Distributed Computing, 47 (1997), pp. 139–152.

Free Software Foundation, *GNU GPL*, 1991. URL: `http://www.gnu.org/copyleft/gpl.html`.

K. Gustafsson, M. Lundh, and G. Sderlind, *A PI stepsize control for the numerical solution of ordinary differential equations*, BIT, 28 (1988), pp. 270–287.

J. Hoffman, J. Jansson, C. Johnson, M. G. Knepley, R. C. Kirby, A. Logg, L. R. Scott, and G. N. Wells, *FEniCS*, 2006. `http://www.fenics.org/`.

J. Hoffman, J. Jansson, A. Logg, and G. N. Wells, *DOLFIN*, 2006. `http://www.fenics.org/dolfin/`.

J. Hoffman and A. Logg, *DOLFIN: Dynamic Object oriented Library for FINite element computation*, Tech. Rep. 2002–06, Chalmers Finite Element Center Preprint Series, 2002.

T. J. R. Hughes, I. Levit, and J. Winget, *Element-by-element implicit algorithms for heat-conduction*, J. Eng. Mech.-ASCE, 109 (1983), pp. 576–585.

———, *An element-by-element solution algorithm for problems of structural and solid mechanics*, Computer Methods in Applied Mechanics and Engineering, 36 (1983), pp. 241–254.

B. L. Hulme, *Discrete Galerkin and related one-step methods for ordinary differential equations*, Math. Comput., 26 (1972), pp. 881–891.

———, *One-step piecewise polynomial Galerkin methods for initial value problems*, Math. Comput., 26 (1972), pp. 415–426.

P. Jamet, *Galerkin-type approximations which are discontinuous in time for parabolic equations in a variable domain*, SIAM J. Numer. Anal., 15 (1978), pp. 912–928.

C. Johnson, *Error estimates and adaptive time-step control for a class of one-step methods for stiff ordinary differential equations*, SIAM J. Numer. Anal., 25 (1988), pp. 908–926.

R. C. Kirby and A. Logg, *A compiler for variational forms*, to appear in ACM Trans. Math. Softw., (2006).

———, *Optimizing the FEniCS Form Compiler FFC: Efficient pretabulation of integrals*. 2006.

A. Lew, J. E. Marsden, M. Ortiz, and M. West, *Asynchronous variational integrators*, Arch. Rational. Mech. Anal., 167 (2003), pp. 85–146.

A. Logg, *Multi-adaptive Galerkin methods for ODEs I*, SIAM J. Sci. Comput., 24 (2003), pp. 1879–1902.

———, *Multi-adaptive Galerkin methods for ODEs II: Implementation and applications*, SIAM J. Sci. Comput., 25 (2003), pp. 1119–1141.

———, *FFC*, 2006. `http://www.fenics.org/ffc/`.

———, *Multi-adaptive Galerkin methods for ODEs III: A priori error estimates*, SIAM J. Numer. Anal., 43 (2006), pp. 2624–2646.

J. Makino and S. Aarseth, *On a Hermite integrator with Ahmad-Cohen scheme for gravitational many-body problems*, Publ. Astron. Soc. Japan, 44 (1992), pp. 141–151.

S. Osher and R. Sanders, *Numerical approximations to nonlinear conservation laws with locally varying time and space grids*, Math. Comp., 41 (1983), pp. 321–336.

V. Savcenco, W. Hundsdorfer, and J. Verwer, *A multirate time stepping strategy for parabolic PDEs*, submitted to SIAM J. Sci. Comput., (2005).

G. Sderlind, *Digital filters in adaptive time-stepping*, ACM Trans. Math. Softw., 29 (2003), pp. 1–26.

SIMULATION OF MECHANICAL SYSTEMS WITH INDIVIDUAL TIME STEPS

JOHAN JANSSON AND ANDERS LOGG

Abstract. The simulation of a mechanical system involves the formulation of a differential equation (modeling) and the solution of the differential equation (computing). The solution method needs to be efficient as well as accurate and reliable. This paper discusses multi-adaptive Galerkin methods in the context of mechanical systems. The primary type of mechanical system studied is an extended mass–spring model. A multi-adaptive method integrates the mechanical system using individual time steps for the different components of the system, adapting the time steps to the different time scales of the system, potentially allowing enormous improvement in efficiency compared to traditional mono-adaptive methods.

1. Introduction

Simulation of mechanical systems is an important component of many technologies of modern society. It appears in industrial design, for the prediction and verification of mechanical products. It appears in virtual reality, both for entertainment in the form of computer games and movies, and in the simulation of realistic environments such as surgical training on virtual and infinitely resurrectable patients. Common to all these applications is that the computation time is critical. Often, an application is real-time, which means that the time inside the simulation must reasonably match the time in the real world.

Simulating a mechanical system involves both *modeling* (formulating an equation describing the system) and *computation* (solving the equation). The model of a mechanical system often takes the form of an initial value problem for a system of ordinary differential equations of the form

$$\begin{aligned} \dot{u}(t) &= f(u(t), t), \quad t \in (0, T], \\ u(0) &= u_0, \end{aligned} \tag{1.1}$$

where $u : [0, T] \to \mathbb{R}^N$ is the solution to be computed, $u_0 \in \mathbb{R}^N$ a given initial value, $T > 0$ a given final time, and $f : \mathbb{R}^N \times (0, T] \to \mathbb{R}^N$ a given function that is Lipschitz-continuous in u and bounded.

Date: April 27, 2004.

Key words and phrases. Multi-adaptivity, individual time steps, local time steps, ODE, continuous Galerkin, discontinuous Galerkin, mcgq, mdgq, mechanical system, mass–spring model.

Johan Jansson, *email*: johanjan@math.chalmers.se. Anders Logg, *email*: logg@math.chalmers.se. Department of Computational Mathematics, Chalmers University of Technology, SE–412 96 Göteborg, Sweden.

The simulation of a mechanical system thus involves the formulation of a model of the form (1.1) and the solution of (1.1) using a time-stepping method. We present below *multi-adaptive Galerkin* methods for the solution of (1.1) with individual time steps for the different parts of the mechanical system.

1.1. **Mass–spring systems.** A mass–spring system consists of a set of point masses connected by springs, typically governed by Hooke's law with other laws optionally present, such as viscous damping and external forces. Mass–spring systems appear to encompass most of the behaviors of elementary mechanical systems and thus represent a simple, intuitive, and powerful model for the simulation of mechanical systems. This is the approach taken in this paper.

However, to obtain a physically accurate model of a mechanical system, we believe it is necessary to solve a system of partial differential equations properly describing the mechanical system, in the simplest case given by the equations of linear elasticity. Discretizing the system of PDEs in space, for example using the Galerkin finite element method, an initial value problem for a system of ODEs of the form (1.1) is obtained. The resulting system can be interpreted as a mass–spring system and thus the finite element method in combination with a PDE model represents a systematic methodology for the generation of a mass–spring model of a given mechanical system.

1.2. **Time-stepping methods.** Numerical methods for the (approximate) solution of (1.1) are almost exclusively based on time-stepping, i.e., the step-wise integration of (1.1) to obtain an approximation U of the solution u satisfying

$$u(t_j) = u(t_{j-1}) + \int_{t_{j-1}}^{t_j} f(u(t), t)\,dt, \quad j = 1, \ldots, M, \tag{1.2}$$

for a partition $0 = t_0 < t_1 < \cdots < t_M = T$ of $[0, T]$. The approximate solution $U \approx u$ is obtained by an appropriate approximation of the integral $\int_{t_{j-1}}^{t_j} f(u(t), t)\,dt$.

Selecting the appropriate size of the time steps $\{k_j = t_j - t_{j-1}\}_{j=1}^M$ is essential for efficiency and accuracy. We want to compute the solution U using as little work as possible, which means using a small number of large time steps. At the same time, we want to compute an accurate solution U which is close to the exact solution u, which means using a large number of small time steps. Often, the accuracy requirement is given in the form of a *tolerance* TOL for the size of the *error* $e = U - u$ in a suitable norm. The competing goals of efficiency and accuracy can be met using an *adaptive* algorithm, determining a sequence of time steps $\{k_j\}_{j=1}^M$ which produces an approximate solution U satisfying the given tolerance with minimal work.

Galerkin finite element methods present a general framework for the numerical solution of (1.1), including adaptive algorithms for the automatic construction of an optimal time step sequence, see [7, 8]. The Galerkin finite element method for (1.1) reads: Find $U \in V$, such that

$$\int_0^T (\dot{U}, v)\,dt = \int_0^T (f, v)\,dt \quad \forall v \in \hat{V}, \tag{1.3}$$

where $(\cdot,\cdot)$ denotes the $\mathbb{R}^N$ inner product and $(V,\hat{V})$ denotes a suitable pair of finite dimensional subspaces (the *trial* and *test spaces*).

Typical choices of approximating spaces include

$$\begin{aligned} V &= \{v \in [\mathcal{C}([0,T])]^N : v|_{I_j} \in [\mathcal{P}^q(I_j)]^N, \quad j = 1,\ldots,M\}, \\ \hat{V} &= \{v : v|_{I_j} \in [\mathcal{P}^{q-1}(I_j)]^N, \quad j = 1,\ldots,M\}, \end{aligned} \tag{1.4}$$

i.e., V represents the space of continuous and piecewise polynomial vector-valued functions of degree $q \geq 1$ and $\hat{V}$ represents the space of discontinuous piecewise polynomial vector-valued functions of degree $q-1$ on a partition of $[0,T]$. We refer to this as the cG(q) method. With both V and $\hat{V}$ representing discontinuous piecewise polynomials of degree $q \geq 0$, we obtain the dG(q) method. Early work on the cG(q) and dG(q) methods include [6, 19, 10, 9].

By choosing a constant test function v in (1.3), it follows that both the cG(q) and dG(q) solutions satisfy the relation

$$U(t_j) = U(t_{j-1}) + \int_{t_{j-1}}^{t_j} f(U(t),t)\,dt, \quad j = 1,\ldots,M, \tag{1.5}$$

corresponding to (1.2).

In the simplest case of the dG(0) method, we note that $\int_{t_{j-1}}^{t_j} f(U(t),t)\,dt \approx k_j f(U(t_j),t_j)$, since U piecewise constant, with equality if f does not depend explicitly on t. We thus obtain the method

$$U(t_j) = U(t_{j-1}) + k_j f(U(t_j),t_j), \quad j = 1,\ldots,M, \tag{1.6}$$

which we recognize as the backward (or implicit) Euler method. In general, a cG(q) or dG(q) method corresponds to an implicit Runge–Kutta method, with details depending on the choice of quadrature for the approximation of the integral of $f(U,\cdot)$.

1.3. **Multi-adaptive time-stepping.** Standard methods for the discretization of (1.1), including the cG(q) and dG(q) methods, require that the same time steps $\{k_j\}_{j=1}^M$ are used for all components $U_i = U_i(t)$ of the approximate solution U of (1.1). This can be very costly if the system exhibits multiple time scales of different magnitudes. If the different time scales are localized to different components, efficient representation and computation of the solution thus requires that this difference in time scales is reflected in the choice of approximating spaces $(V,\hat{V})$. We refer to the resulting methods, recently introduced in a series of papers [20, 21, 22, 23, 16], as *multi-adaptive Galerkin methods*.

Surprisingly, individual time-stepping (multi-adaptivity) has previously received little attention in the large literature on numerical methods for ODEs, see e.g. [3, 12, 13, 2, 27, 1], but has been used to some extent for specific applications, including specialized integrators for the n-body problem [24, 4, 26], and low-order methods for conservation laws [25, 18, 5].

1.4. **Obtaining the software.** The examples presented below have been obtained using **DOLFIN** version 0.4.11 [14]. **DOLFIN** is licensed under the GNU General Public License [11], which means that anyone is free to use or modify the software, provided these rights are preserved. The source code of **DOLFIN**, including numerous example programs, is available at the **DOLFIN** web page, `http://www.phi.chalmers.se/dolfin/`, and each new release is announced on freshmeat.net. Alternatively, the source code can be obtained through anonymous CVS as explained on the web page. Comments and contributions are welcome.

The mechanical systems presented in the examples have been implemented using *Ko*, which is a software system for the simulation of mass–spring models, based on **DOLFIN**s multi-adaptive ODE-solver. Ko will be released shortly under the GNU General Public License and will be available at `http://www.phi.chalmers.se/ko/`.

1.5. **Outline of the paper.** We first describe the basic mass–spring model in Section 2 and then give a short introduction to multi-adaptive Galerkin methods in Section 3. In Section 4, we discuss the interface of the multi-adaptive solver and its application to mass–spring models. In Section 5, we investigate and analyze the performance of the multi-adaptive methods for a set of model problems. Finally, we present in Section 6 results for a number of large mechanical systems to demonstrate the potential and applicability of the proposed methods.

2. Mass–spring model

We have earlier in [17] described an extended mass–spring model for the simulation of systems of deformable bodies.

The mass–spring model represents bodies as systems of discrete *mass elements*, with the forces between the mass elements transmitted using explicit *spring connections*. (Note that "spring" is a historical term, and is not limited to pure Hookean interactions.) Given the forces acting on an element, we can determine its motion from Newton's second law,

$$F = ma, \tag{2.1}$$

where F denotes the force acting on the element, m is the mass of the element, and $a = \ddot{x}$ is the acceleration of the element with $x = (x_1, x_2, x_3)$ the position of the element. The motion of the entire body is then implicitly described by the motion of its individual mass elements.

The force given by a standard spring is assumed to be proportional to the elongation of the spring from its rest length. We extend the standard model with contact, collision and fracture, by adding a radius of interaction to each mass element, and dynamically creating and destroying spring connections based on contact and fracture conditions.

In Table 1 and Figure 1, we give the basic properties of the mass–spring model consisting of mass elements and spring connections. With these definitions, a mass–spring model may thus be given by just listing the mass elements and spring connections of the model.

A *mass element* e is a set of parameters $\{x,\ v,\ m,\ r,\ C\}$:

x	:	position
v	:	velocity
m	:	mass
r	:	radius
C	:	a set of spring connections

A *spring connection* c is a set of parameters $\{e_1,\ e_2,\ \kappa,\ b,\ l_0,\ l_f\}$:

e_1	:	the first mass element connected to the spring
e_2	:	the second mass element connected to the spring
κ	:	Hooke spring constant
b	:	damping constant
l_0	:	rest length
l_f	:	fracture length

TABLE 1. Descriptions of the basic elements of the mass–spring model: mass elements and spring connections.

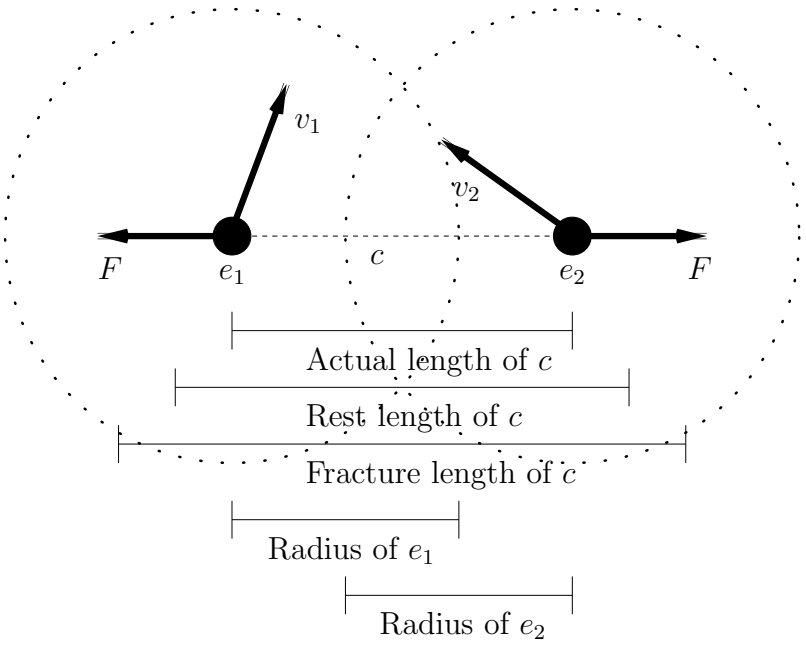


FIGURE 1. Schema of two mass elements e_1 and e_2, a spring connection c, and important quantities.

3. Multi-adaptive Galerkin methods

The multi-adaptive methods mcG(q) and mdG(q) used for the simulation are obtained as extensions of the standard cG(q) and dG(q) methods by enriching the trial and test

spaces $(V, \hat{V})$ of (1.3) to allow each component U_i of the approximate solution U to be piecewise polynomial on an individual partition of $[0, T]$.

3.1. **Definition of the methods.** To give the definition of the multi-adaptive Galerkin methods, we introduce the following notation: Subinterval j for component i is denoted by $I_{ij} = (t_{i,j-1}, t_{ij}]$, and the length of the subinterval is given by the local *time step* $k_{ij} = t_{ij} - t_{i,j-1}$ for $j = 1, \ldots, M_i$. This is illustrated in Figure 2. We also assume that the interval $[0, T]$ is partitioned into blocks between certain synchronized time levels $0 = T_0 < T_1 < \cdots < T_M = T$. We refer to the set of intervals $\mathcal{T}_n$ between two synchronized time levels T_{n-1} and T_n as a *time slab*.

With this notation, we can write the mcG(q) method for (1.1) in the following form: Find $U \in V$, such that

$$\int_0^T (\dot{U}, v)\, dt = \int_0^T (f, v)\, dt \quad \forall v \in \hat{V}, \tag{3.1}$$

where the trial space V and test space $\hat{V}$ are given by

$$\begin{aligned} V &= \{v \in [\mathcal{C}([0, T])]^N : v_i|_{I_{ij}} \in \mathcal{P}^{q_{ij}}(I_{ij}), \quad j = 1, \ldots, M_i, \quad i = 1, \ldots, N\}, \\ \hat{V} &= \{v : v_i|_{I_{ij}} \in \mathcal{P}^{q_{ij}-1}(I_{ij}), \quad j = 1, \ldots, M_i, \quad i = 1, \ldots, N\}. \end{aligned} \tag{3.2}$$

The mcG(q) method is thus obtained as a simple extension of the standard cG(q) method by allowing each component to be piecewise polynomial on an individual partition of $[0, T]$. Similarly, we obtain the mdG(q) method as a simple extension of the standard dG(q) method. For a detailed description of the multi-adaptive Galerkin methods, we refer the reader to [20, 21, 22, 23, 16]. In particular, we refer to [20] or [22] for the full definition of the methods.

3.2. **Adaptivity.** The individual time steps $\{k_{ij}\}_{j=1,i=1}^{M_i,N}$ are chosen adaptively based on an a posteriori error estimate for the global error $e = U - u$ at final time $t = T$, as discussed in [20, 21]. The a posteriori error estimate for the mcG(q) method takes the form

$$\|e(T)\|_{l_2} \leq C_q \sum_{i=1}^{N} S_i^{[q_i]}(T) \max_{[0,T]} |k_i^{q_i} R_i(U, \cdot)|, \tag{3.3}$$

where C_q is an interpolation constant, $S_i^{[q_i]}(T)$ are the individual *stability factors*, $k_i = k_i(t)$ are the individual time steps, and $R_i(U, \cdot) = \dot{U}_i - f_i(U, \cdot)$ are the individual *residuals* for $i = 1, \ldots, N$. The individual stability factors $S_i^{[q_i]}(T)$, measuring the effect of local errors introduced by a nonzero local residual on the global error, are obtained from the solution ϕ of an associated *dual problem*, see [7] or [20].

Thus, to determine the individual time steps, we measure the individual residuals and take each individual time step k_{ij} such that

$$k_{ij}^{q_{ij}} \max_{I_{ij}} |R_i(U, \cdot)| = \mathrm{TOL}/(N C_q S_i^{[q_i]}(T)), \tag{3.4}$$

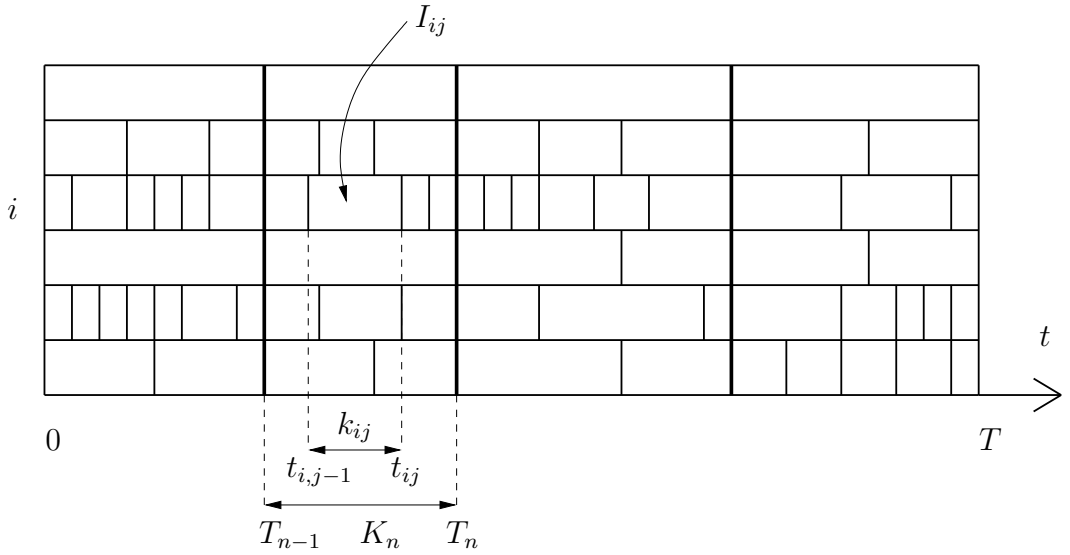


FIGURE 2. Individual partitions of the interval $[0, T]$ for different components. Elements between common synchronized time levels are organized in time slabs. In this example, we have $N = 6$ and $M = 4$.

where TOL is a tolerance for the error at final time. See [21] or [15] for a detailed discussion of the algorithm for the automatic selection of the individual time steps.

3.3. **Iterative methods.** The system of discrete nonlinear equations defined by (3.1) is solved by fixed point iteration on time slabs, as described in [16]. For a stiff problem, the fixed point iteration is automatically stabilized by introducing a damping parameter which is adaptively determined through feed-back from the computation. We refer to this as *adaptive fixed point iteration*.

4. Multi-adaptive simulation of mass–spring systems

The simulation of a mechanical system involves the formulation of a differential equation (modeling) and the solution of the differential equation (computing). Having defined these two components in the form of the mass–spring model presented in Section 2 and the multi-adaptive solver presented in Section 3, we comment briefly on the user interface of the multi-adaptive solver.

The user interface of the multi-adaptive solver is specified in terms of an ODE base class consisting of a right hand side f, a time interval $[0, T]$, and an initial value u_0, as shown in Table 2. To solve an ODE, the user implements a subclass which inherits from the ODE base class.

The mass–spring model presented above has been implemented using Ko, a software system for the simulation and visualization of mass–spring models. Ko automatically generates a mass–spring model from a geometric representation of a given system, as shown in Figure 3. The mass–spring model is then automatically translated into a system

```
class ODE
{
  ODE(int N);

  virtual real u0(int i);
  virtual real f(Vector u, real t, int i);
}
```

TABLE 2. Sketch of the C++ interface of the multi-adaptive ODE-solver.

of ODEs of the form (1.1). Ko specifies the ODE system as an ODE subclass and uses **DOLFIN** to compute the solution.

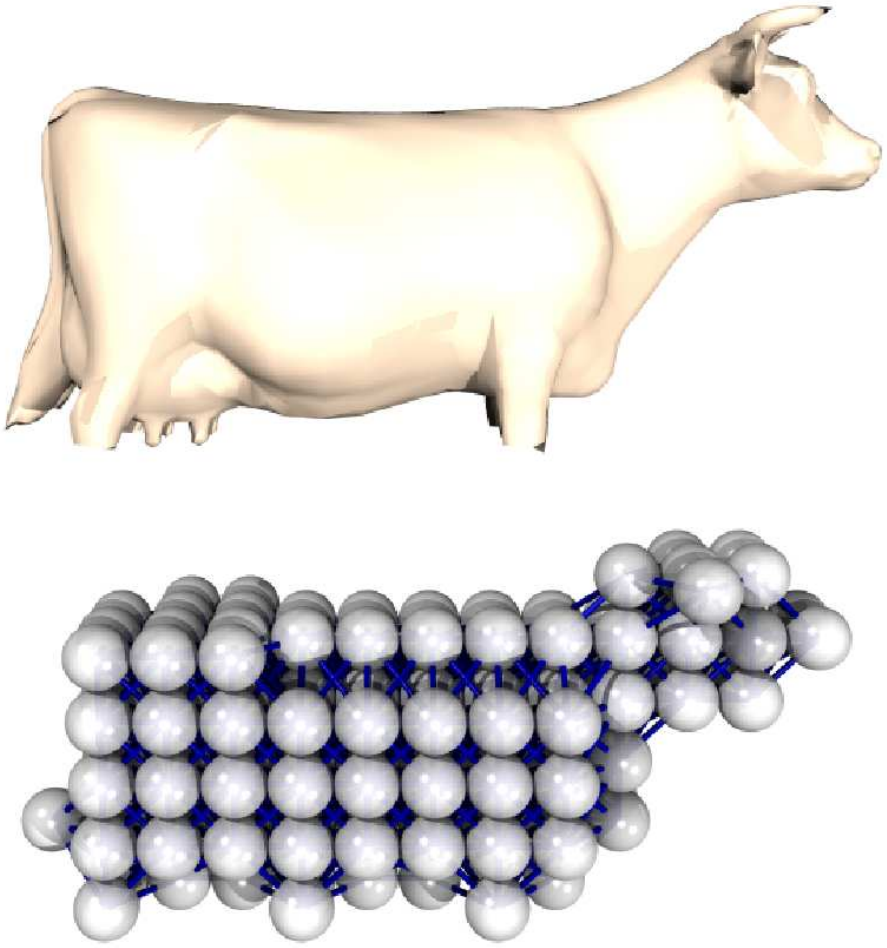


FIGURE 3. A geometric representation of a cow is automatically translated into a mass–spring model.

Ko represents a mass–spring model internally as lists of mass elements and spring connections. To evaluate the right-hand side f of the corresponding ODE system, a translation or mapping is thus needed between a given mass element and a component number in the system of ODEs. This mapping may take a number different forms; Ko uses the mapping presented in Algorithm 1.

Algorithm 1 FromComponents(Vector u, Mass m)

$i \leftarrow \text{index}(m)$
$N \leftarrow \text{size}(u)$

$m.x_1 \leftarrow u(3(i-1)+1)$
$m.x_2 \leftarrow u(3(i-1)+2)$
$m.x_3 \leftarrow u(3(i-1)+3)$

$m.v_1 \leftarrow u(N/2+3(i-1)+1)$
$m.v_2 \leftarrow u(N/2+3(i-1)+2)$
$m.v_3 \leftarrow u(N/2+3(i-1)+3)$

5. Performance

We consider a simple model problem consisting of a long string of n point masses connected with springs as shown in Figure 4. The first mass on the left is connected to a fixed support through a hard spring with large spring constant $\kappa \gg 1$. All other springs are connected together with soft springs with spring constant $\kappa = 1$. As a result, the first mass oscillates at a high frequency, with the rest of the masses oscillating slowly. In Figure 5, we plot the coordinates for the first three masses on $[0, 1]$.

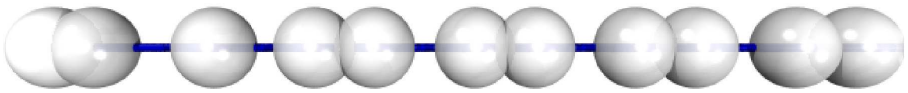


Figure 4. The mechanical system used for the performance test. The system consists of a string of masses, fixed at the left end. Each mass has been slightly displaced to initialize the oscillations.

To compare the performance of the multi-adaptive solver (in the case of the mcG(1) method) with a mono-adaptive method (the cG(1) method), we choose a fixed small time step k for the first mass and a fixed large time step $K > k$ for the rest of the masses in the multi-adaptive simulation, and use the same small time step k for all masses in the mono-adaptive simulation. We let $M = K/k$ denote the number of small time steps per each large time step.

We run the test for $M = 100$ and $M = 200$ with large spring constant $\kappa = 10M$ for the hard spring connecting the first mass to the fixed support. We use a large time step of size

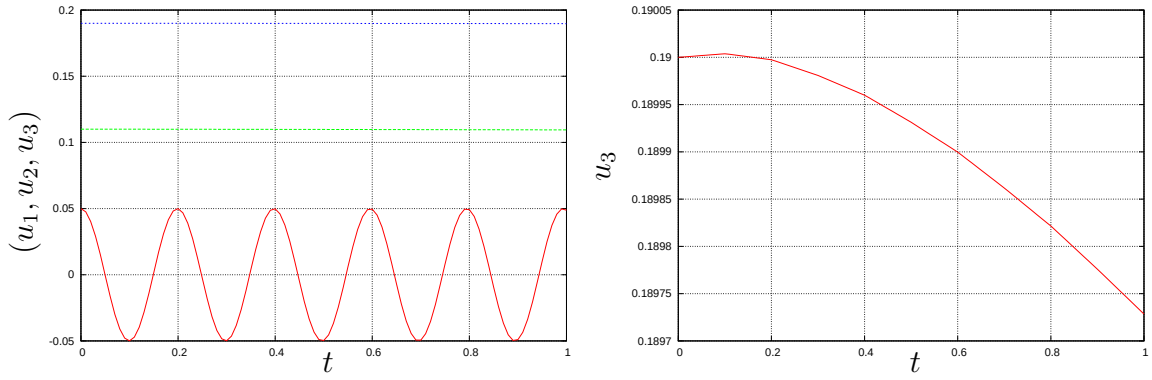


FIGURE 5. Coordinates for the first three masses of the simple model problem (left) and for the third mass (right).

$K = 0.1$ and, consequently, a small time step of size $k = 0.1/M$. The computation time T_c is recorded as function of the number of masses n.

As shown in Figure 6, the computation time for the multi-adaptive solver grows slowly with the number of masses n, practically remaining constant; small time steps are used only for the first rapidly oscillating mass and so the work is dominated by frequently updating the first mass, independent of the total number of masses. On the other hand, the work for the mono-adaptive method grows linearly with the total number of masses, as expected.

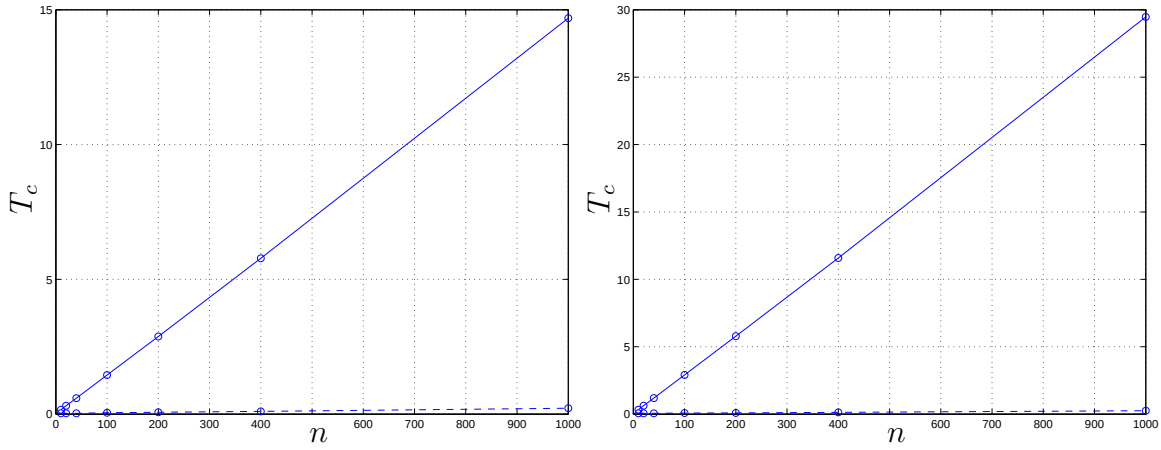


FIGURE 6. Computation time T_c as function of the number of masses n for the multi-adaptive solver (dashed) and a mono-adaptive method (solid), with $M = 100$ (left) and $M = 200$ (right).

More specifically, the complexity of the mono-adaptive method may be expressed in terms of M and n as follows:

$$T_c(M, n) = C_1 + C_2 M n, \tag{5.1}$$

while for the multi-adaptive solver, we obtain

$$T_c(M, n) = C_3 M + C_4 n. \tag{5.2}$$

Our general conclusion is that the multi-adaptive solver is more efficient than a mono-adaptive method for the simulation of a mechanical system if M is large, i.e., when small time steps are needed for a part of the system, and if n is large, i.e, if large time steps may be used for a large part of the system.

The same result is obtained if we add damping to the system in the form of a damping constant of size $b = 100$ for the spring connections between the slowly oscillating masses, resulting in gradual damping of the slow oscillations, while keeping the rapid oscillations of the first mass largely unaffected. With $b = 100$, adaptive fixed point iteration is automatically activated for the solution of the discrete equations, as discussed in Section 3.3.

6. Large problems and applications

To demonstrate the potential and applicability of the proposed mass–spring model and the multi-adaptive solver, we present results for a number of large mechanical systems.

6.1. **Oscillating tail.** For the first example, we take the mass–spring model of Figure 3 representing a heavy cow and add a light mass representing its tail, as shown in Figure 7. The cow is given a constant initial velocity and the tail is given an initial push to make it oscillate. A sequence of frames from an animation of the multi-adaptive solution is given in Figure 8.

We compare the performance of the multi-adaptive solver (in the case of the mcG(1) method) with a mono-adaptive method (the cG(1) method) using the same time steps for all components. We also make a comparison with a simple non-adaptive implementation of the cG(1) method, with minimal overhead, using constant time steps equal to the smallest time step selected by the mono-adaptive method.

As expected, the multi-adaptive solver automatically selects small time steps for the oscillating tail and large time steps for the rest of the system. In Figure 9, we plot the time steps as function of time for relevant components of the system. We also plot the corresponding solutions in Figure 11. In Figure 10, we plot the time steps used in the mono-adaptive simulation.

The computation times are given in Table 3. The speed-up of the multi-adaptive method compared to the mono-adaptive method is a factor 70. Compared to the simple non-adaptive implementation of the cG(1) method, using a minimal amount of work, the speed-up is a factor 3. This shows that the speed-up of a multi-adaptive method can be significant. It also shows that the overhead is substantial for the current implementation of the multi-adaptive solver, including the organization of the multi-adaptive time slabs, interpolation of solution values within time slabs, and the evaluation of residuals for multi-adaptive time-stepping. However, we believe it is possible to remove a large part of this overhead.

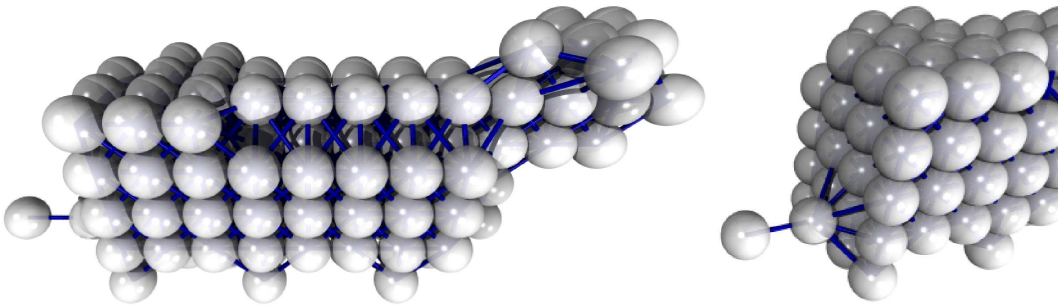


FIGURE 7. A cow with an oscillating tail (left) with details of the tail (right).

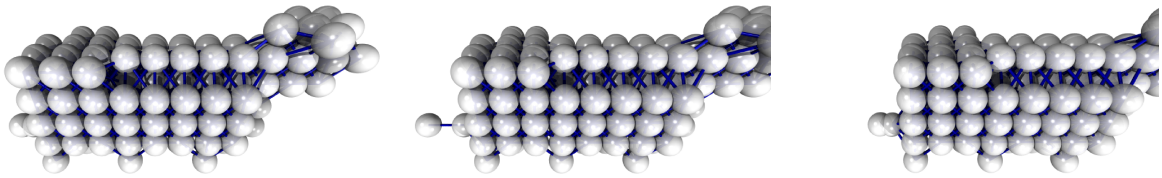


FIGURE 8. The tail oscillates rapidly while the rest of the cow travels at a constant velocity to the right.

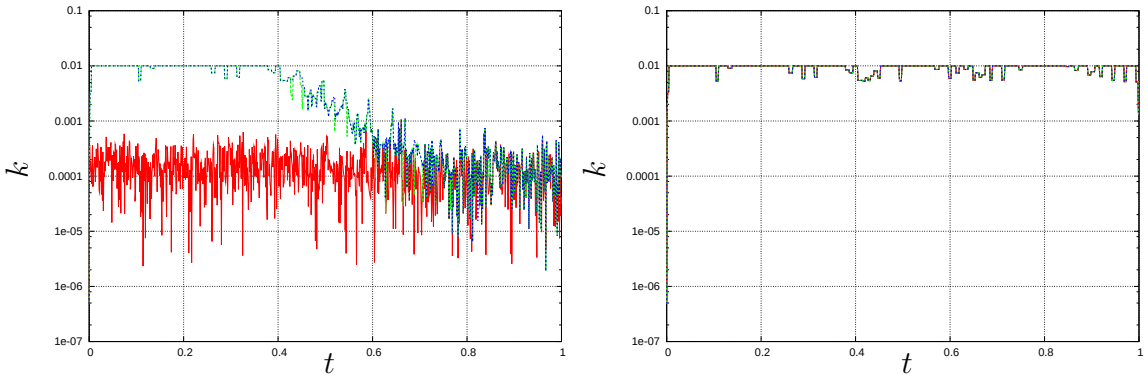


FIGURE 9. Multi-adaptive time steps used in the simulation of the cow with oscillating tail. The plot on the left shows the time steps for components 481–483 corresponding to the velocity of the tail, and the plot on the right shows the time steps for components 13–24 corresponding to the positions for a set of masses in the interior of the cow.

6.2. **Local manipulation.** For the next example, we fix a damped cow shape at one end and repeatedly push the other end with a manipulator in the form of a large sphere, as illustrated in Figure 12.

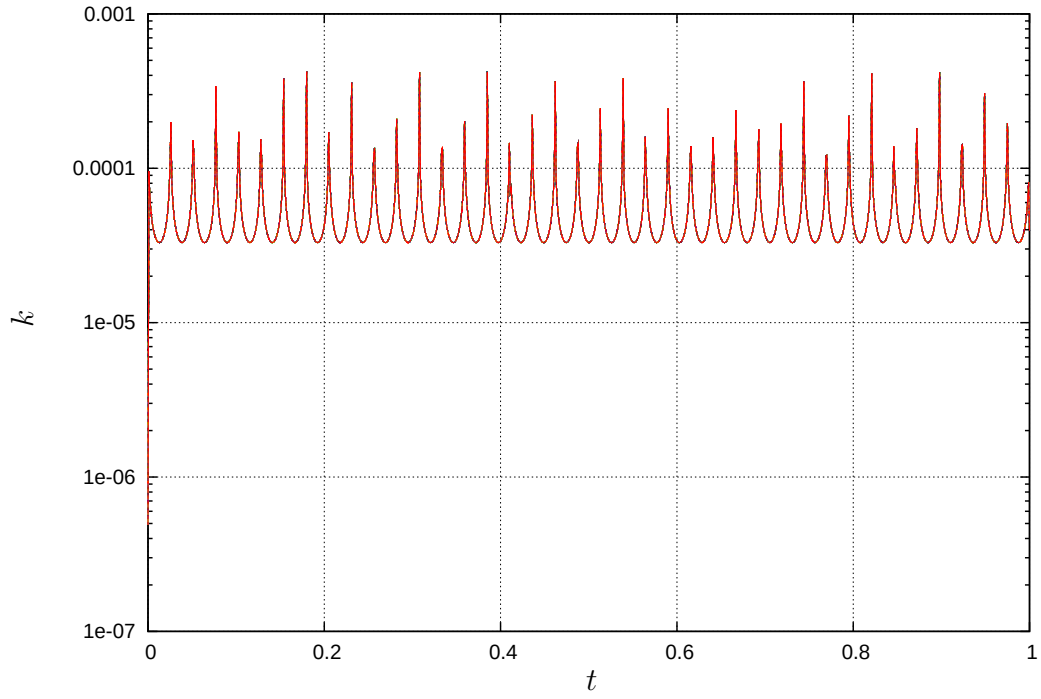


FIGURE 10. Mono-adaptive time steps for the cow with oscillating tail.

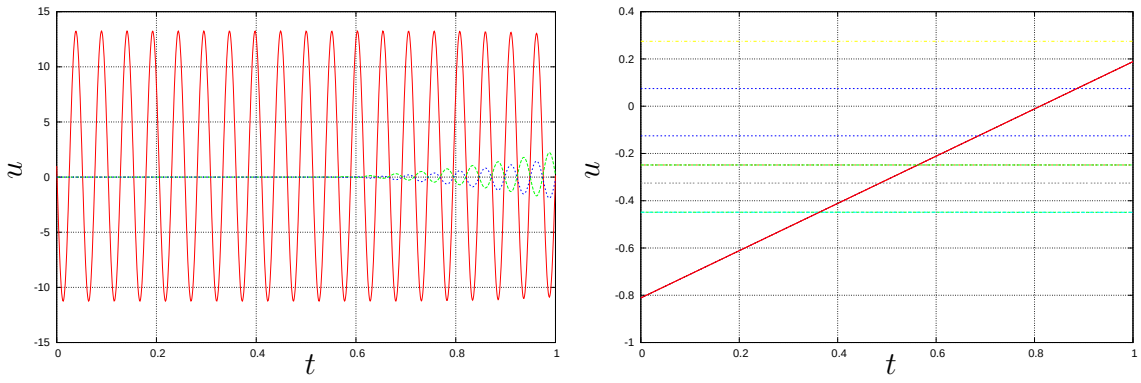


FIGURE 11. Solution for relevant components of the cow with oscillating tail. The plot on the left shows the solution for components 481–483 corresponding to the velocity of the tail, and the plot on the right shows the solution for components 13–24 corresponding to the positions for a set of masses in the interior of the cow.

Algorithm	Time / s
Multi-adaptive	40
Mono-adaptive	2800
Non-adaptive	130

TABLE 3. Computation times for the simulation of the cow with oscillating tail for three different algorithms: multi-adaptive mcG(1), mono-adaptive cG(1), and a simple implementation of non-adaptive cG(1) with fixed time steps and minimal overhead.

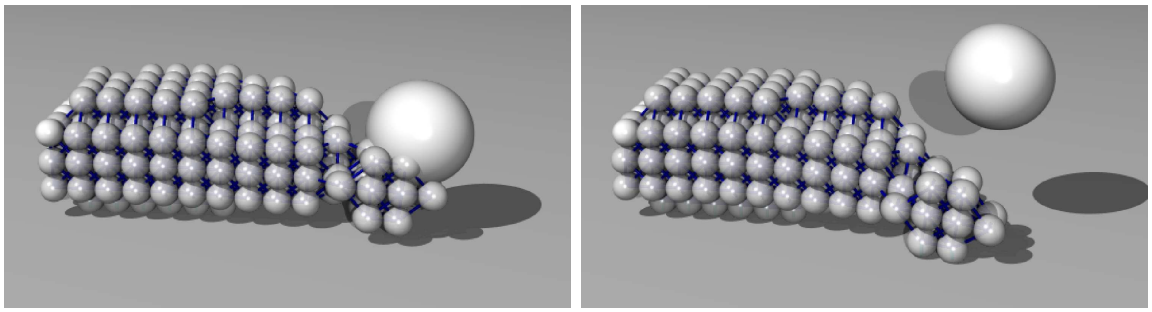


FIGURE 12. A cow shape is locally manipulated. Small time steps are automatically selected for the components affected by the local manipulation, with large time steps for the rest of the system.

As shown in Figure 13, the multi-adaptive solver automatically selects small time steps for the components directly affected by the manipulation. This test problem also illustrates the basic use of adaptive time-stepping; the time steps are drastically decreased at each impact to accurately track the effect of the impact.

6.3. **A stiff beam.** Our final example demonstrates the applicability of the multi-adaptive solver to a stiff problem consisting of a block being dropped onto a stiff beam, as shown in Figure 14. The material of both the block and the beam is very hard and very damped, with spring constant $\kappa = 10^7$ and damping constant $b = 2 \cdot 10^5$ for each spring connection. The multi-adaptive time steps for the simulation are shown in Figure 15. Note that the time steps are drastically reduced at the time of impact, with large time steps before and after the impact.

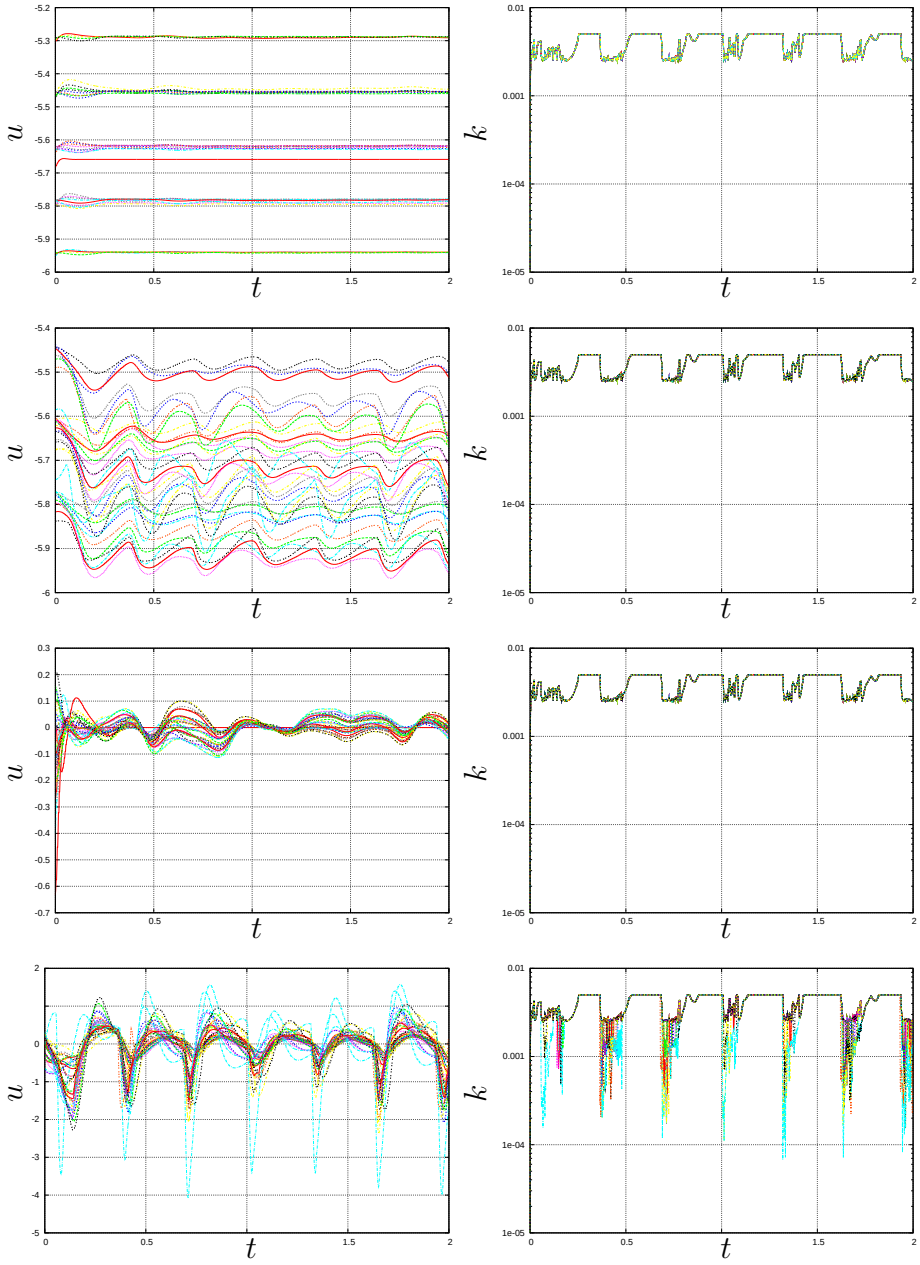


FIGURE 13. Solution (left) and multi-adaptive time steps (right) for selected components of the manipulated cow. The two top rows correspond to the positions of the left- and right-most masses, respectively, and the two rows below correspond to the velocities of the left- and right-most masses, respectively. Note that smaller time steps are used for the components mostly affected by the manipulation, in particular at the point of impact, while larger time steps are used for other components.

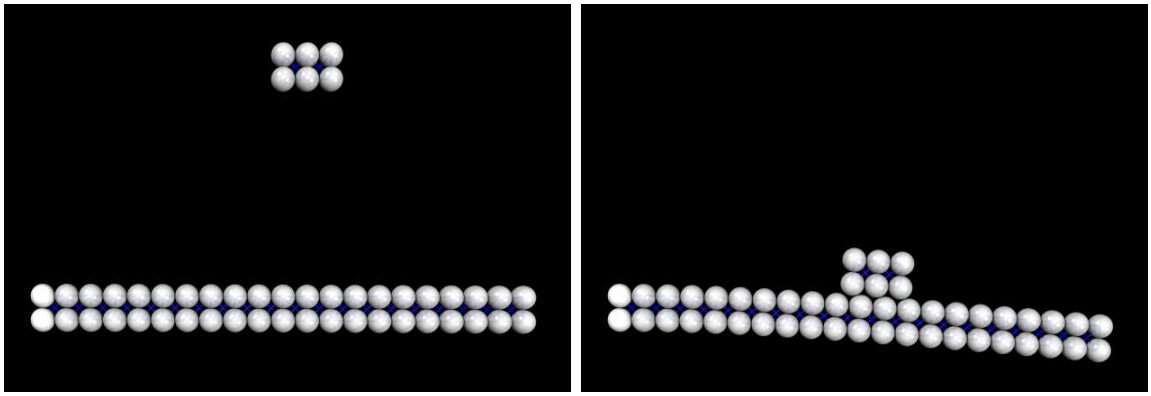


FIGURE 14. A block is dropped onto a beam. The material of both the block and the beam is very hard and very damped, with spring constant $\kappa = 10^7$ and damping constant $b = 2 \cdot 10^5$.

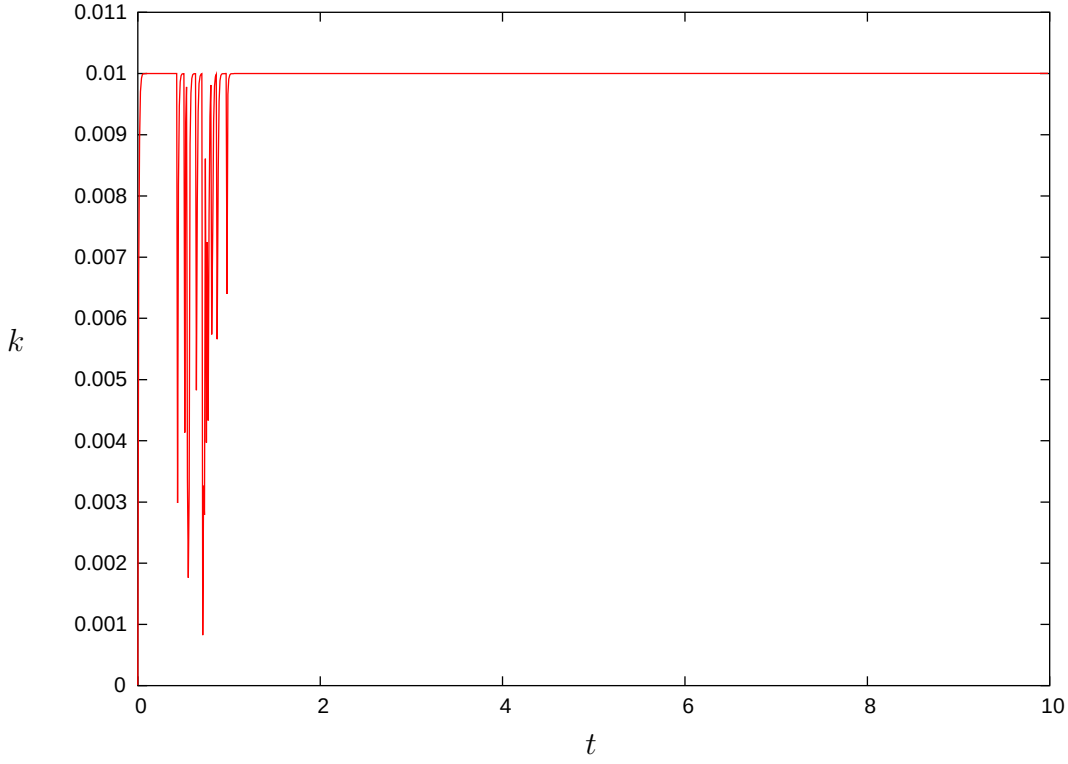


FIGURE 15. Multi-adaptive time steps for the block and beam. Note that the time steps are drastically reduced at the time of impact. The maximum time step is set to 0.01 to track the contact between the block and the beam.

7. Conclusions

From the results presented above, we make the following conclusions regarding multi-adaptive time-stepping:

- A multi-adaptive method outperforms a mono-adaptive method for systems containing different time scales if there is a significant separation of the time scales and if the fast time scales are localized to a relatively small part of the system.
- Multi-adaptive time-stepping, and in particular the current implementation, works in practice for large and realistic problems.

References

[1] U. Ascher and L. Petzold, *Computer Methods for Ordinary Differential Equations and Differential-Algebraic Equations*, SIAM, 1998.

[2] J. Butcher, *The Numerical Analysis of Ordinary Differential Equations — Runge–Kutta and General Linear Methods*, Wiley, 1987.

[3] G. Dahlquist, *Stability and Error Bounds in the Numerical Integration of Ordinary Differential Equations*, PhD thesis, Stockholm University, 1958.

[4] R. Davé, J. Dubinski, and L. Hernquist, *Parallel treeSPH*, New Astronomy, 2 (1997), pp. 277–297.

[5] C. Dawson and R. Kirby, *High resolution schemes for conservation laws with locally varying time steps*, SIAM J. Sci. Comput., 22, No. 6 (2001), pp. 2256–2281.

[6] M. Delfour, W. Hager, and F. Trochu, *Discontinuous Galerkin methods for ordinary differential equations*, Math. Comp., 36 (1981), pp. 455–473.

[7] K. Eriksson, D. Estep, P. Hansbo, and C. Johnson, *Introduction to adaptive methods for differential equations*, Acta Numerica, (1995), pp. 105–158.

[8] ——, *Computational Differential Equations*, Cambridge University Press, 1996.

[9] D. Estep, *A posteriori error bounds and global error control for approximations of ordinary differential equations*, SIAM J. Numer. Anal., 32 (1995), pp. 1–48.

[10] D. Estep and D. French, *Global error control for the continuous Galerkin finite element method for ordinary differential equations*, M^2AN, 28 (1994), pp. 815–852.

[11] Free Software Foundation, *GNU GPL*, http://www.gnu.org/copyleft/gpl.html.

[12] E. Hairer and G. Wanner, *Solving Ordinary Differential Equations I — Nonstiff Problems*, Springer Series in Computational Mathematics, vol 8, 1991.

[13] ——, *Solving Ordinary Differential Equations II — Stiff and Differential-Algebraic Problems*, Springer Series in Computational Mathematics, vol 14, 1991.

[14] J. Hoffman and A. Logg et al., *DOLFIN*, http://www.phi.chalmers.se/dolfin/.

[15] J. Jansson and A. Logg, *Algorithms for multi-adaptive time-stepping*, submitted to ACM Trans. Math. Softw., (2004).

[16] ——, *Multi-adaptive Galerkin methods for ODEs V: Stiff problems*, submitted to BIT, (2004).

[17] J. Jansson and J. Vergeest, *A discrete mechanics model for deformable bodies*, Computer-Aided Design, 34 (2002).

[18] J.E. Flaherty, R.M. Loy, M.S. Shephard, B.K. Szymanski, J.D. Teresco, and L.H. Ziantz, *Adaptive local refinement with octree load balancing for the parallel solution of three-dimensional conservation laws*, Journal of Parallel and Distributed Computing, 47 (1997), pp. 139–152.

[19] C. Johnson, *Error estimates and adaptive time-step control for a class of one-step methods for stiff ordinary differential equations*, SIAM J. Numer. Anal., 25 (1988), pp. 908–926.

[20] A. Logg, *Multi-adaptive Galerkin methods for ODEs I*, SIAM J. Sci. Comput., 24 (2003), pp. 1879–1902.

[21] ——, *Multi-adaptive Galerkin methods for ODEs II: Implementation and applications*, SIAM J. Sci. Comput., 25 (2003), pp. 1119–1141.
[22] ——, *Multi-adaptive Galerkin methods for ODEs III: Existence and stability*, Submitted to SIAM J. Numer. Anal., (2004).
[23] ——, *Multi-adaptive Galerkin methods for ODEs IV: A priori error estimates*, Submitted to SIAM J. Numer. Anal., (2004).
[24] J. Makino and S. Aarseth, *On a Hermite integrator with Ahmad-Cohen scheme for gravitational many-body problems*, Publ. Astron. Soc. Japan, 44 (1992), pp. 141–151.
[25] S. Osher and R. Sanders, *Numerical approximations to nonlinear conservation laws with locally varying time and space grids*, Math. Comp., 41 (1983), pp. 321–336.
[26] S.G. Alexander and C.B. Agnor, *n-body simulations of late stage planetary formation with a simple fragmentation model*, ICARUS, 132 (1998), pp. 113–124.
[27] L. Shampine, *Numerical Solution of Ordinary Differential Equations*, Chapman & Hall, 1994.

Mathematical Models and Methods in Applied Sciences
Vol. 15, No. 3 (2005) 471–481

World Scientific
www.worldscientific.com

COMPUTATIONAL MODELING OF DYNAMICAL SYSTEMS

JOHAN JANSSON* and CLAES JOHNSON†

Department of Computational Mathematics, Chalmers University of Technology
SE–412 96 Göteborg, Sweden
**johanjan@math.chalmers.se*
†*claes@math.chalmers.se*

ANDERS LOGG

Toyota Technological Institute, 1427 E. 60th Street
Chicago, IL 60637, USA
logg@tti-c.org

Received 13 September 2004
Communicated by F. Brezzi

In this short note, we discuss the basic approach to computational modeling of dynamical systems. If a dynamical system contains multiple time scales, ranging from very fast to slow, computational solution of the dynamical system can be very costly. By resolving the fast time scales in a short time simulation, a model for the effect of the small time scale variation on large time scales can be determined, making solution possible on a long time interval. This process of computational modeling can be completely automated. Two examples are presented, including a simple model problem oscillating at a time scale of 10^{-9} computed over the time interval $[0, 100]$, and a lattice consisting of large and small point masses.

Keywords: Modeling; dynamical system; reduced model; automation.

1. Introduction

We consider a dynamical system of the form

$$\begin{aligned} \dot{u}(t) &= f(u(t), t), \quad t \in (0, T], \\ u(0) &= u_0, \end{aligned} \tag{1.1}$$

where u: $[0, T] \to \mathbb{R}^N$ is the solution to be computed, $u_0 \in \mathbb{R}^N$ a given initial value, $T > 0$ a given final time, and f: $\mathbb{R}^N \times (0, T] \to \mathbb{R}^N$ a given function that is Lipschitz-continuous in u and bounded. We consider a situation where the exact solution u varies on different time scales, ranging from very fast to slow. Typical examples include meteorological models for weather prediction, with fast time scales on the range of seconds and slow time scales on the range of years, protein folding represented by a molecular dynamics model of the form (1.1), with fast time scales

on the range of femtoseconds and slow time scales on the range of microseconds, or turbulent flow with a wide range of time scales.

In order to make computation feasible in a situation where computational resolution of the fast time scales would be prohibitive because of the small time steps, the given model (1.1) containing the fast time scales needs to be replaced with a *reduced model* for the variation of the solution u of (1.1) on resolvable time scales. As discussed below, the key step is to correctly model the effect of the variation at the fast time scales on the variation on slow time scales.

The problem of model reduction is very general and various approaches have been taken.[8,6] We present below a new approach to model reduction, based on resolving the fast time scales in a short time simulation and determining a model for the effect of the small time scale variation on large time scales. This process of computational modeling can be completely *automated* and the validity of the reduced model can be evaluated *a posteriori*.

2. A Simple Model Problem

We consider a simple example illustrating the basic aspects: Find $u = (u_1, u_2)$: $[0, T] \to \mathbb{R}^2$, such that

$$\begin{aligned} \ddot{u}_1 + u_1 - u_2^2/2 &= 0 \quad \text{on } (0, T], \\ \ddot{u}_2 + \kappa u_2 &= 0 \quad \text{on } (0, T], \\ u(0) = (0, 1), &\quad \dot{u}(0) = (0, 0), \end{aligned} \tag{2.1}$$

which models a moving unit point mass M_1 connected through a soft spring to another unit point mass M_2, with M_2 moving along a line perpendicular to the line of motion of M_1, see Fig. 1. The second point mass M_2 is connected to a fixed support through a very stiff spring with spring constant $\kappa = 10^{18}$ and oscillates rapidly on a time scale of size $1/\sqrt{\kappa} = 10^{-9}$. The oscillation of M_2 creates a force

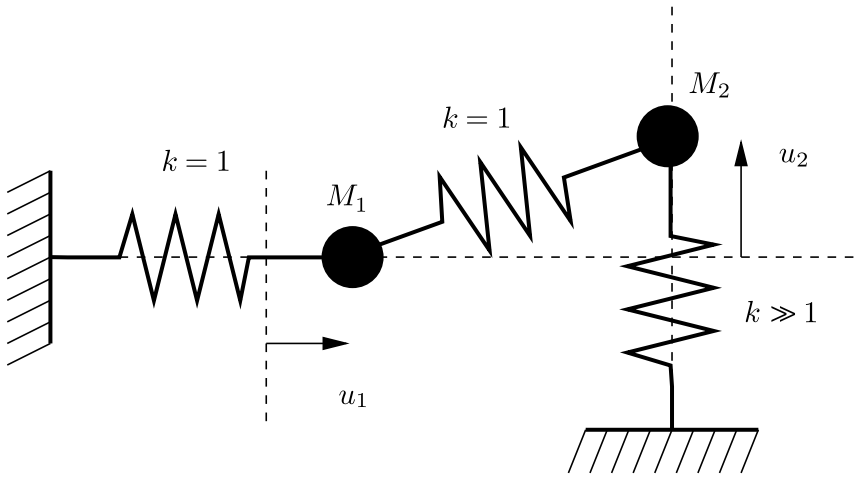


Fig. 1. A simple mechanical system with large time scale ~ 1 and small time scale $\sim 1/\sqrt{\kappa}$.

$\sim u_2^2$ on M_1 proportional to the elongation of the spring connecting M_2 to M_1 (neglecting terms of order u_2^4).

The short time scale of size 10^{-9} requires time steps of size $\sim 10^{-10}$ for full resolution. With $T = 100$, this means a total of $\sim 10^{12}$ time steps for solution of (2.1). However, by replacing (2.1) with a reduced model where the fast time scale has been removed, it is possible to compute the (averaged) solution of (2.1) with time steps of size ~ 0.1 and consequently only a total of 10^3 time steps.

3. Taking Averages to Obtain the Reduced Model

Having realized that pointwise resolution of the fast time scales of the exact solution u of (1.1) may sometimes be computationally very expensive or even impossible, we seek instead to compute a time average $\bar{u}$ of u, defined by

$$\bar{u}(t) = \frac{1}{\tau}\int_{-\tau/2}^{\tau/2} u(t+s)\,ds, \quad t \in [\tau/2, T-\tau/2], \tag{3.1}$$

where $\tau > 0$ is the size of the average. The average $\bar{u}$ can be extended to $[0,T]$ in various ways. We consider here a constant extension, i.e. we let $\bar{u}(t) = \bar{u}(\tau/2)$ for $t \in [0,\tau/2)$, and let $\bar{u}(t) = \bar{u}(T-\tau/2)$ for $t \in (T-\tau/2, T]$.

We now seek a dynamical system satisfied by the average $\bar{u}$ by taking the average of (1.1). We obtain

$$\dot{\bar{u}}(t) = \bar{\dot{u}}(t) = \overline{f(u,\cdot)}(t) = f(\bar{u}(t),t) + (\overline{f(u,\cdot)}(t) - f(\bar{u}(t),t)),$$

or

$$\dot{\bar{u}}(t) = f(\bar{u}(t),t) + \bar{g}(u,t), \tag{3.2}$$

where the *variance* $\bar{g}(u,t) = \overline{f(u,\cdot)}(t) - f(\bar{u}(t),t)$ accounts for the effect of small scales on time scales larger than τ. (Note that we may extend (3.2) to $(0,T]$ by defining $\bar{g}(u,t) = -f(\bar{u}(t),t)$ on $(0,\tau/2] \cup (T-\tau/2, T]$.)

We now seek to model the variance $\bar{g}(u,t)$ in the form $\bar{g}(u,t) \approx \tilde{g}(\bar{u}(t),t)$ and replace (3.2) and thus (1.1) by

$$\begin{aligned} &\dot{\tilde{u}}(t) = f(\tilde{u}(t),t) + \tilde{g}(\tilde{u}(t),t), \quad t \in (0,T], \\ &\tilde{u}(0) = \bar{u}_0, \end{aligned} \tag{3.3}$$

where $\bar{u}_0 = \bar{u}(0) = \bar{u}(\tau/2)$. We refer to this system as the *reduced model* with *subgrid model* $\tilde{g}$ corresponding to (1.1).

To summarize, if the solution u of the full dynamical system (1.1) is computationally unresolvable, we aim at computing the average $\bar{u}$ of u. However, since the variance $\bar{g}$ in the averaged dynamical system (3.2) is unknown, we need to solve the reduced model (3.3) for $\tilde{u} \approx \bar{u}$ with an approximate subgrid model $\tilde{g} \approx \bar{g}$. Solving the reduced model (3.3) using e.g. a Galerkin finite element method, we obtain an approximate solution $U \approx \tilde{u} \approx \bar{u}$. Note that we may not expect U to be close to u pointwise in time, while we hope that U is close to $\bar{u}$ pointwise.

4. Modeling the Variance

There are two basic approaches to the modeling of the variance $\bar{g}(u,t)$ in the form $\tilde{g}(\tilde{u}(t),t)$; (i) scale-extrapolation or (ii) local resolution. In (i), a sequence of solutions is computed with increasingly fine resolution, but without resolving the fastest time scales. A model for the effects of the fast unresolvable scales is then determined by extrapolation from the sequence of computed solutions.[3] In (ii), the approach followed below, the solution u is computed accurately over a short time period, resolving the fastest time scales. The reduced model is then obtained by computing the variance

$$\bar{g}(u,t) = \overline{f(u,\cdot)}(t) - f(\bar{u}(t),t) \tag{4.1}$$

and then determining $\tilde{g}$ for the remainder of the time interval such that $\tilde{g}(\tilde{u}(t),t) \approx \bar{g}(u,t)$.

For the simple model problem (2.1), which we can write in the form (1.1) by introducing the two new variables $u_3 = \dot{u}_1$ and $u_4 = \dot{u}_2$ with

$$f(u,\cdot) = (u_3, u_4, -u_1 + u_2^2/2, -\kappa u_2),$$

we note that $\bar{u}_2 \approx 0$ (for $\sqrt{\kappa}\tau$ large) while $\overline{u_2^2} \approx 1/2$. By the linearity of f_1, f_2 and f_4, the (approximate) reduced model takes the form

$$\begin{aligned} \ddot{\tilde{u}}_1 + \tilde{u}_1 - 1/4 = 0 \quad &\text{on } (0,T], \\ \ddot{\tilde{u}}_2 + \kappa\tilde{u}_2 = 0 \quad &\text{on } (0,T], \\ \tilde{u}(0) = (0,0), \quad \dot{\tilde{u}}(0) &= (0,0), \end{aligned} \tag{4.2}$$

with solution $\tilde{u}(t) = \big(\frac{1}{4}(1-\cos t), 0\big)$.

In general, the reduced model is constructed with subgrid model $\tilde{g}$ varying on resolvable time scales. In the simplest case, it is enough to model $\tilde{g}$ with a constant and repeatedly checking the validity of the model by comparing the reduced model (3.3) with the full model (1.1) in a short time simulation. Another possibility is to use a piecewise polynomial representation for the subgrid model $\tilde{g}$.

5. Solving the Reduced System

Although the presence of small scales has been decreased in the reduced system (3.3), the small scale variation may still be present. This is not evident in the reduced system (4.2) for the simple model problem (2.1), where we made the approximation $\tilde{u}_2(0) = 0$. In practice, however, we compute $\tilde{u}_2(0) = \frac{1}{\tau}\int_0^\tau u_2(t)\,dt = \frac{1}{\tau}\int_0^\tau \cos(\sqrt{\kappa}t)\,dt \sim 1/(\sqrt{\kappa}\tau)$ and so $\tilde{u}_2$ oscillates at the fast time scale $1/\sqrt{\kappa}$ with amplitude $1/(\sqrt{\kappa}\tau)$.

To remove these oscillations, the reduced system needs to be stabilized by introducing damping of high frequencies. Following the general approach,[5] a least squares stabilization is added in the Galerkin formulation of the reduced system (3.3) in the form of a modified test function. As a result, damping is introduced for high frequencies without affecting low frequencies.

Alternatively, components such as u_2 in (4.2) may be *inactivated*, corresponding to a subgrid model of the form $\tilde{g}_2(\tilde{u},\cdot) = -f_2(\tilde{u},\cdot)$. We take this simple approach for the examples presented below.

6. Error Analysis

The validity of a proposed subgrid model may be checked *a posteriori*. To analyze the modeling error introduced by approximating the variance $\bar{g}$ with the subgrid model $\tilde{g}$, we introduce the *dual problem*

$$\begin{aligned} -\dot{\phi}(t) &= J(\bar{u},U,t)^\top \phi(t), \quad t \in [0,T), \\ \phi(T) &= \psi, \end{aligned} \tag{6.1}$$

where J denotes the Jacobian of the right-hand side of the dynamical system (1.1) evaluated at a mean value of the average $\bar{u}$ and the computed numerical (finite element) solution $U \approx \tilde{u}$ of the reduced system (3.3),

$$J(\bar{u},U,t) = \int_0^1 \frac{\partial f}{\partial u}(s\bar{u}(t) + (1-s)U(t),t)\,ds, \tag{6.2}$$

and where ψ is the initial data for the backward dual problem.

To estimate the error $\bar{e} = U - \bar{u}$ at final time, we note that $\bar{e}(0) = 0$ and $\dot{\phi} + J(\bar{u},U,\cdot)^\top \phi = 0$, and write

$$\begin{aligned} (\bar{e}(T),\psi) &= (\bar{e}(T),\psi) - \int_0^T (\dot{\phi} + J(\bar{u},U,\cdot)^\top \phi, \bar{e})\,dt \\ &= \int_0^T (\phi, \dot{\bar{e}} - J\bar{e})\,dt = \int_0^T (\phi, \dot{U} - \dot{\bar{u}} - f(U,\cdot) + f(\bar{u},\cdot))\,dt \\ &= \int_0^T (\phi, \dot{U} - f(U,\cdot) - \tilde{g}(U,\cdot))\,dt + \int_0^T (\phi, \tilde{g}(U,\cdot) - \bar{g}(u,\cdot))\,dt \\ &= \int_0^T (\phi, \tilde{R}(U,\cdot))\,dt + \int_0^T (\phi, \tilde{g}(U,\cdot) - \bar{g}(u,\cdot))\,dt. \end{aligned}$$

The first term, $\int_0^T (\phi, \tilde{R}(U,\cdot))\,dt$, in this *error representation* corresponds to the *discretization error* $U - \tilde{u}$ for the numerical solution of (3.3). If a Galerkin finite element method is used,[1,2] the *Galerkin orthogonality* expressing the orthogonality of the residual $\tilde{R}(U,\cdot) = \dot{U} - f(U,\cdot) - \tilde{g}(U,\cdot)$ to a space of test functions can be used to subtract a test space interpolant $\pi\phi$ of the dual solution ϕ. In the simplest case of the cG(1) method for a partition of the interval $(0,T]$ into M subintervals $I_j = (t_{j-1},t_j]$, each of length $k_j = t_j - t_{j-1}$, we subtract a piecewise constant interpolant to obtain

$$\begin{aligned} \int_0^T (\phi, \tilde{R}(U,\cdot))\,dt &= \int_0^T (\phi - \pi\phi, \tilde{R}(U,\cdot))\,dt \le \sum_{j=1}^M k_j \max_{I_j} \|\tilde{R}(U,\cdot)\|_{l_2} \int_{I_j} \|\dot{\phi}\|_{l_2}\,dt \\ &\le S^{[1]}(T) \max_{[0,T]} \|k\tilde{R}(U,\cdot)\|_{l_2}, \end{aligned}$$

where the *stability factor* $S^{[1]}(T) = \int_0^T \|\dot{\phi}\|_{l_2}\,dt$ measures the sensitivity to discretization errors for the given output quantity $(\bar{e}(T), \psi)$.

The second term, $\int_0^T (\phi, \tilde{g}(U,\cdot) - \bar{g}(u,\cdot))\,dt$, in the error representation corresponds to the *modeling error* $\tilde{u} - \bar{u}$. The sensitivity to modeling errors is measured by the stability factor $S^{[0]}(T) = \int_0^T \|\phi\|_{l_2}\,dt$. We notice in particular that if the stability factor $S^{[0]}(T)$ is of moderate size, a reduced model of the form (3.3) for $\tilde{u} \approx \bar{u}$ may be constructed.

We thus obtain the error estimate

$$|(\bar{e}(T), \psi)| \leq S^{[1]}(T) \max_{[0,T]} \|k\tilde{R}(U,\cdot)\|_{l_2} + S^{[0]}(T) \max_{[0,T]} \|\tilde{g}(U,\cdot) - \bar{g}(u,\cdot)\|_{l_2}, \quad (6.3)$$

including both discretization and modeling errors. The initial data ψ for the dual problem (6.1) is chosen to reflect the desired output quantity, e.g. $\psi = (1, 0, \ldots, 0)$ to measure the error in the first component of U.

To estimate the modeling error, we need to estimate the quantity $\tilde{g} - \bar{g}$. This estimate is obtained by repeatedly solving the full dynamical system (1.1) at a number of control points and comparing the subgrid model $\tilde{g}$ with the computed variance $\bar{g}$. As initial data for the full system at a control point, we take the computed solution $U \approx \bar{u}$ at the control point and add a perturbation of appropriate size, with the size of the perturbation chosen to reflect the initial oscillation at the fastest time scale.

7. Numerical Results

We present numerical results for two model problems, including the simple model problem (2.1), computed with DOLFIN[4] version 0.4.10. With the option *automatic modeling* set, DOLFIN automatically creates the reduced model (3.3) for a given dynamical system of the form (1.1) by resolving the full system in a short time simulation and then determining a constant subgrid model $\bar{g}$. Components with constant average, such as u_2 in (2.1), are automatically marked as inactive and are kept constant throughout the simulation. The automatic modeling implemented in DOLFIN is rudimentary and many improvements are possible, but it represents a first attempt at the automation of modeling, following the recently presented[7] directions for the *automation of computational mathematical modeling*.

7.1. *The simple model problem*

The solution for the two components of the simple model problem (2.1) is shown in Fig. 2 for $\kappa = 10^{18}$ and $\tau = 10^{-7}$. The value of the subgrid model $\bar{g}_1$ is automatically determined to $0.2495 \approx 1/4$.

7.2. *A lattice with internal vibrations*

The second example is a lattice consisting of a set of p^2 large and $(p-1)^2$ small point masses connected by springs of equal stiffness $\kappa = 1$, as shown in Figs. 3 and 4.

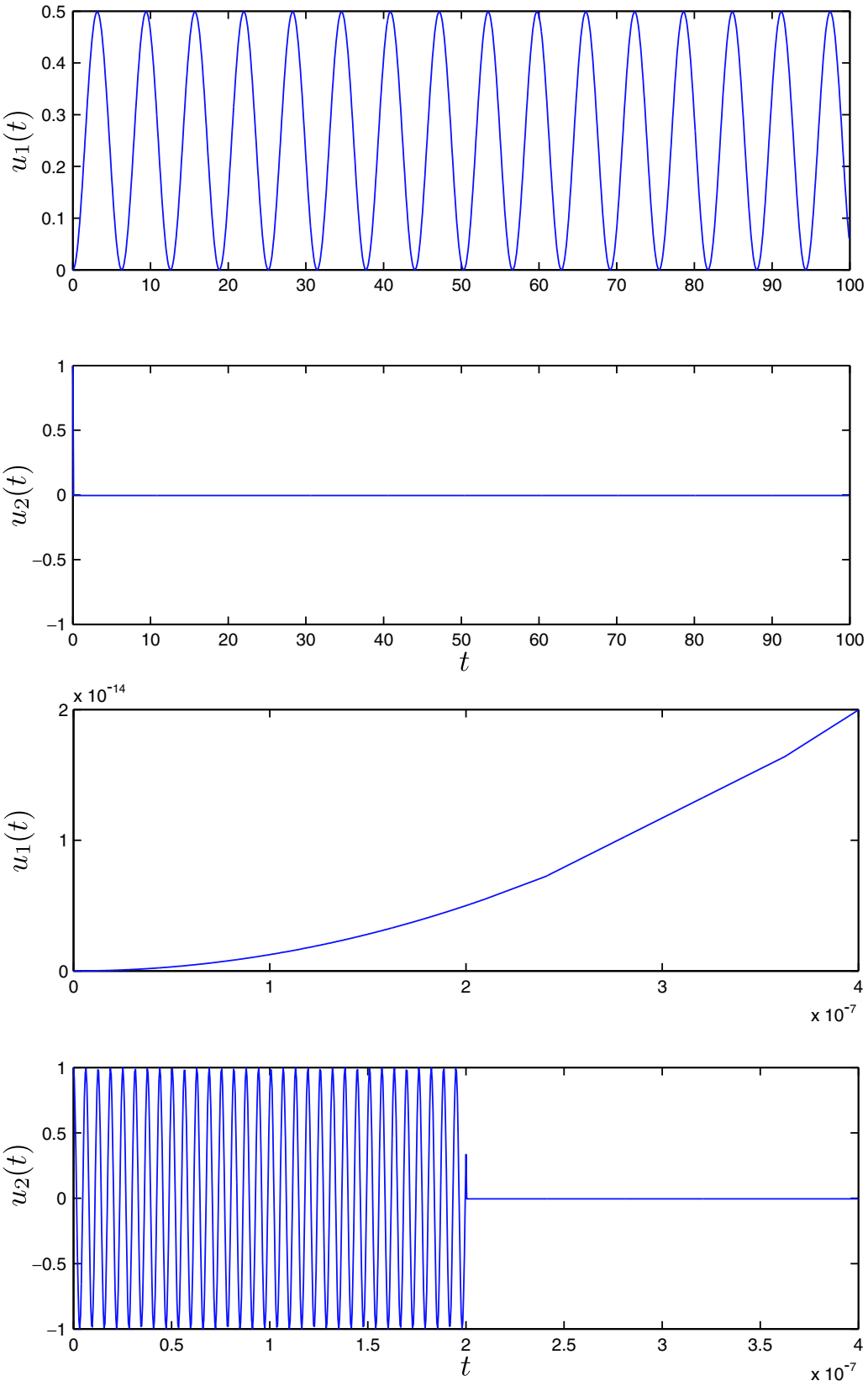


Fig. 2. The solution of the simple model problem (2.1) on $[0, 100]$ (above) and on $[0, 4 \times 10^{-7}]$ (below). The automatic modeling is activated at time $t = 2\tau = 2 \times 10^{-7}$.

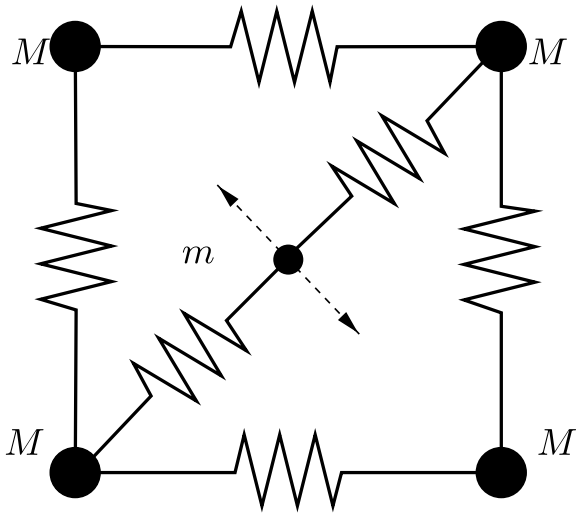


Fig. 3. Detail of the lattice. The arrows indicate the direction of vibration perpendicular to the springs connecting the small mass to the large masses.

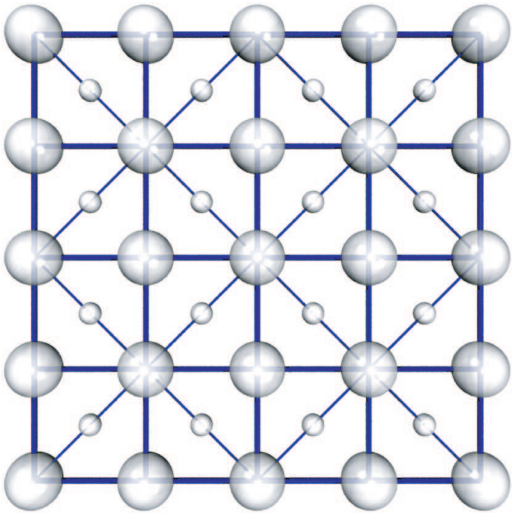
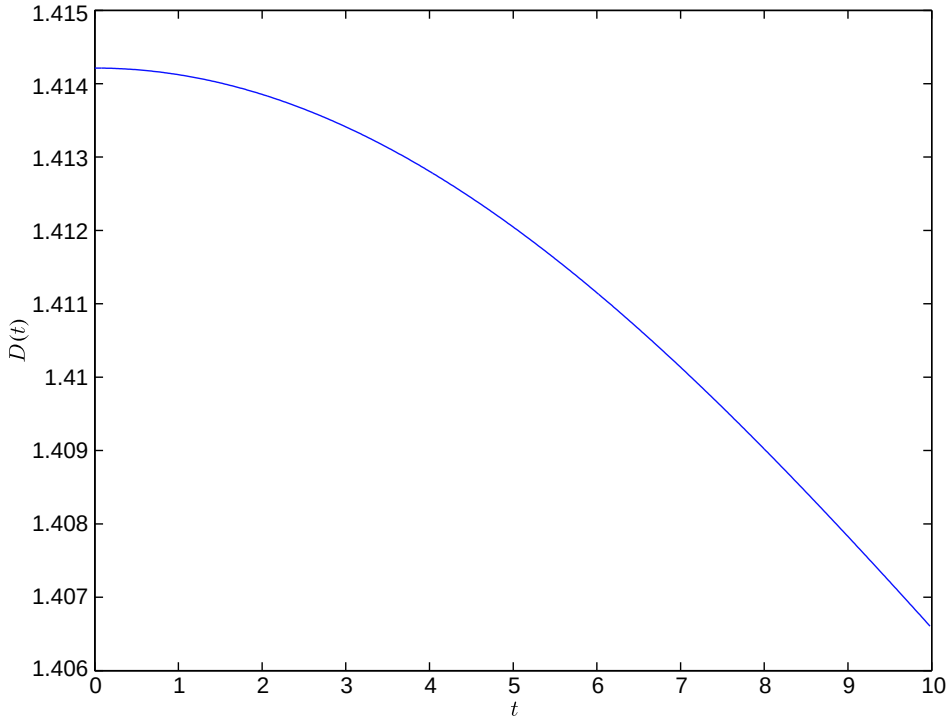


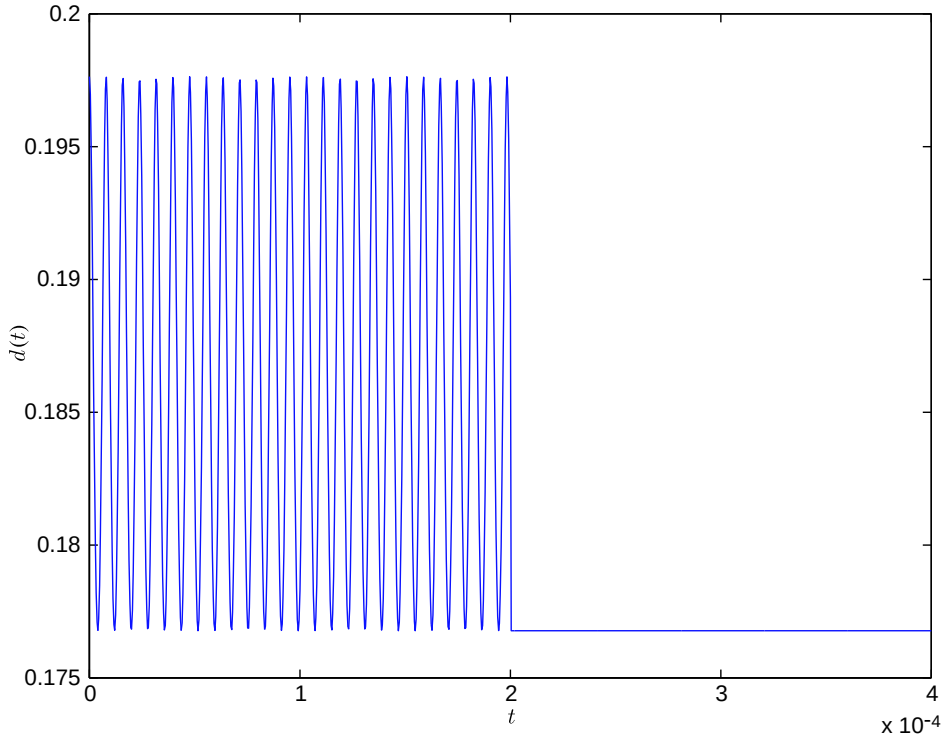
Fig. 4. Lattice consisting of p^2 large masses and $(p-1)^2$ small masses.

Each large point mass is of size $M = 100$ and each small point mass is of size $m = 10^{-12}$, giving a large time scale of size ~ 10 and a small time scale of size $\sim 10^{-6}$.

The fast oscillations of the small point masses make the initially stationary structure of large point masses contract. Without resolving the fast time scales and ignoring the subgrid model, the distance D between the lower left large point mass at $x = (0, 0)$ and the upper right large point mass at $x = (1, 1)$ remains constant, $D = \sqrt{2}$. In Fig. 5, we show the computed solution with $\tau = 10^{-4}$, which manages to correctly capture the oscillation in the diameter D of the lattice as a consequence of the internal vibrations at time scale 10^{-6}.



(a)



(b)

Fig. 5. (a) Distance D between the lower left large mass and the upper right large mass and (b) the distance d between the lower left large mass and the lower left small mass as function of time on $[0, 10]$ and on $[0, 4 \times 10^{-4}]$, respectively.

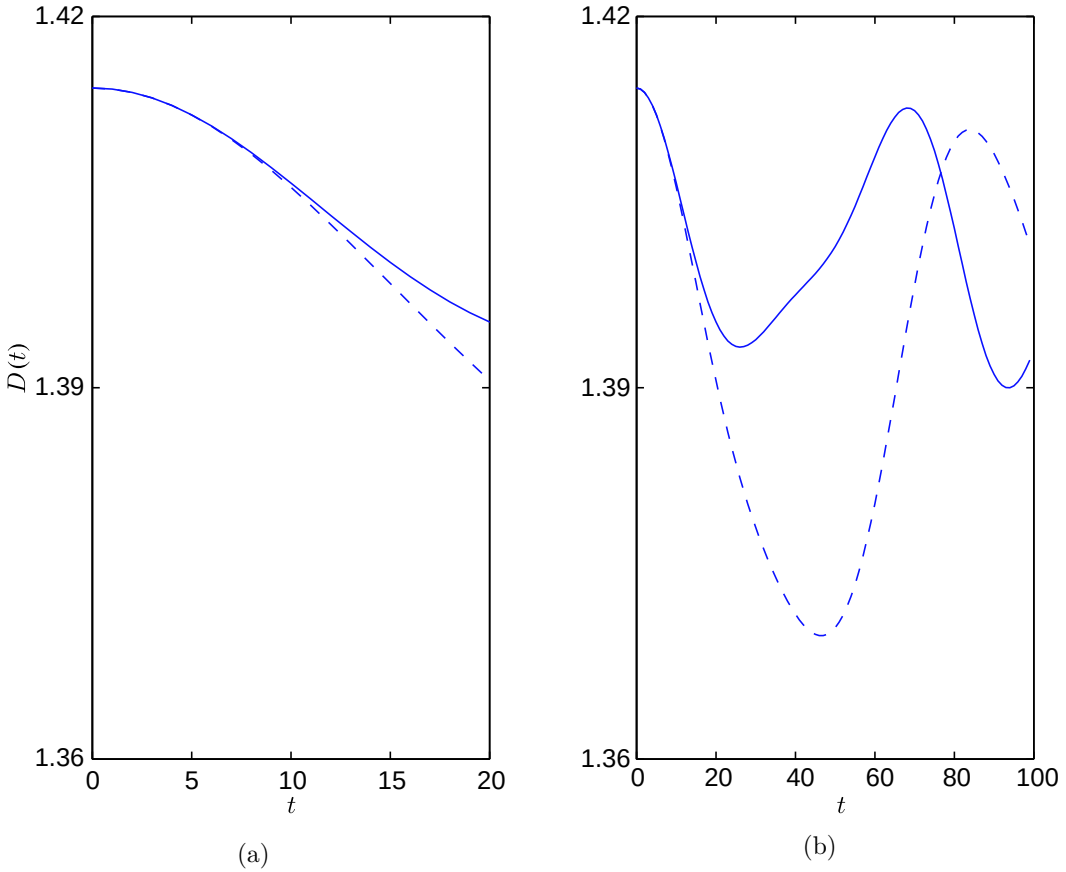


Fig. 6. The diameter D of the lattice as function of time on (a) $[0, 20]$ and on (b) $[0, 100]$ for $m = 10^{-4}$ and $\tau = 1$. The solid line represents the diameter for the solution of the reduced system (3.3) and the dashed line represents the solution of the full system (1.1).

With a constant subgrid model $\bar{g}$ as in the example, the reduced model stays accurate until the configuration of the lattice has changed sufficiently. When the change becomes too large, the reduced model can no longer give an accurate representation of the full system, as shown in Fig. 6. At this point, the reduced model needs to be reconstructed in a new short time simulation.

References

1. K. Eriksson, D. Estep, P. Hansbo and C. Johnson, Introduction to adaptive methods for differential equations, *Acta Numer.* **4** (1995) 105–158.
2. ——, *Computational Differential Equations* (Cambridge Univ. Press, 1996).
3. J. Hoffman, Computational modeling of complex flows, PhD thesis, Chalmers Univ. of Technology, 2002.
4. J. Hoffman, J. Jansson and A. Logg, DOLFIN, http://www.fenics.org/dolfin/
5. J. Hoffman and C. Johnson, Computability and adaptivity in CFD, to appear in *Encyclopedia of Computational Mechanics* (2004).
6. H.-O. Kreiss, Problems with different time scales, *Acta Numer.* (1991) 1.

7. A. Logg, Automation of computational mathematical modeling, PhD thesis, Chalmers Univ. of Technology, 2004.
8. A. Ruhe and D. Skoogh, Rational Krylov algorithms for eigenvalue computation and model reduction, in *Applied Parallel Computing — Large Scale Scientific and Industrial Problems*, eds. B. Kågström, J. Dongarra, E. Elmroth and J. Waśniewski, Lecture Notes in Computer Science, No. 1541, 1988, pp. 491–502.

DOLFIN: AN AUTOMATED PROBLEM SOLVING ENVIRONMENT

JOHAN HOFFMAN, JOHAN JANSSON, ANDERS LOGG, AND GARTH N. WELLS

1. Introduction

Natural science can be broken down into two components:

(I) formulating mathematical equations (modeling),
(II) solving equations (computation).

Problem solving is the task of selecting appropriate equations and data (geometry, coefficients) and interpreting the solution of the equations to solve a specific problem.

By *discretization* a given set of differential equations is translated into a discrete system of algebraic equations, which is solved using numerical algebra on a computer, to produce an approximation U of the solution u. Traditionally this has been done by hand. **DOLFIN** is part of the **FEniCS** project which is free software for the Automation of Computational Mathematical Modeling (ACMM) based on the finite element method (FEM). **DOLFIN** improves problem solving by providing automated discretization, error control (in time, with space in sight) and equation solving.

A differential equation can be written in the form:

$$A(u) = f \quad \text{in } \Omega \tag{1}$$

where A is a differential operator on some domain Ω, f is given input, i.e. forces and u is the solution.

Typically differential equations are used in variational form:

$$a(u, v) = L(v) \quad \text{in } \Omega, \forall v \in V, \tag{2}$$

with

Date: May 12, 2006.

Key words and phrases. ODE, PDE, FEM.

Johan Hoffman, School of Computer Science and Communication, Royal Institute of Technology KTH, SE-10044 Stockholm, Sweden, *email*: jhoffman@csc.kth.se

Johan Jansson, Computational Technology Group, Department of Applied Mechanics, Chalmers University of Technology, SE-412 96 Gteborg, Sweden, *email*: johan.jansson@chalmers.se

Anders Logg, Toyota Technological Institute at Chicago, University Press Building, 1427 East 60th Street, Chicago, Illinois 60637, USA, *email*: logg@tti-c.org

Garth N. Wells, Delft University of Technology, Faculty of Civil Engineering and Geosciences, Stevinweg 1, 2628 CN Delft, The Netherlands, *email*: g.n.wells@tudelft.nl.

$$a(u, v) = \int_\Omega A(u)v \tag{3}$$

$$L(f) = \int_\Omega fv \tag{4}$$

where V is a function space with some integrability requirement, typically $V = H^1(\Omega)^n$ for a system of n equations.

Traditionally problem solving has been equation-specific. An equation is selected, a method and solver is then derived or chosen specifically for that equation. For example, it is common to talk about a "Maxwell solver" or a "Navier-Stokes solver" which have been developed specifically for those equations.

When the equation is significantly changed, or a new equation is selected, the process needs to start from the beginning again. This implies much redundant manual work.

There exists tools which have a general perspective. However, they are typically limited either by flexibility (the form must be input in a specialized language) or performance (the form is slow to evaluate). See [8] for a survey of existing tools.

We present a free software tool called DOLFIN which combines generality with optimal efficiency. "Generality" means here that any equation can be input into DOLFIN essentially as it looks on paper. "Optimal" means here that DOLFIN is able to reach the same efficiency as a manually-developed solver for a specific equation.

DOLFIN is a component of the **FEniCS** tool-chain, where the role of **DOLFIN** is the problem solving environment, or programmer user interface for formulating and solving equations.

FEniCS is an implementation of the Finite Element Method (FEM) for arbitrary equations and arbitrary finite elements. The FEM is a general method for automating discretization of differential equations. However, this has seldom been reflected in practice. Traditionally, the FEM has been applied individually to a specific equation, and a solver has been manually constructed from the resulting formulation. We remedy this situation with **FEniCS**.

In this paper we will illustrate the generality and efficiency of **DOLFIN** by presenting the following aspects:

Simple form language: We can input any equation into **DOLFIN** in mathematical notation, meaning that no extensive re-formatting or manual manipulation is required.

Assembly efficiency: Assembly is the forming of a discrete system (equation system for the degrees of freedom) given a discretization (finite element and mesh) of an equation. This is the key step for solving an equation. **DOLFIN** achieves generality and full efficiency by generating assembly code from a description of the equation.

High-level programming interface: Generality means that we should enforce a high level of abstraction, and this also applies to the programming interface. **DOLFIN** publishes a high-level programming interface in C++ and Python. The Python interface enables Just-In-Time (JIT) compilation of generated assembly code, which

means that the code generation and compilation is transparent to the user of the interface.

PDE/ODE solver integration: DOLFIN provides capability for solving both initial value Ordinary Differential Equations (ODE) as well as Partial Differential Equations (PDE). We present a method which allows the general ODE solver in **DOLFIN** to be used for solving PDE by writing the PDE in the form $\dot{u} = f(t, u)$.

Applications: We present applications in incompressible fluid flow (Navier-Stokes' equations) and large deformation elasto-plasticity (Euler-Almansi and hyperelastic strain models).

2. Fenics

DOLFIN is a component of **FEniCS** [2], a free software [1] system for the Automation of Computational Mathematical Modeling (ACMM). The overall goal of ACMM is to build a computational machine which takes any PDE (in variational form) and a tolerance for the error as input, and automatically computes a solution to the model which satisfies the tolerance.

This task can be broken down into several sub-tasks, automation of:

(a) discretization of differential equations,
(b) solution of discrete systems,
(c) error control of computed solutions,
(d) optimization,
(e) modeling.

The essential step of (a), which concerns both (a1) *time-discretization* and (a2) *space-discretization*, is automated computation of finite element stiffness matrices and assembly to a global stiffness matrix. This requires efficient evaluation of integrals of combinations of derivatives of finite element basis functions over finite elements. **FEniCS** achieves this in the Fenics Form Compiler (**FFC**) by factorization of the element stiffness matrix into a reference and geometry tensor. The reference tensor contains integrals over a reference element computed once, which for each element upon multiplication a geometry tensor gives the element stiffness matrix for each element. The input to **FFC**is then the equation (2) in standard mathematical notation and a given finite element mesh and finite elements, and the output is computer code (e.g. C++) specifying the discrete system.

The step (b) is automated in **FEniCS** using, with input from **FFC**, the parallel numerical linear algebra package *Petsc*.

The essential step of (c) is automated computation of (c1) discrete residuals and (c2) stability factors/weights by automated formulation and solution of a dual linearized problem. **FEniCS** currently achives these requirements partially, with a full implementation in sight.

FEniCS consists of the following components:

FIAT: FInite element Automatic Tabulator [5]. Automates the generation of finite elements. Provides representation of finite elements and evaluation of basis functions as well as general quadrature for integrating basis functions.

FFC: Fenics Form Compiler [6]. Automates the evaluation of variational forms. Provides assembly code generation and a form language for equation input.

FErari: Finite Element rearrangement to automatically reduce instructions [7]. Optimizes the evaluation of variational forms. Detects and exploits structure in element tensors.

DOLFIN: Dynamic Object oriented Library for FINite element computation. The programmer user interface for solving equations. Provides a high-level C++ and Python interface to:

- assembly
- variational form representation
- finite element representation
- function representation (typically a solution or coefficient)
- mesh representation
- linear algebra algorithms
- initial value multi- and mono-adaptive ODE solver
- file input/output

The assembly algorithm of a finite element method is typically of high complexity, and is not trivial to implement efficiently manually. A modification of the variational form or the choice of finite element normally means that large parts of the code need to be reimplemented. This is a waste of human resources and, due to the complexity of the algorithm, may easily introduce errors in the implementation.

The finite element assembly algorithm for a bilinear form $a(\cdot, \cdot)$ generating a matrix A can be formulated as follows: For each element K, add the local element matrix $A_{ij}^K = a_K(\hat{\phi}_i, \phi_j)$ to A, where $a_K(\cdot, \cdot)$ is the bilinear form restricted to the current element K.

FFC parses the form and, together with a description of the finite element, generates source code for evaluation of the local element matrix A^K. **FFC** (using FIAT) precomputes integrals on the reference element. **FFC** uses FErari to exploit the structure of A^K to produce efficient code. This results in automatically generated source code which is as efficient as hand-written code. The generated assembly code is then linked into **DOLFIN** and can be accessed through the assembly interface.

3. Form language

Performance of the form parsing and code generation stage is not critical. It's a one-time cost which does not scale with the size of the final problem. This means that we can afford to implement and present the form language at a high level of abstraction, which enables a powerful, user-friendly and easily extendable form language.

FFC defines a form language which is very close to standard mathematical notation. The language is implemented as Python functions and operators.

A form is expressed using a combination of basic data types and operators. **FFC** compiles a given multilinear form

$$a : V_h^1 \times V_h^2 \times \cdots \times V_h^r \to \mathbb{R} \tag{5}$$

into code that can be used to compute the corresponding tensor

$$A_i = a(\phi^1_{i_1}, \phi^2_{i_2}, \ldots, \phi^r_{i_r}). \tag{6}$$

In the form language, a multilinear form is defined by first specifying the set of function spaces, $V^1_h, V^2_h, \ldots, V^r_h$, and then expressing the multilinear form in terms of the basis functions of these functions spaces.

A function space is defined in the form language through a `FiniteElement`, and a corresponding basis function is represented as a `BasisFunction`. The following code defines a pair of basis functions `v` and `U` for a first-order Lagrange finite element on triangles:

```
element = FiniteElement(''Lagrange'', ''triangle'', 1)
v = BasisFunction(element)
U = BasisFunction(element)
```

The two basis functions can now be used to define a bilinear form:

```
a = v*D(U, 0)*dx
```

corresponding to the mathematical notation

$$a(v, U) = \int_\Omega v \, \frac{\partial U}{\partial x_0} \, \mathrm{d}x. \tag{7}$$

The arity of a multilinear form is determined by the number of basis functions appearing in the definition of the form. Thus, `a = v*U*dx` defines a *bilinear form*, namely $a(v, U) = \int_\Omega v\, u \, \mathrm{d}x$, whereas `L = v*f*dx` defines a *linear form*, namely $L(v) = \int_\Omega v\, f \, \mathrm{d}x$.

In the case of a bilinear form, the first of the two basis functions is referred to as the *test function* and the second is referred to as the *trial function*.

See the **FFC** manual [2] for more detailed reference.

4. Assembly efficiency

The *assembly* algorithm is the core component of a PDE solver. It is thus imperative that it is efficient. A comparison between automatically generated **FFC** code and quadrature, which is what is typically used for general PDE, can be seen in figure 1 showing massive speedups [6].

5. High-level programming interface

5.1. **Interface documentation. DOLFIN** publishes a high-level programming interface in C++ (see the **DOLFIN** manual [3] for detailed reference). From this we automatically generate a Python interface by using SWIG: an interface generator. Such tools give us the choice of generating an interface in most major high-level langauges from a base C/C++ interface. We will thus not be language-specific when we describe the interface.

Having a high-level interface does not sacrifice performance. The interface wraps high-performance data structures and algorithms:

Linear algebra: Wraps PETSc parallel data structures and algorithms.

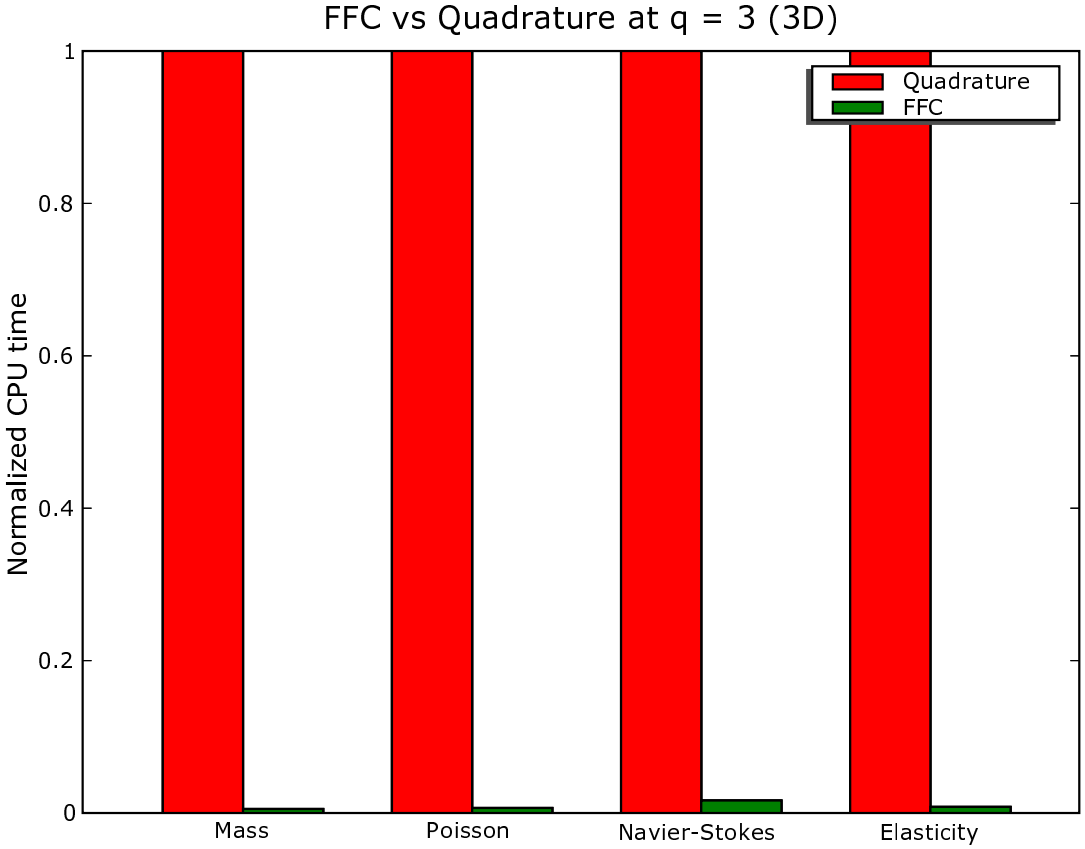


FIGURE 1. Benchmark of FFC-generated assembly code versus quadrature of some standard forms.

Form evaluation: FFC generates optimal C++ code for evaluating a form (the element tensor).

Mesh: Arguably not high-performance, but a full implementation of an adaptive mesh. Planned to be replaced by a high-performance parallel mesh representation.

Assembly: Implementation based on the mesh, form evaluation and linear algebra (insert element tensor into global matrix, vector). Reaches high performance due to the underlying components.

We present a list of the fundamental data structures and algorithms in the **DOLFIN** interface:

Data structures:

Fundamental data structures of **DOLFIN**.

Variational form:

Represents a PDE: $a(u, v) = L(v)$.

`Form:`
A general form.

`LinearForm:`
The linear form $L(v)$ representing the right hand side in a PDE. Publishes an evaluation function for computing the element vector representing the discrete form.

`BilinearForm:`
The bilinear form $a(u, v)$ representing the left hand side in a PDE. Publishes an evaluation function for computing the element matrix representing the discrete form.

Finite element:
Represents the programming interface of a finite element. Publishes functions such as a node map, mapping local to global degrees of freedom, and dimension of the finite element space.

`FiniteElement:`

Function:
Represents a spatial function $u(x, y, z)$ at a fixed time t. Used for representing solutions and coefficients of PDE.

`Function:`
A general spatial function.

`UserFunction:`
An arbitrary user-defined function.

`DiscreteFunction:`
A discrete function U defined on a finite element space V_h (spanned by basis functions $\{\phi_i\}_{i=0}^N$), represented by its degrees of freedom ξ on a mesh: $U = \sum_{i=0}^N \xi_i \phi_i$.

Mesh:
Data structures related to a computational mesh.

`Mesh:`
A mesh, publishing selector functions for cells, vertices, etc. and number of cells, vertices, etc. and mesh entitity iterators.

`Cell:`
A cell in the mesh: `Tetrahedron` or `Triangle`.

`Vertex:`
A vertex in the mesh.

Linear algebra:
Represents vectors and (sparse) matrices of values.

`Vector:`
Typically used for storing degrees of freedom ξ or an assembled discrete linear form $L(v)$.

`Matrix:`
Typically used for storing an assembled discrete bilinear form $a(u, v)$.

ODE:
Represents an ordinary differential equation (ODE) in the form: $\dot{u} = f(t, u)$ with initial value $u(0) = u_0$ and $t \in [0, T]$. Can be used to discretize time-dependent PDE in time.
`ODE:`

Input/output:
Data structures for input/output to files.
`File:`
Represents a file. Can be used to input/output meshes, functions, vectors, etc.

Algorithms:
Fundamental algorithms of **DOLFIN**.

Assembly:
Form a discrete system (equation system for the degrees of freedom) given a discretization (finite element and mesh) of a PDE.
`FEM::assemble():`
Assembles a `Vector` given a `LinearForm` or a `Matrix` given a `BilinearForm`, together with a `Mesh`.

Linear algebra:
Standard algorithms for linear algebra, such as direct and iterative (Krylov) linear solve, eigenvalue computation, etc.
`LinearSolver::solve():`
Solve a linear system $Ax = b$.

Time stepping:
Multi-adaptive (allows individual timestep selection per component) arbitrary order $cG(q)$ and $dG(q)$ Galerkin time-stepping methods for solving initial value ODE problems.
`ODE::solve():`
Solve the ODE generating an approximate solution $U(t)$, $t \in (0, T]$.
`TimeStepper::step():`
Take one step at a time, or one timeslab at a time in the multi-adaptive case.

5.2. **Example usage.** We present a simple example of using the interface in Python due to compactness and readability, demonstrating the elements of the interface. Coefficients need to be implemented in C++ (but can be accessed in Python) for full efficiency. Here we have implemented coefficients in Python.

The example consists of solving Poisson's equation: $\triangle u = f$.

We define a form (figure 2), representing the PDE and a solver (figure 3), where we define coefficients and problem parameters and construct the solution. The form is compiled transparently to the user by a JIT (Just In Time) procedure, using **FFC** and a C++ compiler.

Alternatively, we could use the **DOLFIN** abstraction `LinearPDE` which encapsulates the solver. We would then only need to input the form, coefficients and parameters (tolerances, etc.).

```
# The bilinear form a(v, U) and linear form L(v) for
# Poisson's equation, 2D version

element = FiniteElement("Lagrange", "triangle", 1)

v = TestFunction(element)
U = TrialFunction(element)
f = Function(element)

a = dot(grad(v), grad(U))*dx
L = f*v*dx
```

FIGURE 2. `Poisson2D.form`

6. PDE/ODE solver integration

We define a *time-dependent* PDE as a differential equation:

$$\begin{aligned} A(u) &= g(t) \quad \text{in } \Omega \times (0,T], \\ u(\cdot,0) &= u^0 \quad \text{in } \Omega, \\ u &= g_D \quad \text{on } \partial\Omega_D \times (0,T], \\ \partial_n u &= g_N \quad \text{on } \partial\Omega_N \times (0,T] \end{aligned} \tag{8}$$

where $u = u(t,x)$, A has derivatives of variables t and x and g_D and g_N are known boundary conditions on $\partial\Omega$.

We rewrite the equation (8) as:

$$\dot{u} = f(t,u) \quad \text{in } \Omega \times (0,T] \tag{9}$$

with f now containing the part of A depending on x and keeping the inital and boundary conditions on u.

We construct a variational formulation in x:

$$\int_\Omega \dot{u}v = \int_\Omega f(t,u)v \quad \text{in } \Omega \times (0,T], \forall v \in V \tag{10}$$

Choosing $V = V_h$ a finite element space allows us to formulate an algebraic equation system in x:

```
from dolfin import *
from math import *

# Define coefficients
class Source(Function):
    def eval(self, point, i):
        return point.y + 1.0

class SimpleBC(BoundaryCondition):
    def eval(self, value, point, i):
        if point.x == 0.0 or point.x == 1.0:
            value.set(0.0)
        return value

# Construct data
f = Source()
bc = SimpleBC()
mesh = UnitSquare(10, 10)

# Import forms
forms = import_formfile("Poisson.form")

a = forms.PoissonBilinearForm()
L = forms.PoissonLinearForm(f)

# Define linear algebra objects
A = Matrix()
x = Vector()
b = Vector()

# Assemble the discrete system
assemble(a, L, A, b, mesh, bc)

# Solve discrete (linear) system
linearsolver = KrylovSolver()
linearsolver.solve(A, x, b)

# Represent the solution (U)
trialelement = a.trial()
U = Function(x, mesh, trialelement)

# Output solution to ParaView (.pvd) format
vtkfile = File("poisson.pvd")
vtkfile << U
```

FIGURE 3. poissonsolver.py

$$M\dot{\xi} = b(t,\xi) \quad \text{in } (0,T] \tag{11}$$

where $U(t,x) = \sum_{i=0}^{N} \xi_i(t)\phi_i(x)$ is the finite element approximation of u and the basis functions $\{\phi_i\}_{i=0}^{N}$ span V_h, and $N = dim(V_h)$. Equation (11) is now in ODE form for the spatial degrees of freedom $\xi(t)$.

DOLFIN can automatically construct M and $b(t,\xi)$ from a description of $\int_\Omega \dot{u}v$ and $\int_\Omega f(t,u)v$ in the **FFC** form language, where $\dot{u}$ is treated as the unknown in the case of M and u is represented as a coefficient in the case of b. The ODE (11) can thus be input to the **DOLFIN** ODE solver, which then automatially and efficiently computes a finite element discretization in time.

This representation can be encapsulated in a **DOLFIN** abstraction which we can call `TimeDependentPDE` which encapsulates the solver. We would then only need to input the form for $f(t,u)$, coefficients and parameters (tolerances, etc.).

This concept is exemplified in the elasticity application below.

7. Applications

7.1. **Incompressible fluid flow.** Johan Hoffman has developed a General Galerkin (G2) solver in **DOLFIN** for the incompressible Navier-Stokes' equations expressing conservation of momentum and incompressibility [4]. In G2 the incompressibility equation takes the form of the following equation for the pressure P, velocity U and δ a stabilization parameter:

$$\int_\Omega \delta\nabla q \cdot \nabla P = \int_\Omega \delta\nabla q \cdot f - \nabla q \cdot \delta\nabla \cdot U - \delta\nabla q \cdot (U \cdot \nabla U), \quad \forall q \in V_h \tag{12}$$

This equation is expressed in the **FFC** form language in figure 4. Example output can be seen in figures 5, 7 and 6.

7.2. **Elasto-plasticity.** The following equations represent the equilibrium equation of elasticity with directly computed $B^{-1} = (FF^\top)^{-1}$ and the Euler-Almansi constitutive relation for σ:

$$\begin{aligned} \sigma &= \mu(I - (FF^\top)^{-1}), \\ \frac{Dv}{Dt} &= \nabla \cdot \sigma + f \quad \text{in } \Omega(t), \\ \frac{Dx}{Dt} &= v \quad \text{in } \Omega(t), \\ v(0,\cdot) &= v^0 \quad \text{in } \Omega^0, \\ x(0,y) &= y \quad \text{in } \Omega^0, \end{aligned} \tag{13}$$

together with suitable boundary conditions where F is the Jacobian of the deformation.

```
# The contiuity equation for the incompressible
# Navier-Stokes equations using cG(1)cG(1)

name = "NSEContinuity3D"
scalar = FiniteElement("Lagrange", "tetrahedron", 1)
vector = FiniteElement("Vector Lagrange", "tetrahedron", 1)
constant_scalar = FiniteElement("Discontinuous Lagrange",
                                "tetrahedron", 0)

q  = TestFunction(scalar)  # test basis function
P  = TrialFunction(scalar) # trial basis function
uc = Function(vector)      # linearized velocity
f  = Function(vector)      # force term

delta1 = Function(constant_scalar) # stabilization parameter

um = mean(uc) # cell mean value of linearized velocity

i0 = Index() # index for tensor notation
i1 = Index() # index for tensor notation

# Bilinear and linear forms
a = delta1*dot(grad(q), grad(P))*dx;
L = delta1*dot(grad(q), f)*dx - q*uc[i0].dx(i0)*dx -
    delta1*q.dx(i0)*um[i1]*uc[i0].dx(i1)*dx
```

FIGURE 4. Form for the continuity equation of Navier-Stokes.

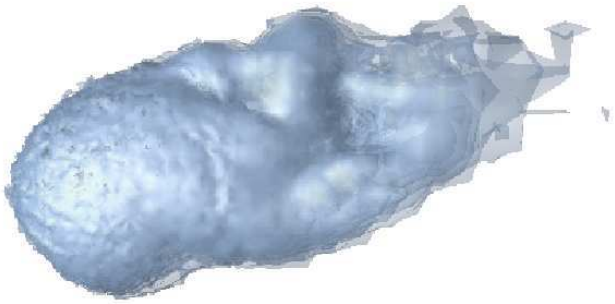


FIGURE 5. Turbulent flow around a spinning tennis ball.

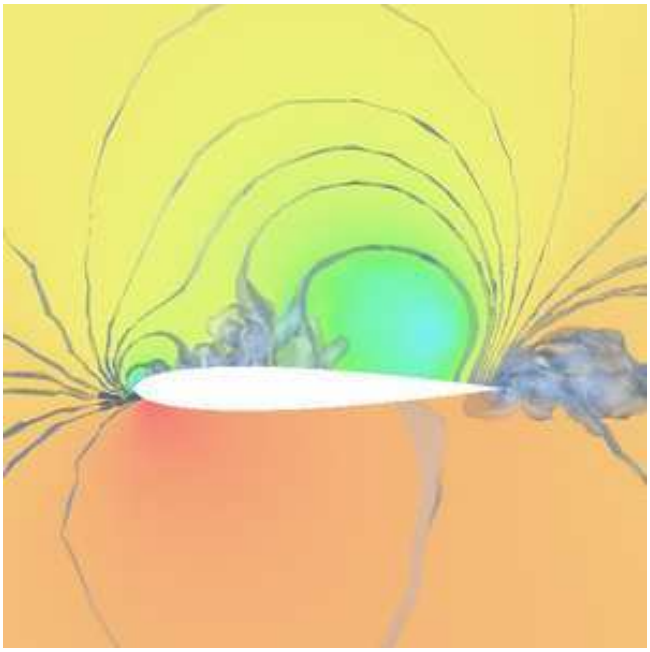


FIGURE 6. Turbulent flow around a wing.

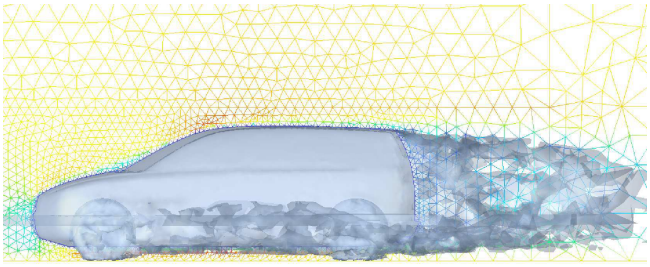


FIGURE 7. Turbulent flow around a car (geometry and mesh from Volvo Car Corporation).

This equation is expressed in the **FFC** form language in figure 8. Example output can be seen in figures 10 and 11 with the latter demonstrating a variant of the model in stress rate formulation with a model for plasticity.

The actual **DOLFIN** solver is very small and simple: most of the work is automated. It consists of declarations of solution and coefficient functions, the variational forms and using the interface of the **DOLFIN** ODE solver (time discretization) by implementing the right hand side $f(t, u)$ of an ODE: $\frac{du}{dt} = f(t, u)$. The solver is mostly written using the **DOLFIN** Python interface, with some utility functions implemented in C++. As an illustration, we present the implementation of $f(t, u)$ for the elasticity model where the task is to compute the vector `dotx` from the vector `x`.

```
# Form representing the equilibrium equation of elasticity

name = "ElasticityDirect"
element1 = FiniteElement("Vector Lagrange", "tetrahedron", 1)
element2 = FiniteElement("Discontinuous vector Lagrange",
                         "tetrahedron", 0, 9)

q = TestFunction(element1) # Test function
dotv = TrialFunction(element1) # Trial function
f = Function(element1) # Body force
B = Function(element2) # Deformation measure

lmbda   = Constant() # Lame coefficient
mu      = Constant() # Lame coefficient

# Dimension
d = len(q)

# Manual tensor representation
def tomatrix(q):
    return [ [q[3 * j + i] for i in range(d)] for j in range(d) ]

Bmatrix = tomatrix(B)

def E(e, lmbda, mu):
    Ee = 2.0 * mult(mu, e) + mult(lmbda, mult(trace(e), Identity(d)))

    return Ee

ematrix = 0.5 * (Identity(d) - Bmatrix)

sigmamatrix = E(ematrix, lmbda, mu)

a = dot(dotv, q) * dx
L = (-dot(sigmamatrix, grad(q)) + dot(f, q)) * dx
```

FIGURE 8. Form for a total stress Euler-Almansi elasticity model.

REFERENCES

[1] FREE SOFTWARE FOUNDATION, *GNU GPL*. http://www.gnu.org/copyleft/gpl.html.

```
def fu(self, x, dotx, t):

    # Scatter from x to components
    Vector_scatter(self.xu, self.x, self.dotxu_sc)
    Vector_scatter(self.xv, self.x, self.dotxv_sc)

    # Mesh
    Elasticity_deform(self.mesh(), self.U)

    # B (computed directly)
    Elasticity_computeB(self.xB, self.xF0, self.xF1,
                        self.B.element(), self.mesh())

    # U
    self.dotxu.copy(self.xv)

    # V
    assemble(self.L(), self.dotxv, self.mesh())
    self.dotxv.div(self.m)

    # Gather components into dotx
    Vector_gather(self.dotxu, self.dotx, self.dotxu_sc)
    Vector_gather(self.dotxv, self.dotx, self.dotxv_sc)
```

FIGURE 9. DOLFIN Python code for the Ko elasticity solver (right hand side of ODE).

[2] J. HOFFMAN, J. JANSSON, C. JOHNSON, M. KNEPLEY, R. C. KIRBY, A. LOGG, AND L. R. SCOTT, *FEniCS*. http://www.fenics.org/.

[3] J. HOFFMAN, J. JANSSON, AND A. LOGG, *DOLFIN*, 2005. http://www.fenics.org/dolfin/.

[4] J. HOFFMAN AND C. JOHNSON, *Applied Mathematics: Body and Soul*, vol. IV, Springer-Verlag, 2006. In press.

[5] R. C. KIRBY, *FIAT: A new paradigm for computing finite element basis functions*, ACM Trans. Math. Software, 30 (2004), pp. 502–516.

[6] R. C. KIRBY AND A. LOGG, *A compiler for variational forms*. submitted to ACM Trans. Math. Softw., 2005.

[7] R. C. KIRBY, A. LOGG, L. R. SCOTT, AND A. R. TERREL, *Topological optimization of the evaluation of finite element matrices*. submitted to SIAM J. Sci. Comput., 2005.

[8] A. LOGG, *Automating the finite element method*, Tech. Rep. 2006–01, Finite Element Center Preprint Series, 2006.

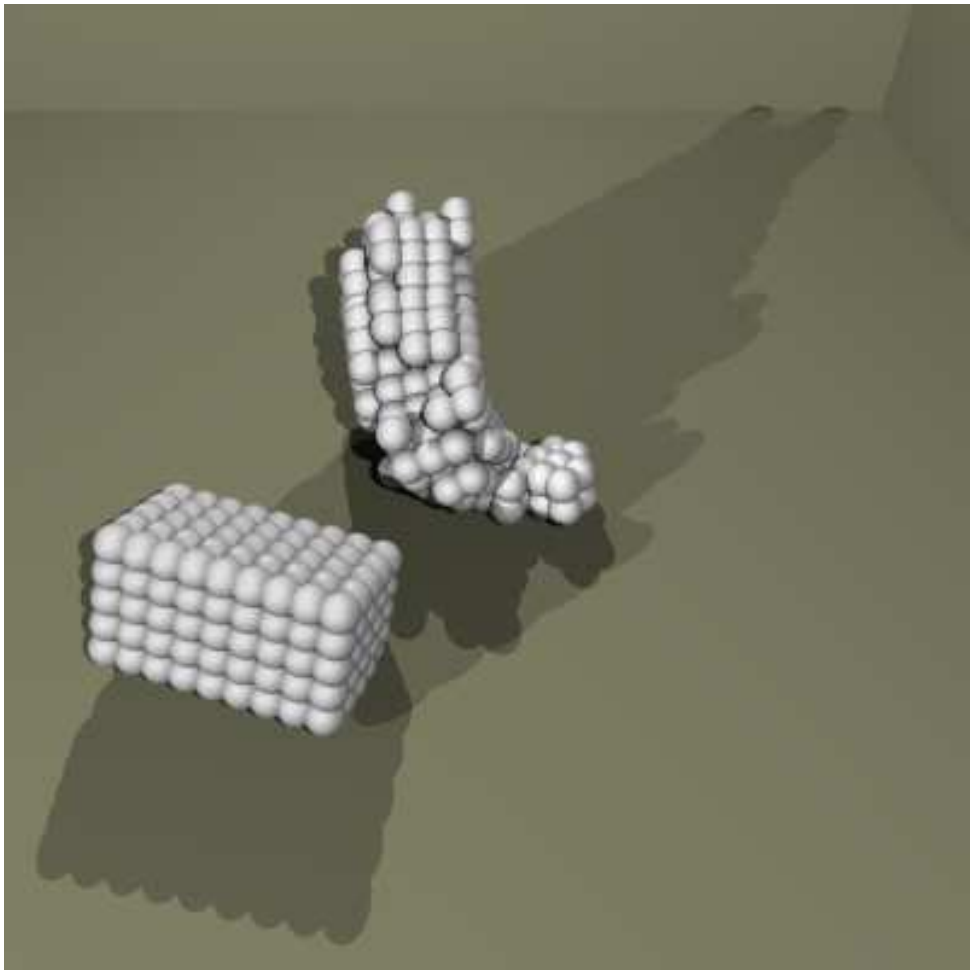


FIGURE 10. Example DOLFIN output of visco-elasto-plastic model solid mechanics model with contact, simulating a cow and a block being thrown in a room.

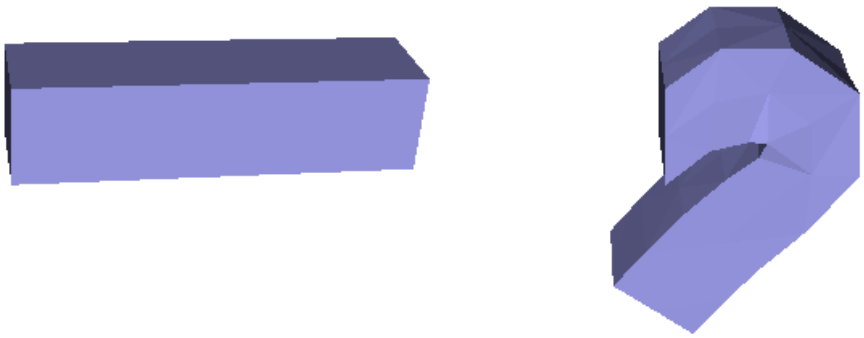


FIGURE 11. Bent beam simulated using a plastic stress rate model. The right image is the final deformed rest state after an initial velocity.

KO: A FENICS SOLID MECHANICS SOLVER

JOHAN JANSSON

1. Introduction

We present a solid mechanics finite element method (FEM) solver according to the following design specifications:

(a) Constitutive modeling including elastic, viscous and plastic materials.
(b) Large displacements, rotations and deformations.
(c) Contact and friction boundary conditions.
(d) Adaptive modeling and discretization with error control.
(e) Efficiency.

The solver is named Ko and is part of the **FEniCS**[3] project of Automation of Computational Mathematical Modeling (ACMM) [9]. Our objective is to develop an automated solid mechanics solver which can be used e.g. in real-time in computer games and simulators for surgical training, for which flexibility and execution speed represent key requirements.

The solver is based on an updated Lagrangian continuum formulation where the equilibrium equation is formulated on the current deformed configuration in terms of the Cauchy stress, and the constitutive model couples the Cauchy stress to the deformation, or rates thereof, on the current configuration.

We will find that the efficiency of Ko is comparable to that of mass-spring models, which represent the current standard for high-performance and flexibility in the computer graphics industry. We achieve this goal by using the **FEniCS** Form Compiler (**FFC**) to compute the element stiffness tensor in each time step or iteration, which represents the main computational work. We show computations on a PC allowing realistic simulations in real-time using adaptive meshes with up to a couple of thousand nodes.

The present continuum model allows automation of modeling and discretization, while automation of mass-spring models is much more difficult (if at all possible) to achieve. We can thus reach the execution speed of a mass-spring model while keeping the advantages of a continuum model as concerns automation of modeling and discretization. We thus present a concrete example of Automated Solid Mechanics (ASM) as a basic aspect of the ACMM of **FEniCS**.

Debunne et. al. [1] have previously presented a manually implemented similar pure elasticity model for real-time applications.

2. Elasto-Visco-Plastic Model

We consider an elasto-visco-plastic body B subject to large deformations under some load. We assume that B occupies the bounded domain Ω^0 in R^3 at time $t = 0$, and we

denote by $x(t, X)$ the coordinates at time $t > 0$ of a material point in B with coordinates $X \in \Omega^0$ at time $t = 0$. Let $\Omega(t) = \{x(t, X) : X \in \Omega^0\}$ be the domain in R^3 occupied by the elastic body B at time $t > 0$. The model consists of a system of partial differential equations on the current configuration $\Omega(t)$ expressing for $t > 0$:

(i) equilibrium of forces.
(ii) constitutive model coupling stress to strain, or rates thereof,

combined with initial and boundary conditions. The equilibrium equation has a generic form, while there is a very large variety of constitutive models. We will limit the constitutive models to a few basic forms including elastic-visco-plastic materials. We focus on the development of an efficient solver including (a)-(d), and not constitutive modeling, which is a very rich (and complex) field.

For purely elastic materials the constitutive equation may relate the total stress to the total strain, from which rate models may be derived. With visco-plastic materials it is natural to formulate the constitutive equation directly in rate form.

We will below denote by E a constant matrix of elasticity coefficients, typically of the form

$$Ee = \lambda \sum_k e_{kk}\delta + 2\mu e,$$

where $\lambda > 0$ and $\mu > 0$ are Lamé coefficients, $e = (e_{ij})$ is a strain tensor and δ denotes the Kronecker delta.

2.1. **Deformation and Strain.** Define $F(t, y)$ with $y = x(t; X) \in \Omega(t)$ by

$$F(t, y) = F(t, x(t; X)) = \frac{\partial}{\partial X} x(t; X),$$

that is, $F(t, y)$ is the Jacobian of the mapping $X \to y = x(t, X)$ evaluated at X.

The *Euler-Almansi strain tensor* $e(t, y)$ defined by

$$e(t, y) = \frac{1}{2}(I - (F(t, y)F(t, y)^\top)^{-1}),$$

measures the strain (or deformation) with respect to the deformed configuration. Here and below, $\top$ denotes the transpose and I is the identity matrix.

2.2. **Material Time-Derivative.** We define the velocity $v(t, y)$ for $y = x(t; X) \in \Omega(t)$ by

$$v(t, x(t, X)) = \frac{d}{dt} x(t; X).$$

We also introduce the *material time derivative* $\frac{Dw}{Dt}$ of a function $w(t, y)$ by

$$\frac{Dw}{Dt}(t, x(t; X)) = \frac{d}{dt} w(t, x(t; X)).$$

2.3. **Deformation Rate.** We compute

$$\begin{aligned}\frac{DF}{Dt}(t,x(t;X)) &= \frac{d}{dt}F(t,x(t;X)) = \frac{d}{dt}\frac{\partial}{\partial X}x(t;X)\\ &= \frac{\partial}{\partial X}v(t,x(t;X)) = \frac{\partial v}{\partial x}(t,x(t;X))\frac{\partial x}{\partial X}(t,x(t;X))\\ &= \nabla v(t,x(t;X))F(t,x(t;X)),\end{aligned}$$

where $\nabla v(t,y)$ is the gradient of $v(t,y)$ with respect to y. Thus

$$\frac{DF}{Dt} = \nabla v\, F \tag{1}$$

and using that $\frac{DF}{Dt}F^{-1} + F\frac{D}{Dt}F^{-1} = 0$, which follows by differentiating $FF^{-1} = I$, we have

$$\frac{D}{Dt}F^{-1} = -F^{-1}\nabla v. \tag{2}$$

2.4. **Equilibrium Equation.** The equilibrium equation at time $t > 0$ in terms of the *Cauchy stress* $\sigma(t,y)$ takes the form

$$\rho\frac{Dv}{Dt}(t,y) - \nabla\cdot\sigma(t,y) = f(t,y) \quad \text{for } y\in\Omega(t) \tag{3}$$

where $\nabla\cdot\sigma$ denotes the divergence of $\sigma(t,y)$ with respect to the y-coordinates, ρ denotes the density, and $f(t,y)$ is a given volume force. For simplicity, we assume here ρ to be constant $\rho = 1$.

2.5. **Hyper/hypo-elasticity.** A simple *hyper-elastic* constitutive model derived from a a stored-energy functional, takes the form ([10]):

$$J\sigma \equiv \tau(t,y) = \lambda\frac{J^2-1}{2}I + \mu(B-I), \tag{4}$$

where $J = \det F$, τ is the *Kirchhoff stress*, $B = FF^\top$ and $\lambda \geq 0$ and $\mu > 0$ are Lamé parameters.

We derive a rate form of (4) by computing, assuming $\lambda = 0$ for simplicity,

$$\begin{aligned}\frac{D\tau}{Dt} &= \mu(\dot F F^\top + F\dot F^\top) = \mu(\nabla v B + B\nabla v^\top)\\ &= 2\mu\epsilon(v) + \nabla v\tau + \tau\nabla v^\top,\end{aligned}$$

where we used that $B = I + \frac{1}{\mu}\tau$ and we define the *strain rate* $\epsilon(v) = \frac{1}{2}(\nabla v + \nabla v^\top)$. Defining now

$$\dot\tau \equiv \frac{D\tau}{Dt} - \nabla v\tau - \tau\nabla v^\top,$$

which coincides with the *Lie derivative* of τ, we can write the hyper-elastic constitutive equation (4) in rate form as

$$\dot\tau = 2\mu\epsilon(v). \tag{5}$$

More generally, we may consider a *hypo-elastic* rate model of the form

$$\dot{\tau} = E\epsilon(v) \tag{6}$$

or

$$\dot{\sigma} = E\epsilon(v). \tag{7}$$

where the dot may indicate any objective stress rate, such as the Jaumann rate.

2.6. **Hypo-elasto-visco-plasticity.** We shall consider a basic hypo-elasto-visco-plastic model of the following rate form:

$$\dot{\sigma} + \frac{1}{\nu}(\sigma - \pi\sigma) = E\epsilon(v), \tag{8}$$

where $\nu > 0$ is a viscosity and $\pi\sigma$ denotes the projection of σ onto a (convex) set of plastically admissible stresses.

2.7. **Euler-Almansi.** A simple constitutive model based on the Euler-Almansi strain e, takes the form

$$\sigma = \mu(I - (FF^\top)^{-1})$$

or more generally $\sigma = Ee$. The corresponding rate model takes the form.

$$\frac{D\sigma}{Dt} = 2\mu\epsilon(v) - \sigma\nabla v - \nabla v^\top\sigma,$$

which is similar, but not identical to the above hyper-elastic model.

3. Hyperelasticity: Total Stress

We formulate the hyperelastic problem with total stress constitutive model as follows:

Find the motion $X \to x(t;X)$ with velocity $v(t,x(t;X)) = \frac{d}{dt}x(t;X)$ and $x(0;X) = X$, such that for $t > 0$

$$\begin{aligned}
\frac{DF}{Dt} &= \nabla v\, F \quad \text{in } \Omega(t),\\
\sigma &= J^{-1}(\lambda\frac{J^2-1}{2}I + \mu(B-I)) \quad \text{in } \Omega(t),\\
\frac{Dv}{Dt} &= \nabla\cdot\sigma + f \quad \text{in } \Omega(t),\\
\frac{Dx}{Dt} &= v \quad \text{in } \Omega(t),\\
v(0,\cdot) &= v^0 \quad \text{in } \Omega^0,\\
x(0,y) &= y \quad \text{in } \Omega^0,\\
F(0,\cdot) &= F^0 \quad \text{in } \Omega^0,
\end{aligned} \tag{9}$$

together with suitable boundary conditions. Note that here ∇ refers to the coordinates $y \in \Omega(t)$ in the current deformed configuration, while the coordinates in the initial configuration are denoted by $X \in \Omega^0$.

In this formulation, the initial configuration of the elastic body is not kept, and the Jacobian F is successively updated using the equation for $\frac{DF}{Dt}$. It is possible to directly compute F by direct differentiation of $x(t;X)$ with respect to X, in which case the initial configuration must be kept. We prefer the rate equation without storage of the initial configuration since it reflects the physics better by storing only the current deformed configuration. In both cases we of course expect the elastic body to return to the initial configuration under appropriate unloading.

We can may replace the constitutive equation with a different stress-strain relation, for example the Euler-Almansi model. We may also introduce a viscous effect by modifying the equlibrium equation to:

$$\frac{Dv}{Dt} - \nu\nabla\cdot\epsilon(v) = \nabla\cdot\sigma + f \quad \text{in } \Omega(t),$$

where ν is a viscosity.

4. Hypo-Elasto-Visco-Plasticity: Stress Rate

We formulate a hypo-elasto-visco-plastic model with stress rate constitutive equation as follows: Find the motion $X \to x(t;X)$ with velocity $v(t,x(t;X)) = \frac{d}{dt}x(t;X)$ and $x(0;X) = X$, such that for $t > 0$

$$\begin{aligned}
\dot\sigma &= E\epsilon(v) - \frac{1}{\nu}(\sigma - \pi\sigma) \quad \text{in } \Omega(t),\\
\frac{Dv}{Dt} &= \nabla\cdot\sigma + f \quad \text{in } \Omega(t),\\
\frac{Dx}{Dt} &= v \quad \text{in } \Omega(t),\\
v(0,\cdot) &= v^0 \quad \text{in } \Omega^0,\\
x(0,y) &= y \quad \text{in } \Omega^0.
\end{aligned} \tag{10}$$

where $\pi\sigma$ is the projection on to the set of admissible stresses, Y_s is the yield stress of the material and ν_p is the plastic viscosity.

If the stress σ in a body remains within the elastic region: $\|\sigma\| \le Y_s$, the plastic term becomes:

$$\frac{1}{\nu_p}(\sigma - \pi\sigma)) = \frac{1}{\nu_p}(\sigma - \sigma)) = 0 \tag{11}$$

and the body remains fully elastic.

5. Variational formulation

We use **FFC** to automate discretization in space and **FEniCS** MG ode-solver to automate discretization in time. The input is the model in variational form, a finite element mesh on the initial configuration and a choice finite elements in space and time.

For the FEM discretization in space, we use piecewise linear continuous functions in the equilibrium equation and piecewise constants in the constitutive law. For the FEM discretization in time we use dG(0) or cG(1).

6. Automated discretization in FEniCS

Ko is implemented based on **FEniCS**[3], a free software [2] system for ACMM. The overall goal of ACMM is to build a computational machine which takes any PDE (in variational form) and a tolerance for the error as input, and automatically computes a solution to the model which satisfies the tolerance.

FEniCS consists of the following components:

FIAT: FInite element Automatic Tabulator [5]. Automates the generation of finite elements. Provides representation of finite elements and evaluation of basis functions as well as general quadrature for integrating basis functions.

FFC: Fenics Form Compiler [6]. Automates the evaluation of variational forms. Provides assembly code generation and a form language for equation input.

FErari: Finite Element rearrangement to automatically reduce instructions [7]. Optimizes the evaluation of variational forms. Detects and exploits structure in element tensors.

DOLFIN: Dynamic Object oriented Library for FINite element computation. The programmer user interface for solving equations. Provides a high-level C++ and Python interface to:

- assembly
- variational form representation
- finite element representation
- function representation (typically a solution or coefficient)
- mesh representation
- linear algebra algorithms
- initial value multi- and mono-adaptive ODE solver
- file input/output

The form for the total stress model with directly computed $B^{-1} = (FF^\top)^{-1}$ is presented in figure 1. The forms for the stress rate model are presented in figures 2, 3 and 4.

A high-performance combination of the models should adaptively transition to the plastic model when the yield condition is exceeded. Currently we implement the elastic and plastic models separately.

FFC currently does not fully support tensor-valued forms. We remedy this by manually defining a function which represents tensors as vectors.

FEniCS supports composing several finite elements into one, this would allow us to express every model as a single form. For now we use separate forms, which implies a small overhead.

6.1. Solver.

The actual **FEniCS** solver is very small and simple: most of the work is automated. It consists of declarations of solution and coefficient functions, the variational

```
# Form representing the equilibrium equation

name = "ElasticityDirect"
element1 = FiniteElement("Vector Lagrange", "tetrahedron", 1)
element2 = FiniteElement("Discontinuous vector Lagrange",
                         "tetrahedron", 0, 9)

v = BasisFunction(element1)
#f = Function(element1)
B = Function(element2)

lmbda   = Constant() # Lame coefficient
mu      = Constant() # Lame coefficient

# Dimension
d = len(v)

def tomatrix(q):
    return [ [q[3 * j + i] for i in range(3)] for j in range(3) ]

Bmatrix = tomatrix(B)

def E(e, lmbda, mu):
    Ee = 2.0 * mult(mu, e) + mult(lmbda, mult(trace(e), Identity(d)))

    return Ee

epsilonm = 0.5 * (Identity(d) - Bmatrix)

sigmamatrix = E(epsilonm, lmbda, mu)

L = (-dot(sigmamatrix, grad(v))) * dx
```

FIGURE 1. Equilibrium equation of total Euler-Almansi model.

forms and using the interface of the MG ODE solver (time discretization) by implementing the right hand side $f(t,u)$ of an ODE: $\frac{du}{dt} = f(t,u)$. The solver is mostly written in Python, using **DOLFIN** as a module, with some utility functions implemented in C++. As an illustration, we present the implementation of $f(t,u)$ for the Euler-Almansi rate model where the task is to compute the vector `dotx` from the vector `x`.

We lump the mass matrix resulting from discretization of $\frac{du}{dt}$ in space which allows us to efficiently use the MG fixed-point iteration solver for the resulting algebraic system.

```
# Form representing the equilibrium equation

name = "ElasticityUpdatedEquilibrium"
element1 = FiniteElement("Discontinuous vector Lagrange",
                         "tetrahedron", 0, 9)
element2 = FiniteElement("Vector Lagrange", "tetrahedron", 1)

nuv = Constant()        # viscosity coefficient

v = TestFunction(element2)
#f = Function(element2)
sigma = Function(element1)
epsilon = Function(element1)

def tomatrix(q):
    return [ [q[3 * i + j] for i in range(3)] for j in range(3) ]

sigmamatrix = tomatrix(sigma)
epsilonmatrix = tomatrix(epsilon)

L = (-dot(sigmamatrix, grad(v)) -
     nuv * (dot(epsilonmatrix, grad(v)))) * dx
```

FIGURE 2. Equilibrium equation of stress rate Euler-Almansi model.

7. The mass-spring model

We have earlier in [4] described an extended mass–spring model for the simulation of systems of deformable bodies.

The mass–spring model represent bodies as systems of discrete *mass elements*, with the forces between the mass elements transmitted using explicit *spring connections*. (Note that "spring" is a historical term, and is not limited to pure Hookian interactions). Given the forces acting on an element, we can determine its motion from Newton's second law,

$$\frac{dv}{dt} = \frac{F}{m} \tag{12}$$

where F denotes the force acting on the element, m is the mass of the element and v is the velocity of the element with x the current coordinate of the element. The motion of the entire body is then implicitly described by the motion of its individual mass elements.

The force given by a standard spring is assumed to be proportional to the elongation of the spring from its rest length. We extend the standard model with contact, collision and fracture, by adding a radius of interaction to each mass element, and dynamically creating and destroying spring connections based on contact and fracture conditions.

```
# Form representing the stress equation

name = "ElasticityUpdatedStress"
element1 = FiniteElement("Vector Lagrange", "tetrahedron", 1)
element2 = FiniteElement("Discontinuous vector Lagrange",
                         "tetrahedron", 0, 9)
element3 = FiniteElement("Discontinuous Lagrange", "tetrahedron", 0)

lmbda   = Constant() # Lame coefficient
mu      = Constant() # Lame coefficient
nuplast = Constant() # Plastic viscosity

q = TestFunction(element2)
v = Function(element1)
sigma = Function(element2)
B = Function(element2)
sigmanorm = Function(element3)

# Dimension
d = len(v)

def epsilon(u):
    return 0.5 * (grad(u) + transp(grad(u)))

def E(e, lmbda, mu):
    Ee = 2.0 * mult(mu, e) + mult(lmbda, mult(trace(e), Identity(d)))

    return Ee

def tomatrix(q):
    return [ [q[3 * j + i] for i in range(3)] for j in range(3) ]

qmatrix = tomatrix(q)
sigmamatrix = tomatrix(sigma)
Bmatrix = tomatrix(B)

Lplast = dot(E(sigmamatrix, lmbda, mu), qmatrix)

ederiv = 0.5 * (mult(Bmatrix, grad(v)) + mult(transp(grad(v)), Bmatrix))

Lelast = dot(E(ederiv, lmbda, mu), qmatrix)

L = (Lelast - nuplast * (1 - sigmanorm) * Lplast) * dx
```

FIGURE 3. Stress rate equation of differentiated Euler-Almansi model.

```
# Form representing the deformation equation

name = "ElasticityUpdatedDeformation"

element1 = FiniteElement("Vector Lagrange", "tetrahedron", 1)
element2 = FiniteElement("Discontinuous vector Lagrange",
                         "tetrahedron", 0, 9)

q = TestFunction(element2)
v = Function(element1)
F = Function(element2)

# Dimension
d = len(v)

def tomatrix(q):
    return [ [q[3 * j + i] for i in range(3)] for j in range(3) ]

qmatrix = tomatrix(q)
Fmatrix = tomatrix(Fmatrix)

L = dot(-mult(Fmatrix, grad(v)), qmatrix) * dx
```

FIGURE 4. Deformation equation of differentiated Euler-Almansi model.

In Table 1, we give the basic properties of the mass–spring model consisting of mass elements and spring connections. With these definitions, a mass–spring model may thus be given by just listing the mass elements and spring connections of the model.

The mass-spring model is analogous to the Euler-Almansi model with directly computed deformation B (or F).

8. The contact model

Traditionally, contact models in solid mechanics (i.e. the penalty method) have been based on kinematics. This means that that a condition of non-penetration of bodies is enforced. We believe that a model should describe the mechanics and not kinematics of a phenomenon: the contact model should model the forces resulting from the contact.

Contact (and its inverse: fracture) is an elastoplastic pheonomenon, and we have formulated several such a model above. However, contact introduces complexity in geometry: the topology of the domain of the problem changes when bodies contact or fracture. We do not yet have geometrical representations which are robust enough to handle these challenges.

```
def fu(self, x, dotx, t):

    # Scatter from x to components
    Vector_scatter(self.xu, self.x, self.dotxu_sc)
    Vector_scatter(self.xv, self.x, self.dotxv_sc)
    Vector_scatter(self.xF, self.x, self.dotxF_sc)
    Vector_scatter(self.xSigma, self.x, self.dotxSigma_sc)

    # Mesh
    Elasticity_deform(self.mesh(), self.U)

    # Compute B = FF^T
    Elasticity_computeB(self.xF, self.xB,
                        self.F.element(),
                        self.mesh())

    # Mass matrix
    Elasticity_computemsigma(self.msigma, self.Sigma.element(),
                             self.mesh())

    # U
    self.dotxu.copy(self.xv)

    # V
    assemble(self.L(), self.dotxv, self.mesh())
    self.dotxv.div(self.m)

    # Sigma
    assemble(self.Lstress, self.dotxSigma, self.mesh())
    self.dotxSigma.div(self.msigma)
    self.xEpsilon.copy(self.dotxSigma)

    # F
    assemble(self.LF, self.dotxF, self.mesh())
    self.dotxF.div(self.msigma)

    # Gather components into dotx
    Vector_gather(self.dotxu, self.dotx, self.dotxu_sc)
    Vector_gather(self.dotxv, self.dotx, self.dotxv_sc)
    Vector_gather(self.dotxF, self.dotx, self.dotxF_sc)
    Vector_gather(self.dotxSigma, self.dotx, self.dotxSigma_sc)
```

FIGURE 5. Python code for the FEniCS elasticity solver (right hand side of ODE).

A *mass element* e is a set of parameters $\{x,\ v,\ m,\ r,\ C\}$:

x	:	current coordinate
v	:	velocity
m	:	mass
r	:	radius
C	:	a set of spring connections

A *spring connection* c is a set of parameters $\{e_1,\ e_2,\ k,\ b,\ l,\ d\}$:

e_1	:	the first mass element connected to the spring
e_2	:	the second mass element connected to the spring
κ	:	Hooke spring constant
b	:	damping constant
l_0	:	rest distance
l_f	:	fracture distance

TABLE 1. Descriptions of the basic elements of the mass–spring model: mass elements and spring connections.

Until we have such representations, we use the mass-spring model to model contact. We represent nodes on the surface of the computational mesh as masses in the mass-spring model, and then simply use the mass-spring model to compute forces on the masses.

This leads to the following discrete contact model:

$$\frac{DV}{Dt} = M_{FEM}^{-1}(B\Sigma + F_{body}) + M_{MS}^{-1}F_{contact} \tag{13}$$

where M_{FEM} is the mass matrix resulting from a FEM discretization of the PDE model and M_{MS} is the mass matrix resulting from the mass-spring model.

9. Simulation Examples

We present some simulation examples of the elastoplastic models. In figure 6 a cow object and a block are thrown in a room, displaying elastic, viscous, contact and friction phenomena. In figure 7 a beam is fixed at one end, and an initial velocity deforms the beam. The beam is plastic and the deformation is permanent, illustrating the plastic stress rate Euler-Almansi model.

See [3] for more mature and larger-scale examples.

10. Performance comparison

10.1. **Experimental setup.** We let an implementation of the mass-spring model (part of Ko in **FEniCS**) represent the minimal amount of work needed to model systems of elastic bodies under large deformation. It is the industry standard for high-performance elastic simulation (see [4] and [8] for a survey) and serves as the benchmark. We then make

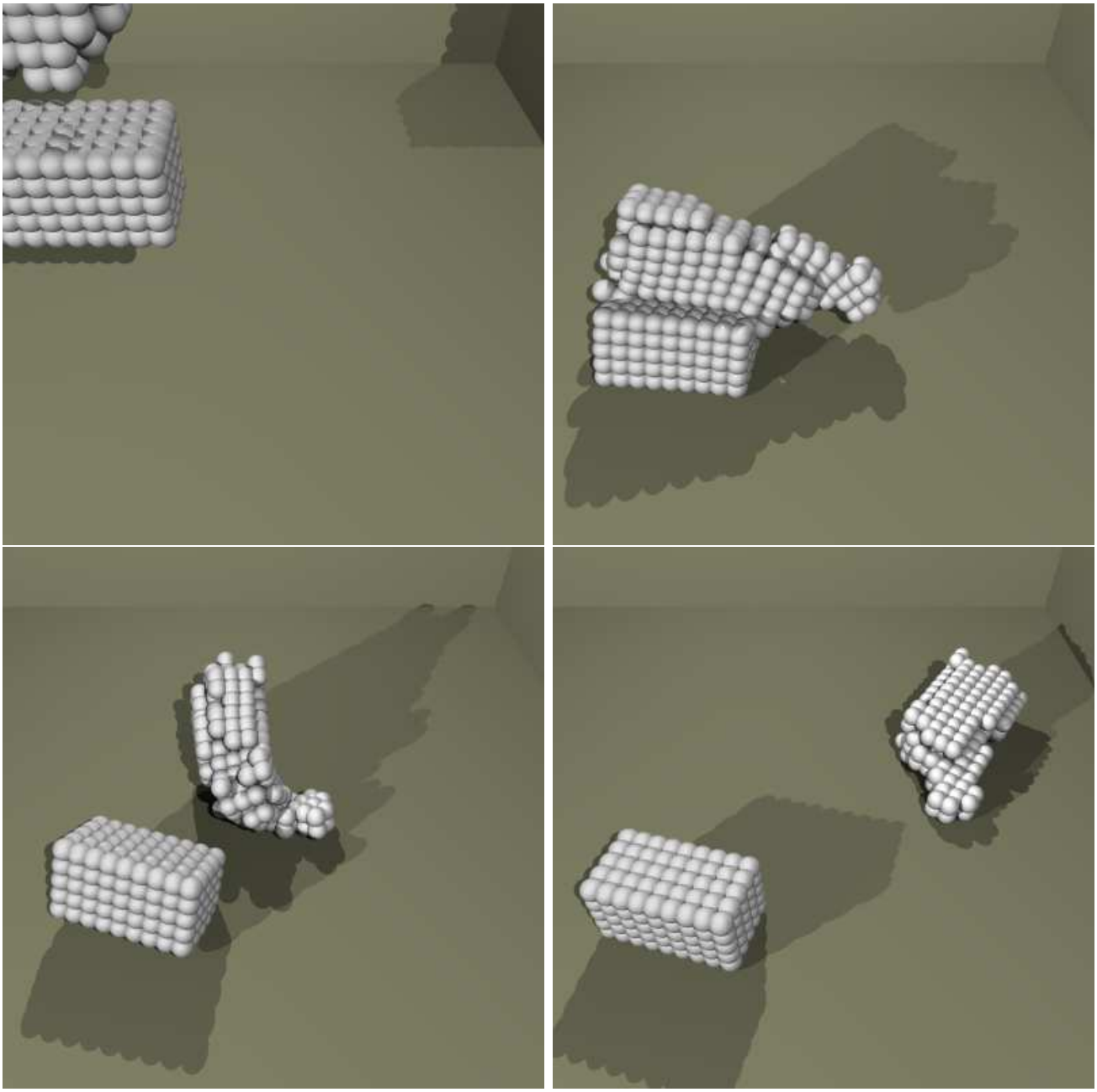


FIGURE 6. Cow and a block thrown in a room, illustrating a solution of the rate form of the Euler-Almansi model. The spheres on the surface of the bodies are the representation of the contact model.

a comparison with an automated implementation (**DOLFIN** in **FEniCS**) of the FEM discretization of the elastic PDE (9).

Both implementations perform the task of assembling a discrete (in space) system representing their corresponding equations. In this case the discrete system is the ODE:

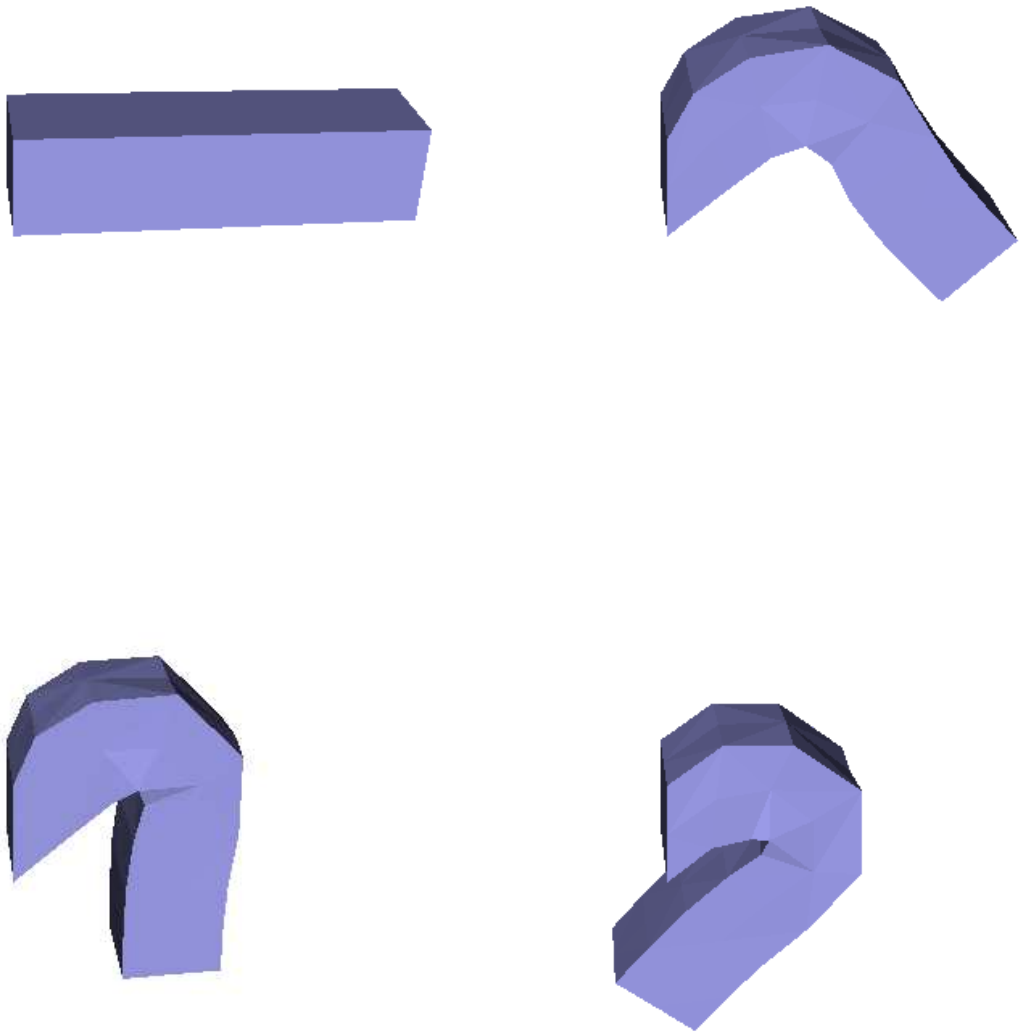


FIGURE 7. Bent beam simulated using the plastic stress rate model. The lower right image is the permanently deformed rest state resulting from an initial velocity.

$$\frac{du}{dt} = f(t, u) \tag{14}$$

This means the task is computing $f(t, u)$ for a given t and u. The ODE is then discretized by the same method, so that is the same amount of work for both models.

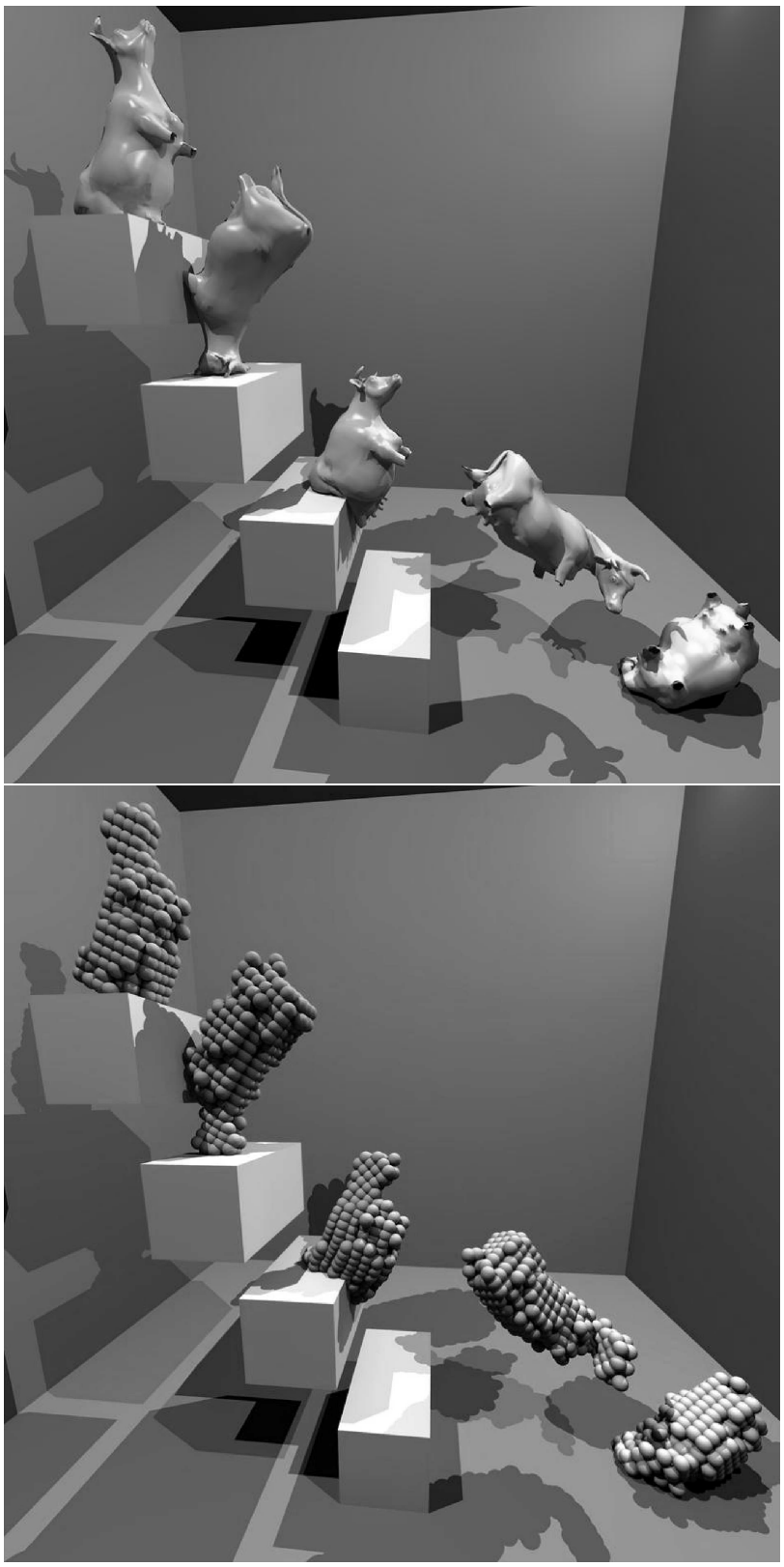


FIGURE 8. Ko simulation of cow falling down stairs. Here we map the deformation from the mechanics representation to the original boundary representation.

The mass-spring assembly and the FEM assembly both use PETSc for linear algebra operations. We also use the same operating system, compiler and optimization flags. The source code for both implementations is published and available at the **FEniCS** website [3].

The experimental setup consists of a unit cube discretized into cells (tetrahedrons). We vary the number of cells in the discretized cube and measure the time to compute $f(t, u)$.

We compare two versions of the PDE: one fully elastic and one viscoelastic.

10.2. **Results.** The results can be seen in plots 10.2 and 10.2. The fully elastic PDE is a factor 2-3 slower and the viscous PDE is a factor 3-4 slower than the mass-spring model.

The fully elastic PDE performs very well. There is still work to be done in making **FEniCS** as a whole more efficient (see performance analysis below), mesh representation for example. We carry out a performance analysis of the implementation below to find out which parts of the implementation are taking the most execution time.

Performance analysis (see figures 11 and 12) tells us that the form evaluation does not dominate the total execution time, which means that the automated discretization in **FEniCS** lives up to its goals. The assembly does dominate the total execution time, but this is expected, since that's what the benchmark consists of. The PDE model spends a considerable time in primitive mesh functions, which means we should find or design a more efficient mesh interface and representation.

With a more efficient mesh representation, we should be able to reach a factor 1-2. We cannot expect better if the mass-spring model is representative of the minimal amount of work.

The PDE model allows space adaptivity, while the mass-spring model does not. This will give the PDE model an efficiency improvement of many factors, depending on the problem parameters. This would mean that the PDE model with space adaptivity is more efficient than the mass-spring model.

10.3. **Performance analysis.** Profiling information (generated by the `gprof` profiler) of the PDE and mass-spring implementations can be seen in figures 11 and 12 respectively. The flat list gives execution time spent in individual functions while the call graph gives execution time spent in functions including their children.

References

[1] G. Debunne, M. Desbrun, M.-P. Cani, and A. H. Barr, *Dynamic real-time deformations using space and time adaptive sampling*, in SIGGRAPH 2001 Computer Graphics Proceedings, ACM Press / ACM SIGGRAPH, 2001, pp. 31–36.

[2] Free Software Foundation, *GNU GPL*. `http://www.gnu.org/copyleft/gpl.html`.

[3] J. Hoffman, J. Jansson, C. Johnson, M. Knepley, R. C. Kirby, A. Logg, and L. R. Scott, *FEniCS*. `http://www.fenics.org/`.

[4] J. Jansson and J. S. M. Vergeest, *A discrete mechanics model for deformable bodies*, Computer-Aided Design, 34 (2002).

[5] R. C. Kirby, *FIAT: A new paradigm for computing finite element basis functions*, ACM Trans. Math. Software, 30 (2004), pp. 502–516.

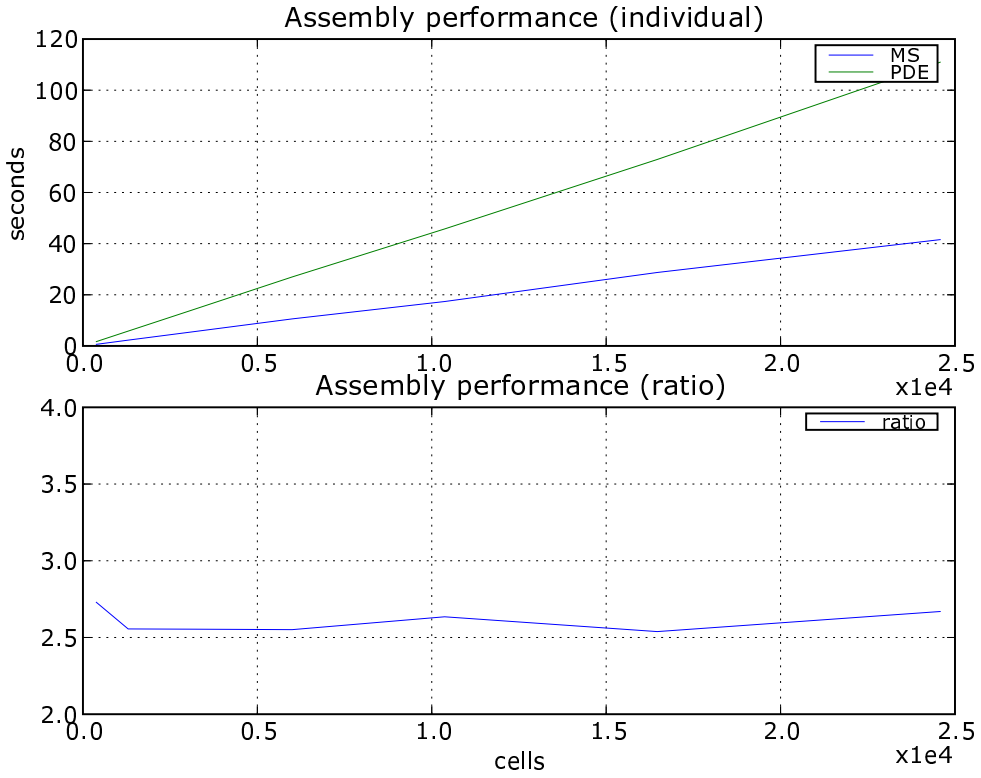


FIGURE 9. Plot of execution time against number of cells for the elastic PDE model (9) versus the mass-spring model.

[6] R. C. KIRBY AND A. LOGG, *A compiler for variational forms*. submitted to ACM Trans. Math. Softw., 2005.

[7] R. C. KIRBY, A. LOGG, L. R. SCOTT, AND A. R. TERREL, *Topological optimization of the evaluation of finite element matrices*. submitted to SIAM J. Sci. Comput., 2005.

[8] A. LIU, F. TENDICK, K. CLEARY, AND C. KAUFMANN, *A survey of surgical simulation: Applications, technology and edudcation*, Presence, 12 (2003).

[9] A. LOGG, *Automation of Computational Mathematical Modeling*, PhD thesis, Chalmers University of Technology, Sweden, 2004.

[10] J. C. SIMO AND T. J. R. HUGHES, *Computational Inelasticity*, Springer-Verlag, 2000.

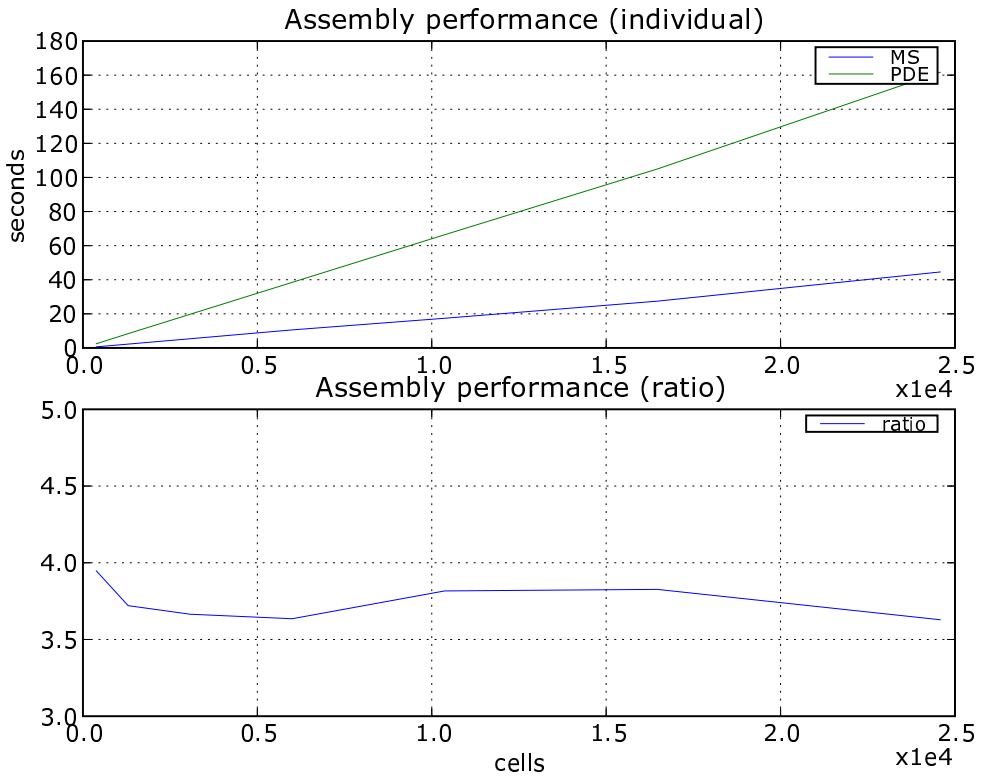


FIGURE 10. Plot of execution time against number of cells for the viscoelastic PDE model (9) versus the mass-spring model.

```
Flat list:

time(%) Name
 33.24  dolfin::ElasticityDirect::LinearForm::eval(...)
  7.90  dolfin::AffineMap::updateTetrahedron(...)
  6.53  dolfin::DiscreteFunction::interpolate(...)
  5.47  dolfin::PArray<Vertex*>::operator()(int)
  5.18  dolfin::Cell::vertexID(int)
  3.92  dolfin::GenericCell::vertexID(int)

Call graph:

time(%) Name
 80.8   dolfin::FEM::assemble_common(...)
 33.2   dolfin::ElasticityDirect::LinearForm::eval(...)
 22.3   dolfin::Form::updateCoefficients(...)
 14.8   dolfin::ElasticityUpdatedUtil::computeB(...)
```

FIGURE 11. Extract from profiling information: flat list and call graph of PDE implementation.

```
Flat list:

time(%) Name
 22.90 ko::BasicEulerIntegrator::residualdG0()
 22.18 ko::SimpleVector::norm() const
 18.77 ko::MechanicsModel::force(...)
 11.97 ko::SimpleVector::axpy(...)

Call graph:

time(%) Name
 99.9 ko::BasicEulerIntegrator::residualdG0(...)
 49.8 ko::MechanicsModel::force(...)
```

FIGURE 12. Extract from profiling information: flat list and call graph of mass-spring implementation.