Computational Mechanics (2022) 69:589–613
https://doi.org/10.1007/s00466-021-02105-2

ORIGINAL PAPER

AutoMat: automatic differentiation for generalized standard materials on GPUs

Johannes Blühdorn[1] · Nicolas R. Gauger[1] · Matthias Kabel[2]

Received: 9 October 2020 / Accepted: 1 October 2021 / Published online: 5 November 2021

Abstract
We propose a universal method for the evaluation of generalized standard materials that greatly simplifies the material law implementation process. By means of automatic differentiation and a numerical integration scheme, *AutoMat* reduces the implementation effort to two potential functions. By moving AutoMat to the GPU, we close the performance gap to conventional evaluation routines and demonstrate in detail that the expression level reverse mode of automatic differentiation as well as its extension to second order derivatives can be applied inside CUDA kernels. We underline the effectiveness and the applicability of AutoMat by integrating it into the FFT-based homogenization scheme of Moulinec and Suquet and discuss the benefits of using AutoMat with respect to runtime and solution accuracy for an elasto-viscoplastic example.

Keywords Automatic differentiation · Generalized standard materials · Numerical methods for ODEs · FFT-based homogenization · GPU computing

Mathematics Subject Classification G.1.4 · G.1.7 · G.4 · J.2

1 Introduction

In recent years, the improving quality of micro x-ray computed tomography (CT) images led to a digitalization of the material characterization process for composites. Nowadays, standard CT-devices have a maximum resolution below one μm and produce 3D images of up to 4096^3 voxels. This permits a detailed view of the microstructure's geometry of composite materials up to the point where continuum approaches are still reasonable. In the context of material characterization, the physical description of the body leads to a partial differential equation (PDE) in which the behavior of the material itself is modeled in terms of a material law. Traditionally, a finite element (FEM) discretization is applied, and during the solution procedure, the material law is evaluated locally at quadrature points. To solve problems of this size with conventional FEM, large computing clusters are required to handle the global stiffness matrices [3,4].

In the last two decades, the FFT-based homogenization scheme of Moulinec and Suquet [40,41] emerged as a memory efficient matrix-free alternative that was adapted to operate on structured finite element meshes [33,54,55,65]. Besides the small memory footprint, the most favorable property of the so-called *basic scheme* is a tangent-free treatment of nonlinear material behavior. However, its required iteration count is proportional to the material contrast, i. e. the maximum of the quotient of the largest and the smallest eigenvalue of the consistent tangential stiffness field. Thus, for certain practical applications such as the homogenization of plastifying materials, the convergence behavior can be exceedingly slow [53].

To accelerate the solution process, Zeman et al. [67] and Brisard and Dormieux [8,9] applied Krylov-subspace solvers to FFT-based homogenization. These methods are extremely fast, but they are restricted to linear problems. By combination with inexact Newton-methods, they were extended to the physically [18] and geometrically [27] nonlinear case

✉ Johannes Blühdorn
johannes.bluehdorn@scicomp.uni-kl.de

Nicolas R. Gauger
nicolas.gauger@scicomp.uni-kl.de

Matthias Kabel
matthias.kabel@itwm.fraunhofer.de

1 Chair for Scientific Computing, Technische Universität Kaiserslautern, Kaiserslautern, Germany

2 Department of Flow and Material Simulation, Fraunhofer ITWM, Kaiserslautern, Germany

and exhibited excellent performance [34,35]. The drawback of this approach consists in either loosing the small memory footprint or the need to calculate the tangential stiffness of the material laws in every iteration of the linear solver. Furthermore, the analytic derivation of the tangent can be tedious and its implementation may require considerable programming effort, and is thus prone to errors. This gave rise to applying Quasi-Newton methods in FFT-based micromechanics [10,11,52,53,57,64]. There, material tangents are replaced by suitable approximations. To sum up, the choice of the solver is driven by compromises between runtime efficiency, memory efficiency and the implementational effort of an accurate material tangent.

Especially during prototyping and modeling, it might be necessary to assess different material laws. Clearly, it is impractical to derive the material tangent from scratch for every material law under consideration. However, it is also undesirable to be restricted to tangent-free solvers during this phase. Motivated by the work of Rothe and Hartmann [45], we started the development of *AutoMat*, which leverages automatic differentiation and GPU computing to simultaneously address issues of flexibility, accuracy and performance.

GPU acceleration for FEM codes often uses the well-known coloring method which partitions the mesh such that no elements of the same color share any common node. The GPU kernel then performs elemental stiffness calculations for the elements of the same color in parallel and assembles them into the global stiffness matrix. In [36], the results show a speedup of 3 to 19 for the matrix assembly on an Nvidia GeForce 8800 GTX card. The authors of [42] additionally accelerated the sparse matrix solver and could reduce the execution time of the entire FEM application from 44.65 s on a CPU (Nehalem architecture, 4 cores, OpenMP) to 17.52 s on a CPU with a GPU (Nvidia Tesla C2050). Recently, [30] made use of the warp shuffle feature of CUDA to optimize numerical integration for the evaluation of elemental stiffness matrices. For different mesh sizes, a speedup between 6.73 and 8.21 was demonstrated on an Nvidia Tesla K40 card.

Automatic differentiation (AD) refers to techniques for the automatic acquisition of machine accurate derivatives of computer codes [19]. These have applications in, e. g., the setup of adjoint solvers [50], parameter identification [6], shape optimization [17], and machine learning [21]. There, AD is applied to a full simulation. Here, we use AD locally for the automatic setup of solvers and eliminate the inconvenience of hand-computed derivatives. For classical CPU architectures, several mature AD tools are available by now, for example ADOL-C [63], dco/c++ [31] and CoDiPack [47]. Advances in the direction of AD for GPU codes are more recent, examples include dco/map with applications in computational finance [32]. In [45], Rothe and Hartmann use the source transformation tool OpenAD [60] for the automatic computation of material tangents and the assembly of Jacobians for implicit solvers in the context of a multi-level Newton algorithm. In this work, the automatic differentiation ansatz is advanced in several directions.

We focus on the class of generalized standard materials (GSM) [24], which we introduce in Sect. 2. There, AD enables us to recover the constitutive equations of the material law automatically from given implementations of two potentials, resulting in a fully automatic solver setup. This allows for a highly usable and convenient integration of GSMs into mechanical solvers. We demonstrate this by integrating AutoMat into the FFT-based homogenization scheme of Moulinec and Suquet [41] as implemented in FeelMath[1]. As our benchmark example for AutoMat, we use an elasto-viscoplastic material model with material parameters adjusted to measurements of a metal-matrix composite. The precise setup is taken from Michel and Suquet [37] and summarized in Sect. 3.

The consistent tangent operator is the algorithmic derivative of the stress as it is computed from the strain according to the material law. Its computation requires a differentiation through an integration scheme for ordinary differential equations (ODEs). The conventional backward Euler step is differentiated in [58] by hand. We show in Sect. 4.1 that this procedure can be fully automatized. To understand the numerical properties of the tangent computation, we interpret it in Sect. 4.2 as a single implicit Euler step applied to an ODE for the derivative. Since this ODE depends on the chosen loading step size, convergence of the tangent for decreasing step size is not guaranteed. This motivated us to explore schemes with adaptive time steps instead. It was noted in [13] that the differentiation of ODE integration schemes in a blackbox manner, that is, without consideration of the structure of the integration algorithm and its approximative nature, usually leads to incorrect or inaccurate derivatives. A particular focus of [13] is on the role of adaptive time steps and an aposteriori error correction. In [62], fixed step size explicit Runge–Kutta methods are applied for the discretization of optimal control problems and it is shown that the sensitivities obtained by blackbox forward AD are consistent with the corresponding tangent linear model. [15] reports on an application of AD to explicit Runge–Kutta methods with adaptive step size control. Further research on AD of ODE integration schemes was conducted in an optimal control context with a focus on the reverse mode of AD [48,49], including adaptive step sizes [2] and reverse mode specifics such as interpolation strategies for the forward solution [1]. In this work, we refine forward differentiation strategies for a class of integration schemes that contains Rosenbrock methods as well as both explicit and implicit Runge–Kutta schemes. We show that two modifications suggested in [13] together with an appropriate treatment of equation solves are sufficient to

[1] https://www.itwm.fraunhofer.de/feelmath.

turn blackbox forward differentiation into a procedure that is equivalent to applying the same scheme simultaneously to an ODE for the derivatives. With the results, we can guarantee that the consistent tangent operator is as accurate as the stress tensor. The overall robustness and accuracy of the proposed scheme is assessed in Sect. 4.3. We achieve further robustness with respect to the choice of the ODE solver by employing a stress-driven error control, which we present in Sect. 4.4.

In FFT-based homogenization, the computationally costly simulation components are the Fourier transform and material law evaluation [20]. For nonlinear materials, the latter tends to dominate the overall runtime [29] and is hence performance critical. The spatial independence of material law evaluations allows for parallelization, which is typically used in an efficient implementation. Throughout Sect. 4, we compare parallelized material law evaluations on the CPU with GPU accelerated material law evaluation. We achieve a notable speedup for conventional material law evaluation, but particularly for the computationally more involved automatic evaluation strategies presented in this paper, there are significant performance gains. In our example and setup, we were able to close the performance gap between conventional material law evaluation on the CPU and automatic material law evaluation on the GPU. The good performance would not be possible without an efficient implementation of automatic differentiation on the GPU. Therefore, we developed an operator overloading AD tool specifically for the application presented in this paper. It is based on expression template techniques; previously in AD, these were successfully applied for the treatment of right hand sides in the forward mode [43] and in Jacobi taping [26] as well as primal value taping [46] in the reverse mode. The details of the implementation and its further optimizations are presented in Sect. 5. In Sect. 6, remaining influence factors on the performance are discussed. We analyze the performance limiters of AutoMat, present design choices and optimizations of the GPU implementation and discuss overlap of CPU workloads, GPU workloads, and data exchange as well as reductions of the memory footprint. This is complemented by scaling studies in Sect. 7.

Finally, we summarize and conclude our work in Sect. 8.

2 Generalized standard materials

The notion of generalized standard materials is originally introduced in [24]; a compact introduction to the subject can be found in [37]. Let $\varepsilon \in \mathbb{R}^6$ denote the right Cauchy-Green strain tensor[2], $\sigma \in \mathbb{R}^6$ the Cauchy stress $tensor^2$ and $a \in \mathbb{R}^m$ the vector of internal variables, all depending on time and space. The constitutive equations of the material law are given in terms of a Helmholtz free energy density $\omega\colon \mathbb{R}^6 \times \mathbb{R}^m \to \mathbb{R}\colon (\varepsilon, a) \mapsto \omega(\varepsilon, a)$ and a force potential $\Psi : \mathbb{R}^m \to \mathbb{R}\colon A \mapsto \Psi(A)$ and read

$$\sigma = \frac{\partial \omega}{\partial \varepsilon}(\varepsilon, a), \tag{1}$$

$$\dot{a} = \frac{\partial \Psi}{\partial A}\left(-\frac{\partial \omega}{\partial a}(\varepsilon, a)\right). \tag{2}$$

$A \in \mathbb{R}^m$ is referred to as generalized stresses and if both ω and Ψ are convex functions of their arguments, we speak of a generalized standard material. The dissipation potential which is the convex dual of Ψ is not used in the present study. We use these identifiers throughout the paper.

After space discretization, evaluations of above stress-strain relationship and evolution of internal variables are required in the quadrature points. We drop the x dependency in the notation as the specific location does not change throughout a single material law evaluation. After time discretization, the material law inputs at a quadrature point consist of a strain tensor ε_n and internal variables a_n at time t_n as well as a strain tensor ε_{n+1} which is usually only a prediction of the actual strain tensor at time t_{n+1} in the context of the surrounding elasticity solver. Then, in each quadrature point, the material law can be evaluated as follows.

1. Solve the ODE for the internal variables (2) with initial data (t_n, a_n) on the time interval $[t_n, t_{n+1}]$. Recover $\varepsilon(t)$ by means of linear interpolation between ε_n and ε_{n+1}. This way, obtain a_{n+1}.
2. Compute σ_{n+1} via (1) from ε_{n+1} and a_{n+1}.

Additionally, the consistent tangent operator $C_{n+1} \in \mathbb{R}^{6\times 6}$ which is the algorithmic derivative of σ_{n+1} with respect to ε_{n+1} is usually computed along with the material law [59]. It is used in the FFT-based homogenization scheme to determine the optimal reference material.

In view of the decision for an integration scheme for (2), negative eigenvalues of the Jacobian of the ODE's right hand side indicate that explicit solvers might display unstable behaviour [22], that is, require extremely small steps. The following theorem states that evolution equations arising from GSMs are subject to this issue. Following [66], $M \in \mathbb{C}^{m\times m}$ is called positive semi-definite if $\forall x \in \mathbb{C}^m\colon x^*Mx \in \mathbb{R}$ and $x^*Mx \geq 0$. This definition implies that each positive semi-definite complex matrix is Hermitean.

Theorem 1 *Let a GSM be specified by ω and Ψ and assume that both are C^2. If λ is an eigenvalue of the Jacobian with respect to a of the right hand side of (2), then $\lambda \in \mathbb{R}$ and $\lambda \leq 0$.*

[2] Note that the Voigt notation [61] for symmetric second order tensors $\in \mathbb{R}^{3\times 3}$ is applied.

Proof The Jacobian with respect to a of the right hand side of (2) reads

$$\frac{\mathrm{d}}{\mathrm{d}a}\left(\frac{\partial\Psi}{\partial A}\left(-\frac{\partial\omega}{\partial a}(\varepsilon, a)\right)\right) = -\frac{\partial^2\Psi}{\partial A^2}\left(-\frac{\partial\omega}{\partial a}(\varepsilon, a)\right)\frac{\partial^2\omega}{\partial a^2}(\varepsilon, a). \tag{3}$$

As Hessians of C^2 functions, both $\frac{\partial^2\Psi}{\partial A^2}$ and $\frac{\partial^2\omega}{\partial a^2}$ are symmetric. Since both ω and Ψ are convex and C^2, $\frac{\partial^2\Psi}{\partial A^2}$ and $\frac{\partial^2\omega}{\partial a^2}$ are also positive semi-definite as real matrices, that is,

$$\forall x \in \mathbb{R}^m : x^{\mathrm{T}}Mx \geq 0,$$

where M denotes any of both Hessians. Symmetric and positive semi-definite real matrices are also positive semi-definite as complex matrices. By Theorem 2.2 in [66], the product of positive semi-definite complex matrices is similar to a positive semi-definite complex matrix, that is, there exists an invertible complex matrix T such that $T^{-1}\frac{\partial^2\Psi}{\partial A^2}\frac{\partial^2\omega}{\partial a^2}T$ is a positive semi-definite complex matrix. All eigenvalues of a positive semi-definite complex matrix lie in $\mathbb{R}_{\geq 0}$. As similarity preserves eigenvalues, all eigenvalues of the product $\frac{\partial^2\Psi}{\partial A^2}\frac{\partial^2\omega}{\partial a^2}$ are contained in $\mathbb{R}_{\geq 0}$; hence all eigenvalues of (3) are contained in $\mathbb{R}_{\leq 0}$. □

Another example for an eigenvalue proof based on definiteness and convexity in the context of material simulation can be found in [12]. There, a time-marching scheme for the solution of a viscoplastic problem is identified as a system of ODEs for the stresses at integration points and the eigenvalues of the Jacobian of the right hand side are used to assess stability properties.

Whether explicit solvers (with adaptive step size control) or implicit solvers are faster depends on the specific material law, internal variable values, applied strain and integration interval length. In Sect. 4, we refine both explicit and implicit solution strategies.

3 Example

Throughout the paper at hand, we perform our numerical studies for a uni-axial tension-compression test of a short fiber reinforced metal-matrix composite (MMC) taken from [37].

Microstructure The MMC consists of 10.2 vol% Al203 fibers embedded in an aluminum matrix. In our periodically generated micro-structure (see Fig. 1), the planar isotropic distributed fibers have a diameter of 9 μm and a length of 135 μm. This volume element of $150 \times 150 \times 150\mu\mathrm{m}^3$ was discretized by $150 \times 150 \times 150$ voxels.

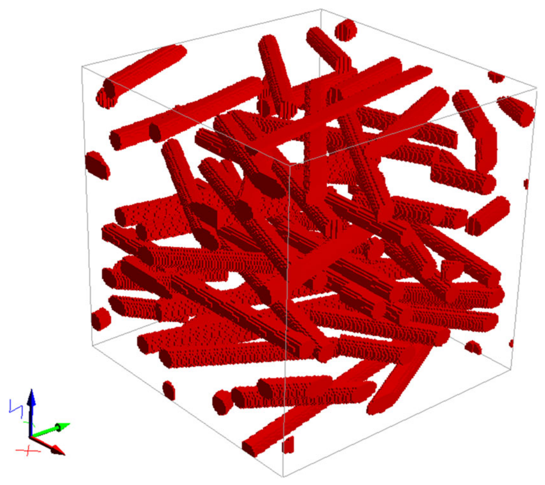


Fig. 1 Microstructure of the MMC generated with GeoDict (https://www.geodict.com/)

Material Model The Al203 fibers are modeled linear elastic with Young's modulus E and Poisson's ratio ν and the aluminum matrix as the elasto-viscoplastic GSM given by the Helmholtz free energy density

$$\omega(\varepsilon, a) = \omega(\varepsilon, \varepsilon_{vp}, \alpha) = \frac{1}{2}\varepsilon_{\mathrm{e}}^T C^{\mathrm{e}}\varepsilon_{\mathrm{e}} + \frac{1}{3}\varepsilon_{vp}^T \mathrm{H}\varepsilon_{vp} + \int_0^{\alpha} \mathcal{K}(q)\,\mathrm{d}q, \tag{4}$$

where $\mathrm{H} = \mathrm{diag}\left(H, H, H, \frac{H}{2}, \frac{H}{2}, \frac{H}{2}\right)$ and $\varepsilon_{\mathrm{e}} = \varepsilon - \varepsilon_{vp}$, and the force potential[3]

$$\Psi(A) = \Psi(A_{vp}, A_\alpha) = \frac{\sigma_d\dot{\varepsilon}_0}{n+1}\left(\frac{\left(\left\|\mathrm{dev}\, A_{vp}\right\|_{\mathrm{eq}} + A_\alpha\right)^+}{\sigma_d}\right)^{n+1} \tag{5}$$

with viscoplastic strain ε_{vp} and equivalent plastic strain α as internal variables. C^{e} is an elastic stiffness matrix given in terms of a second (E, ν) pair and $\mathcal{K}(\alpha)$ describes the isotropic hardening and H the (linear) kinematic hardening, whereas the viscous effects are given by the drag stress σ_d, the rate sensitivity n and the reference strain rate $\dot{\varepsilon}_0$. For computational efficiency, the Voigt notation [61] is used for strain and stiffness tensors.

For the studied example, the nonlinear parameters of the aluminum matrix were calibrated without isotropic hardening, i. e. $\mathcal{K}(\alpha)$ was assumed to be equal to the initial yield stress σ_Y, $\mathcal{K}(\alpha) \equiv \sigma_Y$. The complete set of material parameters is reproduced in Table 1.

[3] For $n > 1$, Ψ is C^2. Note that in later parts of this paper, n denotes the time step index.

Table 1 Parameters for elasto-viscoplastic GSM [37]

Parameter	Unit	Aluminum	Al203
E	GPa	55	300
ν	1	0.33	0.25
σ_Y	MPa	25	–
H	GPa	1.8	–
$\dot{\varepsilon}_0$	1/s	1	–
σ_d	MPa	130	–
n	1	3.6	–

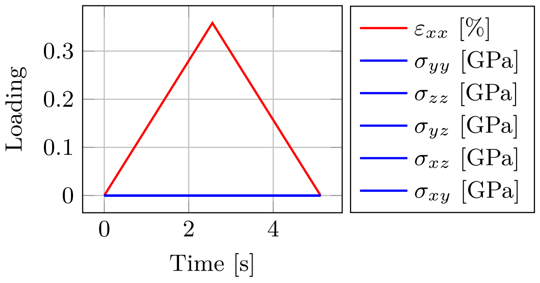


Fig. 2 Mixed boundary conditions [28] for the uni-axial experiment. (Color figure online)

Boundary Conditions As described in detail by Michel and Suquet, the volume element is submitted to a uni-axial tension-compression test at constant strain rate with alternating sign in loading direction,

$$\dot{\varepsilon}_{xx} = \pm 1.4 \cdot 10^{-3}\,\mathrm{s}^{-1},$$
$$-3.48441 \cdot 10^{-3} \le \varepsilon_{xx} \le 3.58454 \cdot 10^{-3}.$$

For our studies, we use the tension part that is displayed in Fig. 2. The loading path is discretized in an equidistant manner with a granularity between 20 and 320 steps. *If not mentioned otherwise, 80 loading steps are used.*

Material Law Evaluations In each loading step, a stationary elastic problem is solved by FFT-based homogenization [41]. This method is relying on an FFT-based preconditioner [27] defined by the constant coefficient linear elastic problem div $\left(C^{\mathrm{ref}}\varepsilon\right) = 0$, where C^{ref} is called the *reference stiffness* and has to be chosen depending on the locally varying tangential stiffness of the material laws [27]. The reference stiffness can be either fixed at the beginning of the time dependent simulation by using only the initial elastic stiffness of the material laws or it can be adjusted in each loading step to the current tangential stiffness to reduce the number of iterations necessary for convergence. In the first case, this involves one material law evaluation per voxel *with* tangent at the beginning of the initial loading step and in the latter case at the beginning of each loading step. The (matrix-free) FFT-based solver itself only performs one material law evaluation *without* tangent per iteration and voxel. The performance impact of the reference material setup prior to the first loading step is negligible; therefore, *whenever we display time spent on material law evaluations with tangent, the configuration at hand updates the reference material.* Then, material law evaluations with and without tangent are timed separately. In order to indicate the impact on the overall simulation time, we usually display the *accumulated* time spent on material evaluations with and without tangent, respectively, during one run of the simulation. We use the types of error control explained in Sect. 4.4 throughout.

Parallelization We perform our tests on a dual-socket cluster node with two Intel Xeon Gold 6132 processors (Skylake architecture) at 2.6 GHz (2 × 14 cores) and an Nvidia V100 graphics card (Volta architecture, 5120 CUDA cores). As this card has uncapped double precision performance, we keep the elasticity solver's double precision also for material law evaluations on the GPU. Nonetheless, single precision seems to work well for the material law presented above. This is of importance on GPUs without good double precision performance, and can also speed up computations in general; especially material law evaluations with tangent seem to benefit performance-wise from single precision. We use OpenMP[4] for CPU parallelization; on the graphics card, CUDA[5] is used. We used version 9.0 of the CUDA SDK. Details on the computational layout can be found in Sect. 6.

4 Automatic evaluation

Conventionally, efficient methods for the evaluation of specific material laws are derived by hand. For example, GSMs such as (4), (5) are discretized in Chapter 3 of [58] by means of a single backward Euler step. With the help of an explicit formula for the flow direction, the resulting nonlinear system of equations is reduced to a scalar equation that is then solved by Newton's method. For the computation of C_{n+1}, the derivative of the corresponding nonlinear equation solve is recovered in an implicit function theorem fashion. Numerical integration and algorithmic differentiation are both carried out by hand. We refer to this approach as *conventional evaluation strategy*—it is material law specific. For our performance studies, it serves as a baseline. In this work, we explore several flavours of the *automatic evaluation strategy* depicted in Fig. 3 that relies on AD to evaluate the various partials of ω and Ψ, to assemble Jacobians (usually of size $\mathbb{R}^{m\times m}$) as required for ODE integration schemes and finally, to compute the material tangent C_{n+1}, which involves a differentiation of the whole algorithm depicted in Fig. 3.

[4] https://www.openmp.org/.

[5] https://developer.nvidia.com/cuda-zone.

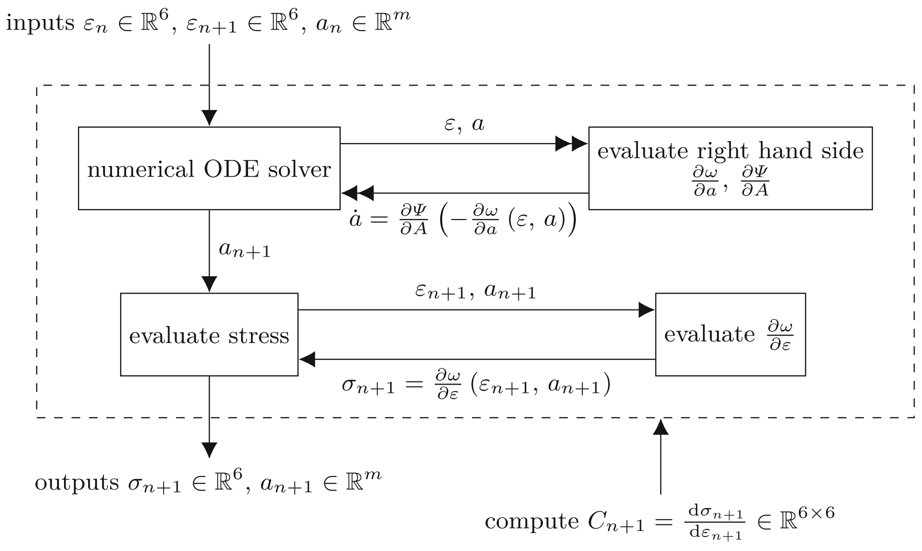

inputs $\varepsilon_n \in \mathbb{R}^6$, $\varepsilon_{n+1} \in \mathbb{R}^6$, $a_n \in \mathbb{R}^m$
numerical ODE solver
ε, a
evaluate right hand side $\frac{\partial\omega}{\partial a}$, $\frac{\partial\Psi}{\partial A}$
$\dot{a} = \frac{\partial\Psi}{\partial A}\left(-\frac{\partial\omega}{\partial a}(\varepsilon, a)\right)$
a_{n+1}
evaluate stress
ε_{n+1}, a_{n+1}
evaluate $\frac{\partial\omega}{\partial\varepsilon}$
$\sigma_{n+1} = \frac{\partial\omega}{\partial\varepsilon}(\varepsilon_{n+1}, a_{n+1})$
outputs $\sigma_{n+1} \in \mathbb{R}^6$, $a_{n+1} \in \mathbb{R}^m$
compute $C_{n+1} = \frac{\mathrm{d}\sigma_{n+1}}{\mathrm{d}\varepsilon_{n+1}} \in \mathbb{R}^{6\times 6}$

Fig. 3 Automatic evaluation strategy

The strategy can easily be adapted to other material laws by exchanging the implementations of the potentials. We also explore the performance benefits of providing hand-derived implementations of the partials of ω and Ψ for an otherwise automatic evaluation; we refer to this as *semi-automatic evaluation strategy*.

4.1 Single implicit Euler step

AD allows us to turn the conventional scheme from [58] into an automatic evaluation strategy that is not specific to a certain GSM and requires only implementations of ω and Ψ. Let $h = t_{n+1} - t_n$ be the loading step size and

$$f(\varepsilon, a) = \frac{\partial\Psi}{\partial A}\left(-\frac{\partial\omega}{\partial a}(\varepsilon, a)\right),$$

that is, the right hand side of the ODE (2). An application of a single implicit Euler step yields

$$a_{n+1} = a_n + h \cdot f(\varepsilon_{n+1}, a_{n+1}),$$

that is, the nonlinear system of equations

$$\underbrace{a_{n+1} - h \cdot f(\varepsilon_{n+1}, a_{n+1}) - a_n}_{=: F(\varepsilon_{n+1}, a_{n+1})} = 0 \tag{6}$$

for a_{n+1}, which we solve with Newton's method. We initialize $a_{n+1}^{(0)} = a_n$ and iterate $a_{n+1}^{(k+1)} = a_{n+1}^{(k)} - \Delta a_{n+1}^{(k)}$ where

$$\Delta a_{n+1}^{(k)} = \left(\frac{\partial F}{\partial a}\left(\varepsilon_{n+1}, a_{n+1}^{(k)}\right)\right)^{-1} F\left(\varepsilon_{n+1}, a_{n+1}^{(k)}\right).$$

The application of AD is twofold. Each evaluation of f (or F) involves evaluations of the partials $\frac{\partial\Psi}{\partial A}$ and $\frac{\partial\omega}{\partial a}$. This can be automatized by first order automatic differentiation. Second, the evaluations of the Jacobian $\frac{\partial F}{\partial a}$ can be realized likewise by AD but require — due to the already involved partials — an additional derivative order. Note $\frac{\partial F}{\partial a} = I - h \cdot \frac{\partial f}{\partial a}$, so it suffices to apply AD to f. The material tangent

$$C_{n+1} = \frac{\mathrm{d}\sigma_{n+1}}{\mathrm{d}\varepsilon_{n+1}} = \frac{\partial^2\omega}{\partial\varepsilon^2}(\varepsilon_{n+1}, a_{n+1}) + \frac{\partial^2\omega}{\partial a\partial\varepsilon}(\varepsilon_{n+1}, a_{n+1})\frac{\mathrm{d}a_{n+1}}{\mathrm{d}\varepsilon_{n+1}} \tag{7}$$

requires the derivative of the evolved internal variables with respect to the predicted strain. Assuming — similar to the derivation of the scheme in [58] — that the primary system of equations was solved exactly, it holds by differentiating (6) with respect to a single strain component

$$\frac{\mathrm{d}a_{n+1}}{\mathrm{d}\varepsilon_{n+1,i}} = h \cdot \frac{\partial f}{\partial\varepsilon_{n+1,i}}(\varepsilon_{n+1}, a_{n+1}) + h \cdot \frac{\partial f}{\partial a}(\varepsilon_{n+1}, a_{n+1})\frac{\mathrm{d}a_{n+1}}{\mathrm{d}\varepsilon_{n+1,i}},$$

that is,

$$\left(I - h \cdot \frac{\partial f}{\partial a}(\varepsilon_{n+1}, a_{n+1})\right)\frac{\mathrm{d}a_{n+1}}{\mathrm{d}\varepsilon_{n+1,i}} = h \cdot \frac{\partial f}{\partial\varepsilon_{n+1,i}}(\varepsilon_{n+1}, a_{n+1}).$$

Hence, the required derivative values can be obtained in a postprocessing step by six additional linear system solves, one for each Voigt component of the strain and with the same coefficient matrix the next Newton iteration would use. $\frac{\partial f}{\partial\varepsilon_{n+1,i}}$ can be evaluated with AD analogously to $\frac{\partial f}{\partial a}$. Since

Table 2 Total time spent on both types of material law evaluations with implicit Euler strategies

Architecture	Evaluation strategy	Material law, no tangent [s]	Material law, tangent [s]
CPU	Conventional	901.53	52.92
CPU	Automatic	4726.82	327.24
CPU	Semi-automatic	1281.32	90.86
GPU	Conventional	489.29	21.92
GPU	Automatic	523.64	27.50
GPU	Semi-automatic	488.57	22.23

One row of the table displays the accumulated time spent on material law evaluations with and without tangent, respectively, during one run of the simulation

$\frac{\mathrm{d}\varepsilon_{n+1}}{\mathrm{d}\varepsilon_{n+1}} = I$, it is then straightforward to propagate the derivatives with respect to the strain with AD through an evaluation of the stress relationship (1) to obtain both σ and C_{n+1}. As before, this involves also a partial of ω and requires second order AD capabilities.

Table 2 provides an overview over time spent with the implicit Euler variants on material law evaluations with and without tangent in our running example. Here, all displayed configurations perform the exact same number of both types of material law evaluations, and the timings are immediately comparable. We should also mention that here, all simulation results obtained are identical up to machine precision. The timings reveal two important trends. First, the automatization on the CPU is costly. Given the significant runtime improvements from switching to the semi-automatic evaluation strategy, part of this cost is due to AD and the automatic computation of the partials of ω and Ψ. Another part of the cost is due to the generality. Unlike in the conventional implementation, we lack additional knowledge about the roles of internal variables. We have no formula for the flow direction and solve a full system of nonlinear equations with Newton's method instead. All evaluation strategies are notably accelerated by the GPU, and here, most important, *even keeping the full automatization does not incur major performance costs*. This is due to overlap of CPU and GPU workloads as detailed in Sect. 6.

4.2 Rosenbrock and Runge–Kutta schemes with adaptive step size

With a single implicit Euler step, there is no direct form of error control for the involved material law evaluations. The surrounding elasticity solver cannot compensate this lack of accuracy and will therefore solve the time discretized elasticity problem with potentially wrong stress (and stiffness) input. This regards nonlinear effects in particular. Since we cannot know in advance if and when these take place, we have to discretize the whole loading path with small loading steps. As we detail in the following, while this improves the accuracy of stresses, the accuracy of tangents can not necessarily be guaranteed this way.

To that end, we first establish an interpretation of the algorithmic derivative of a single implicit Euler step as introduced in the previous section as a single implicit Euler step applied to an ODE for the derivative. Let a parameter dependent ODE system

$$\dot{y} = f(y,\ p) \tag{8}$$

be given. We differentiate both sides of (8) with respect to p and formally interchange the order of derivatives on the left hand side to obtain

$$\frac{\mathrm{d}}{\mathrm{d}t}\left(\frac{\mathrm{d}y}{\mathrm{d}p}\right) = \frac{\partial f}{\partial y}(y,\ p)\frac{\mathrm{d}y}{\mathrm{d}p} + \frac{\partial f}{\partial p}(y,\ p). \tag{9}$$

Assuming sufficient smoothness [44], the derivative of y with respect to p is the unique solution to (9) together with an initial value. The implicit Euler scheme with step size h applied to the coupled system formed by (8) and (9) yields

$$y_{n+1} = y_n + hf(y_{n+1},\ p), \tag{10}$$

$$\frac{\mathrm{d}y_{n+1}}{\mathrm{d}p} = \frac{\mathrm{d}y_n}{\mathrm{d}p} + h\frac{\partial f}{\partial y}(y_{n+1},\ p)\frac{\mathrm{d}y_{n+1}}{\mathrm{d}p} + h\frac{\partial f}{\partial p}(y_{n+1},\ p). \tag{11}$$

Clearly, this can be solved in two stages. After a solve of the nonlinear equation (10) for y_{n+1}, one linear solve of (11) is sufficient to recover the derivative $\frac{\mathrm{d}y_{n+1}}{\mathrm{d}p}$. However, (11) can equivalently be obtained in an algorithmic manner by differentiating (10) with respect to p as long as $\frac{\mathrm{d}h}{\mathrm{d}p} = 0$. Hence, the algorithmic derivative of a single implicit Euler step has an interpretation as a single implicit Euler step applied to the ODE for the derivative.

This holds likewise for the single implicit Euler step applied in the schemes in Sect. 4.1 where we have already seen the two-step solution procedure. Now we deduce properties of the numerical tangent approximation via the ODE it approximates. Let $f(\varepsilon,\ a) = \frac{\partial \Psi}{\partial A}\left(-\frac{\partial \omega}{\partial a}(\varepsilon,\ a)\right)$ denote the right hand side of the evolution equation (2). In the setting of Sect. 4.1, we have to consider the numerical ODE solve in the context of a single material law evaluation with initial data a_n and $\frac{\mathrm{d}a_n}{\mathrm{d}\varepsilon_{n+1}} = 0$ with step size $h = t_{n+1} - t_n$ over the time interval $[t_n, t_{n+1}]$. Here, ε_{n+1} plays the role of the parameters. The evolution equation $\dot{a} = f(\varepsilon(t), a)$ leads to the ODE

$$\frac{\mathrm{d}}{\mathrm{d}t}\left(\frac{\mathrm{d}a}{\mathrm{d}\varepsilon_{n+1}}\right) = \frac{\partial f}{\partial a}(\varepsilon(t), a)\frac{\mathrm{d}a}{\mathrm{d}\varepsilon_{n+1}} + \frac{\partial f}{\partial \varepsilon}(\varepsilon(t), a)\frac{\mathrm{d}\varepsilon(t)}{\mathrm{d}\varepsilon_{n+1}} \qquad (12)$$

for the derivative. By the properties of the implicit Euler scheme, the numerical solve of the undifferentiated evolution equation is guaranteed to converge with order one to the exact solution as $h \to 0$. Here, the user can influence accuracy by choosing smaller loading steps.

For the ODE for the derivative (12), the situation is different. Independent of the loading step size, ε_{n+1} always refers to the strain value at time t_{n+1}. The linear interpolation

$$\varepsilon(t) = \varepsilon_n \cdot \frac{t_{n+1} - t}{h} + \varepsilon_{n+1} \cdot \frac{t - t_n}{h}$$

between the known strain values leads to

$$\frac{\mathrm{d}\varepsilon(t)}{\mathrm{d}\varepsilon_{n+1}} = \frac{t - t_n}{h},$$

which is the linear interpolation between 0 and 1 over the integration interval $[t_n, t_{n+1}]$. Therefore, the ODE for this particular derivative changes its shape with h. As the ODE is not invariant with respect to the integration interval, we cannot expect convergence to the exact solution with $h \to 0$ if only a single implicit Euler step is applied.

The following example illustrates that the relative error in the differentiated internal variables might even increase for $h \to 0$. We compare the results obtained by single implicit Euler steps to the results obtained by implicit Euler with a simple step size control mechanism. Consider a single voxel of the elasto-viscoplastic material (4), (5) with the parameters from Table 1. We use the mixed boundary conditions from Fig. 2. This loading path is discretized by varying numbers of equidistant loading steps. For each loading step, a material law evaluation with or without substeps is performed. The relative errors observed in the derivative $\frac{\mathrm{d}\varepsilon_{vp,n+1,xx}}{\mathrm{d}\varepsilon_{n+1,xx}}$ can be seen in Fig. 4. Clearly, the relative error increases for $h \to 0$.

This shows that an accurate tangent evaluation cannot be performed without further discretization of the integration interval $[t_n, t_{n+1}]$ and serves as an additional motivation for adaptive substeps that are otherwise studied e. g. in [5] in the context of material law evaluation. Specifically, the material law inputs and outputs still follow the global time discretization, but locally, each material law evaluation uses a further discretization of $[t_n, t_{n+1}]$ to meet specified tolerances. In this section, we analyze well-known integration schemes with respect to automatic differentiation in the presence of step size control. *Note that implicit Euler with adaptive steps is not used in the remaining parts of this paper; instead, schemes with step size control via an embedded method are considered.*

For adaptive time step sizes, the computation of the material tangent still requires the derivative of the evolved internal variables with respect to the predicted strain. Even if it is in principle possible to propagate those derivatives by AD through multiple steps of an ODE integration scheme in a blackbox manner, this corresponds to an algorithmic differentiation of an approximation and comprises a risk of inaccurate derivatives. The issues of blackbox differentiation of ODE integration schemes and possible solutions are discussed in [13]. Particularly, two problems are mentioned. First, the step size is solely determined by the integration of the primal equation. Hence, there are no guarantees for the accuracy of the derivatives. Second, the differentiation of the step size control mechanism spoils the result with discretization dependent components. In [13], the focus is on an aposteriori error correction that recovers the desired derivatives from quantities obtained by blackbox differentiation. Here, we study the continuous approach to the problem in greater detail and refine the strategy of solving simultaneously an ODE for the derivative for the case of Rosenbrock methods and both explicit and implicit Runge–Kutta schemes in the presence of step size control. In Theorem 2 and Corollary 1, we show that the ansatz is equivalent to suitably modified blackbox differentiation. Particularly, we guarantee that the derivatives are as accurate as the primal solutions.

For the sake of notational simplicity, we develop the following theory for autonomous ODEs and require implicitly that the used integration schemes satisfy the consistency condition that they yield the same numerical solution before and after transformation of the ODE to autonomous form.

Assuming sufficient smoothness [44], the derivative of y with respect to p is the unique solution to (9). The combined system (8) and (9) inherits the stability properties of (8) in the sense that the Jacobian of the right hand side with respect to the unknowns is of block type

$$\frac{\partial}{\partial \begin{bmatrix} y & \frac{\mathrm{d}y}{\mathrm{d}p} \end{bmatrix}} \begin{bmatrix} f(y, p) \\ \frac{\partial f}{\partial y}(y, p)\frac{\mathrm{d}y}{\mathrm{d}p} + \frac{\partial f}{\partial p}(y, p) \end{bmatrix} = \begin{bmatrix} \frac{\partial f}{\partial y}(y, p) & 0 \\ * & \frac{\partial f}{\partial y}(y, p) \end{bmatrix} \qquad (13)$$

and has the same eigenvalues as $\frac{\partial f}{\partial y}(y, p)$. The formula for the lower left block — it does not influence the eigenvalues — is displayed in the proof of Theorem 2.

For some classes of integration schemes, the simultaneous solve of (8) and (9) can be realized by means of automati-

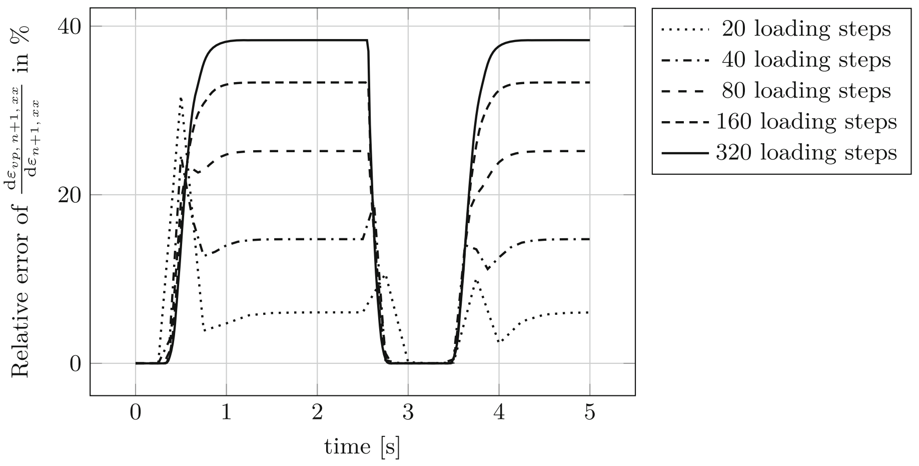


Fig. 4 Influence of loading step size on the relative error of a differentiated internal variable. Solution obtained by single implicit Euler steps compared to solution obtained by implicit Euler with adaptive substeps

cally differentiating the numerical solve of (8) with respect to p in a blackbox manner with some additional adaptions. Let an integration scheme with s stages and both linear and nonlinear implicit terms be specified by the update relations

$$k_i = hf\left(y_n^{(i)},\ p\right) + hJ\sum_{j=1}^{s}\gamma_{ij}k_j, \tag{14}$$

$$y_{n+1} = y_n + \sum_{j=1}^{s} b_j k_j \tag{15}$$

where $J = \frac{\partial f}{\partial y}(y_n,\ p)$ is the Jacobian of the right hand side and

$$y_n^{(i)} = y_n + \sum_{j=1}^{s} a_{ij}k_j.$$

Theorem 2 *Let initial data y_n and $\frac{\mathrm{d}y_n}{\mathrm{d}p}$ be given. The algorithmic derivative of a single step of the scheme (14), (15) with step size h applied to (8) yields the same value $\frac{\mathrm{d}y_{n+1}}{\mathrm{d}p}$ as an application of the same integration step to the combined system (8) and (9) as long as $\frac{\mathrm{d}h}{\mathrm{d}p} = 0$ and as long as the derivatives of equation solves are recovered according to the implicit function theorem. In terms of automatic differentiation, it is sufficient if h does not carry derivative values and equation solves are treated as elementary operations.*

Proof For notational simplicity, let p be scalar. Let y denote the solution to (8), $\frac{\mathrm{d}y}{\mathrm{d}p}$ its algorithmic derivative with respect to p and $\left[y\ \tilde{y}\right]$ the solution to the combined system (8) and (9). Likewise, we refer to the stage vectors for the solution step of (8) as k_i and to the stage vectors for the solution step of the combined system as $\left[k_i\ \tilde{k}_i\right]$. By the linearity of (15) and the initial value relation $\tilde{y}_n = \frac{\mathrm{d}y_n}{\mathrm{d}p}$, it is sufficient to ensure that $\tilde{k}_i = \frac{\mathrm{d}k_i}{\mathrm{d}p}$, $i = 1,\ \ldots,\ s$. If we apply the integration step to the combined ODEs (8) and (9), the equations for the stage vector components $\tilde{k}_i$ read

$$\tilde{k}_i = h\frac{\partial f}{\partial y}\left(y_n^{(i)},\ p\right)\tilde{y}_n^{(i)} + h\frac{\partial f}{\partial p}\left(y_n^{(i)},\ p\right) + h\tilde{J}\sum_{j=1}^{s}\gamma_{ij}k_j + hJ\sum_{j=1}^{s}\gamma_{ij}\tilde{k}_j, \tag{16}$$

where

$$\tilde{J} = \frac{\partial}{\partial y}\left(\frac{\partial f}{\partial y}(y_n,\ p)\tilde{y}_n + \frac{\partial f}{\partial p}(y_n,\ p)\right)$$

is the lower left block of (13) evaluated at y_n, $\tilde{y}_n$ and p. However, as long as $\frac{\mathrm{d}h}{\mathrm{d}p} = 0$, the same system of equations is obtained if we differentiate both sides of (14) with respect to p and identify $\tilde{k}_i = \frac{\mathrm{d}k_i}{\mathrm{d}p}$. To that end, note $\tilde{J} = \frac{\mathrm{d}J}{\mathrm{d}p}$. Hence, if we recover the algorithmic derivative of the k_i from solves of the equations obtained by implicit differentiation, we obtain the same result as by solving an ODE for the derivative. □

Like in [48], we assume that systems of equations are solved exactly. In the case of prescribed step sizes, Theorem 2 extends inductively to multiple subsequent integration steps. Compared to forward mode results in [39] or [62], a larger class of integration schemes is covered and a proof with insights for an AD implementation is given. Theorem 2 extends to the case of automatic step size control, for example via an embedded method according to [23], after small additional modifications.

1. To meet the assumption $\frac{\mathrm{d}h}{\mathrm{d}p} = 0$ of Theorem 2 in terms of AD, the step size control mechanism must remain undifferentiated.
2. To achieve the same accuracy for the solution components y and $\frac{\mathrm{d}y}{\mathrm{d}p}$, all of them must be regarded in the step size control error measure.

These additional modifications can also be found among the general suggestions in [13]. Here, we have shown that they are — together with the appropriate treatment of equation solves — sufficient to turn blackbox differentiation of an ODE integration scheme of the type (14), (15) into an algorithm that is equivalent to solving an ODE for the derivative.

Corollary 1 *Theorem 2 generalizes to subsequent integration steps also in the presence of automatic step size control as long as step size control is excluded from differentiation and derivative components are regarded in the error measure. The obtained derivative is as accurate as the primal solution.*

Theorem 2 and Corollary 1 cover various classes of well-known integration schemes. If we choose $a_{ij} = 0$ for $j \geq i$ and $\gamma_{ij} = 0$ for $j > i$, (14) and (15) turn into a Rosenbrock scheme [22]. There, only linear implicit terms are used and (16) can be simplified to s linear solves

$$(I - \gamma_{ii} h J)\tilde{k}_i = h \frac{\partial f}{\partial y}\left(y_n^{(i)},\ p\right) \tilde{y}_n^{(i)} + h \frac{\partial f}{\partial p}\left(y_n^{(i)},\ p\right) + h\tilde{J} \sum_{j=1}^{i} \gamma_{ij} k_j + hJ \sum_{j=1}^{i-1} \gamma_{ij} \tilde{k}_j. \quad (17)$$

The solve for $\tilde{k}_i$ can be performed immediately after the solve for k_i. For the choice $\gamma_{ij} = 0$ for all i and j, we obtain an implicit Runge–Kutta scheme [23]. The implicit Euler step discussed at the beginning of this section is an example for this and hence a special instance of Theorem 2. If additionally $a_{ij} = 0$ for $j \geq i$, we obtain an explicit Runge–Kutta scheme [23]. There, no equation solves are required and Theorem 2 simplifies to a straightforward application of forward AD to the stage vector updates. Otherwise, AD can be used to compute the derivatives required in the setup of (16).

In the GSM context, the components of ε_{n+1} play the role of the parameter p, a_{n+1} corresponds to y and f is the right hand side of (2). We apply Corollary 1 for the computation of $\frac{\mathrm{d}a_{n+1}}{\mathrm{d}\varepsilon_{n+1}}$. For each class of integration schemes, the AD tool must be capable of computing various higher order derivatives. For explicit Runge–Kutta schemes, as before, we need one derivative order for the computation of the material tangent and one for the evaluation of the partials. For Rosenbrock methods, however, the computation of the Jacobian of the right hand side requires an additional derivative order. This is due to the term $\tilde{J} = \frac{\mathrm{d}J}{\mathrm{d}p} = \frac{\mathrm{d}}{\mathrm{d}p}\frac{\partial}{\partial a}\left(\frac{\partial \Psi}{\partial A}(\ldots)\right)$ in (17). It is in principle possible to extend the AD tool presented Sect. 5 to third order derivatives. However, additional derivative orders incur an exponential increase in memory and/or runtime [19] and we do not expect reasonable performance. Thus, to recover one derivative order, the user has to implement the partials of ω and Ψ explicitly in this case, i. e. only the semi-automatic evaluation strategy is available.

We consider the pair of explicit Runge–Kutta schemes from [7] that is known from MATLAB's[6] routine ode23 and a lower-order Runge–Kutta pair formed by the explicit Euler scheme and Heun's method. This pair is also used for DAE integration in the context of material law evaluation in [25] and we refer to it as ode12. Finally, we include the Rosenbrock scheme from [56] that is behind MATLAB's ode23s. We implement all three with automatic step size control according to [23] and keep the MATLAB default tolerances $a_{\text{tol}} = 10^{-6}$ and $r_{\text{tol}} = 10^{-3}$. If we solve additionally for the derivatives, the solutions for the derivative of a with respect to ε_{n+1} enter the error measure in the same way as primal solution components.

Table 3 displays the timings for Runge–Kutta and Rosenbrock evaluation strategies. Compared to the previous timings in Table 2 without adaptive step size control, we take notice that on the CPU, semi-automatic evaluations without tangent with ode12 and especially ode23 can be performed even faster than the conventional evaluation strategy. Often, one or a few adaptive steps are sufficient, and Runge–Kutta steps are computationally cheaper than those of implicit schemes since no equation solves are involved. Note that semi-automatic evaluation without tangent does not require AD. Material law evaluations with adaptive step size and tangent are quite expensive. This is attributed to the effort of solving an ODE coupled with one for the derivative components. Again, the GPU improves the performance significantly, especially for evaluations with tangents. While there are no significant performance differences without tangent, ode23 is fastest with tangent, and is also competitive to the implicit Euler scheme on the CPU. Also, semi-automatic evaluation improves performance on the CPU, and automatic evaluation has often no performance drawbacks on the GPU. The bad tangent performance of ode23s is related to register usage; this is explained in Sect. 6.

As can be seen in Table 4 for the case of 80 loading steps, adaptive substeps tend to reduce the overall number of elasticity solver iterations so that there are less material law evaluations without tangent in total. Figure 5, however, reveals that the loading step size remains — consistently across all ODE solvers — the key influence factor on the number of iterations per loading step.

In Fig. 6, the average number of substeps per loading steps are visualized for the four different ODE solvers. By design, implicit Euler always uses one substep per loading step. For the other three solvers, the average number of substeps varies. It is strongly increasing when nonlinear effects occur in the composite. As expected, the first/second order solver ode12 needs the most substeps to reach the prescribed accuracy. The second/third order solvers ode23 and ode23s need a comparable number of substeps. Consequently, the semi-implicit and

[6] https://de.mathworks.com/products/matlab.html.

Table 3 Total time spent on both types of material law evaluations with Runge–Kutta and Rosenbrock schemes

Architecture	ODE solver	Evaluation strategy	Material law [s] no tangent	Material law [s] tangent
CPU	ode12	Automatic	1398.49	7241.91
CPU	ode12	Semi-automatic	506.66	1656.17
CPU	ode23	Automatic	891.82	1340.88
CPU	ode23	Semi-automatic	339.65	353.18
CPU	ode23s	Semi-automatic	1185.79	2704.99
GPU	ode12	Automatic	463.76	117.74
GPU	ode12	Semi-automatic	463.98	97.71
GPU	ode23	Automatic	445.56	46.40
GPU	ode23	Semi-automatic	454.85	42.17
GPU	ode23s	Semi-automatic	482.63	494.70

One row of the table displays the accumulated time spent on material law evaluations with and without tangent, respectively, during one run of the simulation. Compare also Table 2

Table 4 Impact of ODE solver choice on number of elasticity solver iterations

ODE solver	Number of iterations
Impl. Euler	1733
ode12	1619
ode23	1575
ode23s	1602

The distinction CPU/GPU and the evaluation strategy types have no influence in this regard

computationally more expensive ode23s cannot outperform the explicit ode23.

4.3 Solution accuracy

FFT-based homogenization of Moulinec-Suquet [41] applied to materials with nonlinear behaviour is subject to a spatial discretization error of the partial differential equation $\operatorname{div}\sigma = 0$ investigated in detail by Schneider [51] and furthermore two types of time discretization errors. First, the interaction between different regions of the material (quadrature points) over time is neglected on the material law evaluation level. Second, each integration of the ordinary differential equations (2), that is, each material law evaluation, introduces a local error in the internal variables.

For our example presented in Sect. 3, we study the influence of the adaptive time step size control on the overall error by comparing the ODE solvers presented above.

The stress response in loading direction is shown in Fig. 7. As expected, due to the error control, all ODE solvers with adaptive time steps predict the same effective stress response within the given tolerances. Moreover, as can be seen in Fig. 8, the error for coarse loading steps is reduced to approximately 30% of the error of the implicit Euler solver. Thus, the error of the material law evolution, that is, the accuracy of the ODE solver, is dominating the overall error of the FFT-based based homogenization for this example.

For the tangential stiffness shown in Figs. 9 and 10, the results depend on the time discretization as explained in detail in Sect. 4.2. Therefore, we cannot perform a convergence analysis with respect to the loading step size. We observe that all ODE solvers with adaptive time step size control predict almost the same tangent due to the error control. The differences observed between single implicit Euler steps and schemes with adaptive substeps are in accordance with the example on the relative error amplification in Sect. 4. Note that the tangent formula (7) reads for the potentials (4) and (5)

$$C_{n+1} = C^{\mathrm{e}} \left(I - \frac{\mathrm{d}\varepsilon_{vp\,n+1}}{\mathrm{d}\varepsilon_{n+1}} \right),$$

that is, linear combinations of errors as depicted in Fig. 4 are substracted from the components of the elastic stiffness matrix. This effect regards voxels that follow the Michel Suquet law and can still be seen in the effective stiffness.

4.4 Stress-driven error control

Internal variables do not always have a physical meaning, and the material law outputs that are of immediate relevance to the elasticity solver are σ_{n+1} and C_{n+1}. Its convergence test, for example, amounts to an equilibrium check of the stress field [41], and the material tangents are used to determine a linear elastic reference material [14,27,38]. In the material law evaluations, however, the tolerances specified for the ODE solver relate to an error in the internal variables. We control the error in a_{n+1} and — if we apply Corollary 1 — as well the error in $\frac{\mathrm{d}a_{n+1}}{\mathrm{d}\varepsilon_{n+1}}$.

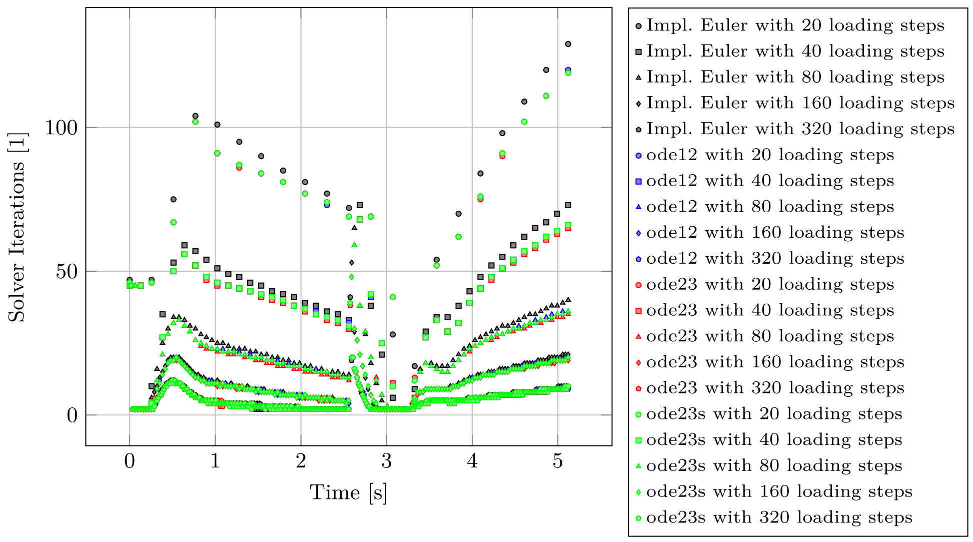


Fig. 5 Iterations of FFT-based homogenization per loading step for the example of Sect. 3. (Color figure online)

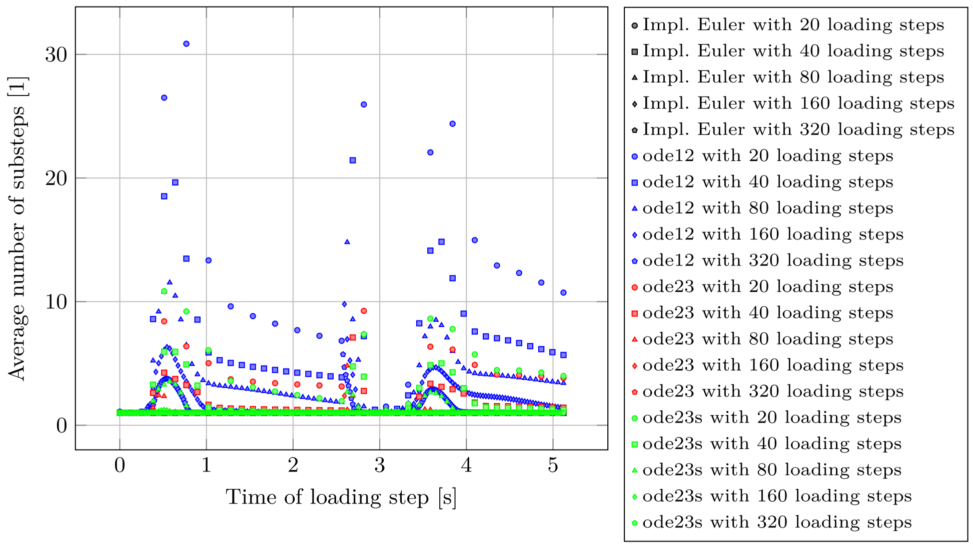


Fig. 6 Spatially averaged number of ODE solver substeps per loading step for the example of Sect. 3. For each loading step, the number of substeps is plotted against the loading step's time. (Color figure online)

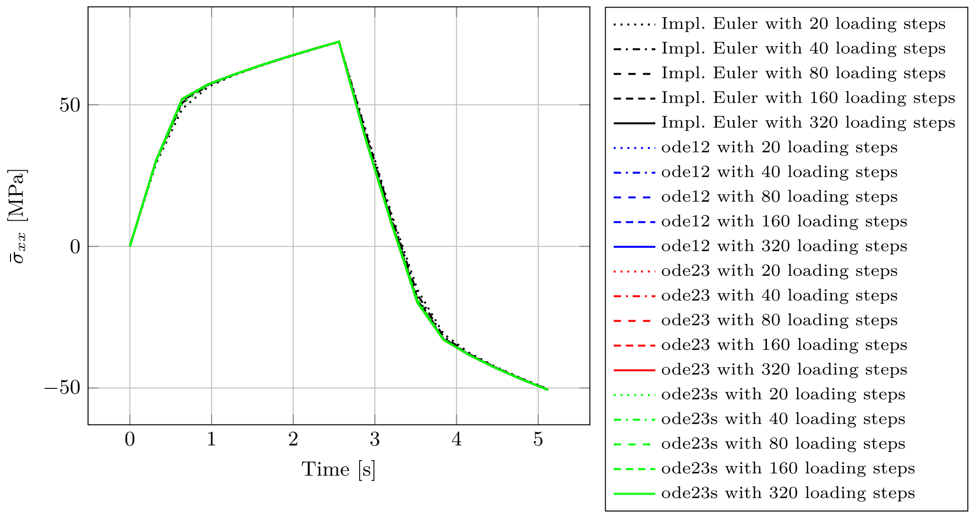


Fig. 7 $\bar{\sigma}_{xx}$ for the example of Sect. 3. (Color figure online)

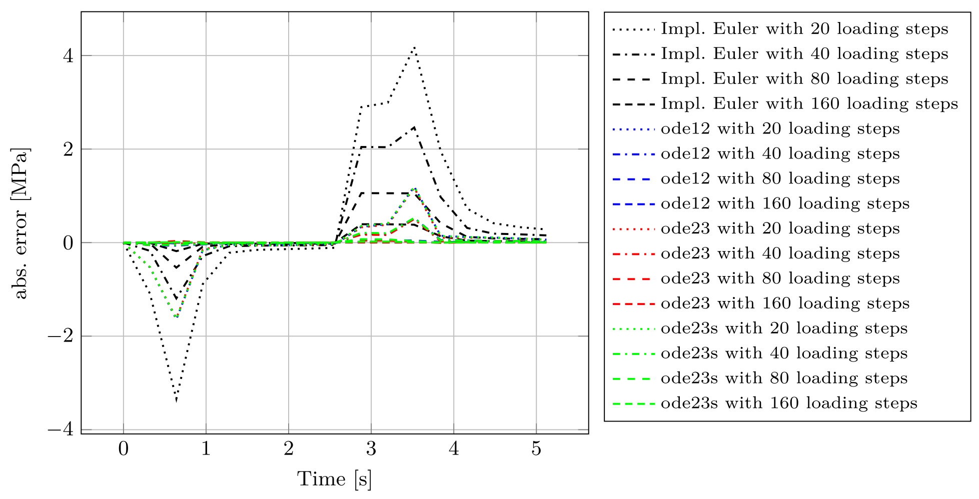


Fig. 8 Difference of $\bar{\sigma}_{xx}$ to finest time discretization (320 loading steps) for the different ODE solvers. (Color figure online)

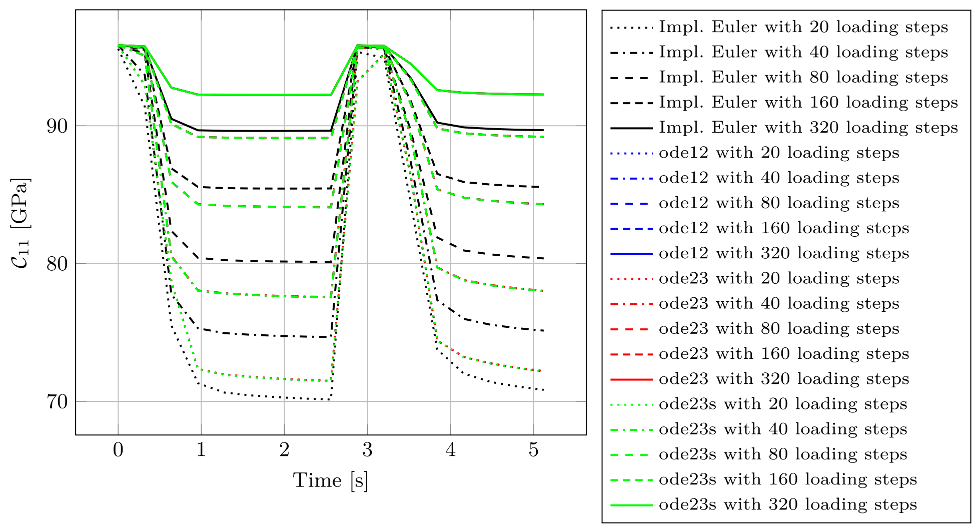


Fig. 9 $\mathcal{C}_{11}$ for the example of Sect. 3. (Color figure online)

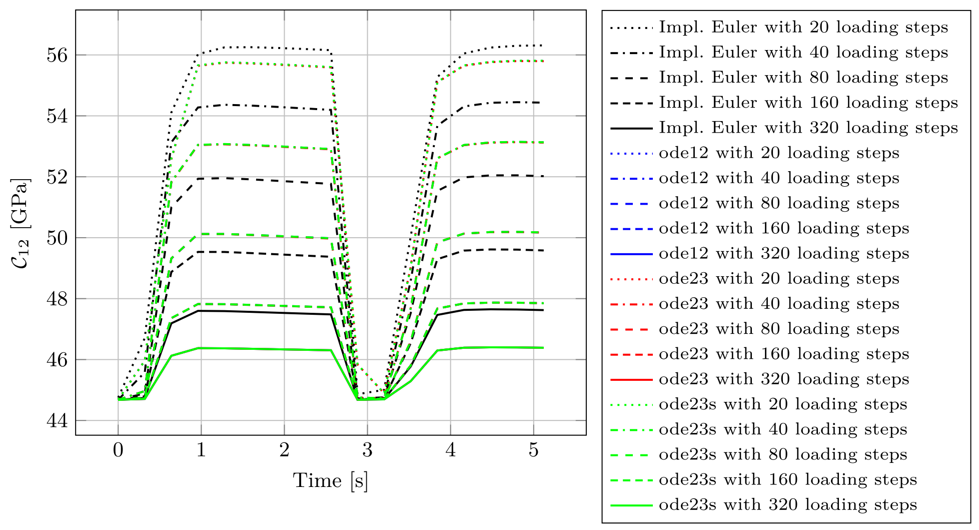


Fig. 10 $\mathcal{C}_{12}$ for the example of Sect. 3. (Color figure online)

In the GSM given by Equations (4) and (5), for example, the stress relationship (1) turns into $\sigma = C^{\mathrm{e}}(\varepsilon - \varepsilon_{vp})$, that is, any error in ε_{vp} enters σ multiplied by the elastic stiffness tensor. Depending on the specific instance of C^{e}, it might be necessary to adapt the tolerances of the ODE solver to end up with stress values that are sufficiently accurate for the PDE solver. This is avoided by an error control on the ODE level that is directly tied to the accuracy of the stresses.

The step size control mechanism from [23] captures the deviation between two ODE solutions of different order of convergence in an error measure. Depending on the error, steps are accepted or rejected and the step size is adapted accordingly. Instead of using the internal variable approximations directly in the error measure, we transform them together with the adequate linear interpolation between ε_n and ε_{n+1} for the substep of interest via the relationship (1) into a pair of stresses. If σ depends — as above — linearly or, more generally, Lipschitz on the internal variables, this yields a pair of stresses with the analogous order relations. The rationale of the step size control carries over, and we evaluate the error measure on the stresses instead. If we solve additionally for the derivative $\frac{\mathrm{d}a_{n+1}}{\mathrm{d}\varepsilon_{n+1}}$, the same evaluation of (1) (performed on forward AD types instead) transforms additionally the approximations of the internal variable derivatives into a corresponding pair of material tangents that may then enter the error measure in the same way the derivative components did before. This way, we control the error in σ_{n+1} and C_{n+1}.

As can be seen in Fig. 11, stress-driven error control also reduces the impact of the ODE solver choice on the effective stress response for all numbers of loading steps.

Similar ideas can be employed for the convergence criterion of Newton's method in the schemes from Sect. 4.1. Instead of iterating until convergence in a, we may compute the stress resulting from the current iterate via (1) in each Newton iteration and converge σ instead.

5 Automatic differentiation on GPUs

To summarize the basic ideas of automatic differentiation, we view a floating point computation with fully evaluated control flow as a function $x \mapsto y$ that is composed of elementary mathematical operations like $+$, $\cdot$ or standard math library functions like sin. If we differentiate the composed operations according to the chain rule, we obtain the *algorithmic derivative* of the computer program. *Automatic differentiation* deals with techniques that obtain algorithmic derivatives in an automatic fashion. A comprehensive introduction is given in [19].

As both ω and Ψ are scalar valued functions and have — with respect to both ε and a — more inputs than outputs, it seems appropriate to use the *reverse mode of automatic differentiation* to evaluate the partial derivatives on the right hand sides of the GSM constitutive equations (1) and (2). C_{n+1}, on the other hand, arises as the derivative of σ_{n+1} with respect to ε_{n+1}, that is, six Voigt components with respect to six Voigt components. We compute it with the *forward mode of automatic differentiation*, possibly the *forward vector mode*. To compute both the partials and C_{n+1} with AD at the same time, we combine the forward and reverse mode in an adjoints of tangents fashion [19]. While the computation is generally executed on a forward AD data type, all local evaluations of partials are obtained by additional applications of the reverse mode. In the context of semi-automatic ode23s, we use the second order forward (vector) mode for the Jacobians and tangents.

The implementation of the first and second order forward (vector) mode follows the same principles as CPU implementations like [47]. The reverse mode of AD, however, is subject to a global information problem that is typically solved by *taping*. The sequence of operations is first executed in forward direction and remembered together with all intermediate results. Then, the corresponding sequence of

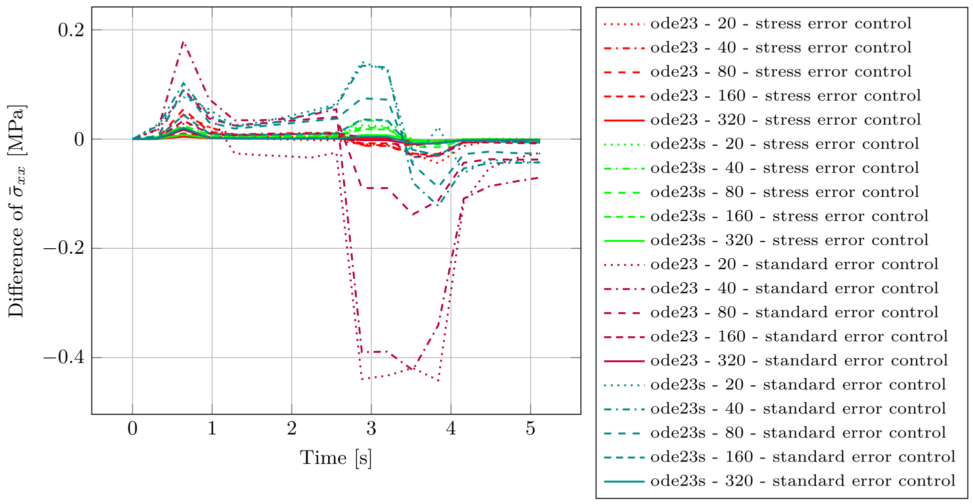
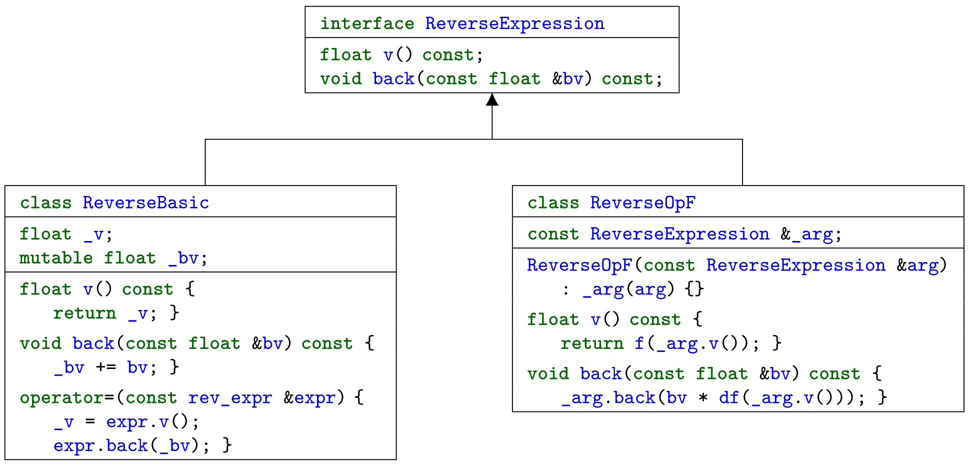


Fig. 11 Difference of $\bar{\sigma}_{xx}$ for ode23 and ode23s compared to ode12 at the same loading path discretization and with the same error measure for step size control. (Color figure online)



```
interface ReverseExpression
float v() const;
void back(const float &bv) const;
```

```
class ReverseBasic
float _v;
mutable float _bv;
float v() const {
   return _v; }
void back(const float &bv) const {
   _bv += bv; }
operator=(const rev_expr &expr) {
   _v = expr.v();
   expr.back(_bv); }
```

```
class ReverseOpF
const ReverseExpression &_arg;
ReverseOpF(const ReverseExpression &arg)
   : _arg(arg) {}
float v() const {
   return f(_arg.v()); }
void back(const float &bv) const {
   _arg.back(bv * df(_arg.v())); }
```

Fig. 12 Schematic implementation of the reverse mode of AD on the expression level. `_arg.back(bv * df(_arg.v()))` is the classical backpropagation formula [19]

derivatives is evaluated according to the chain rule in reverse order. On the GPU, this memory-intensive approach is prohibitive. Since the reverse mode of AD is only needed in a very local manner, we may replace taping by recomputations: If an intermediate value is required during reverse evaluation, the sequence of operations is partially re-evaluated in forward direction up to the required point. Similar approaches are pursued in [32].

This can be realized by an *operator overloading* ansatz at low computational overhead on the expression level. We employ *expression template techniques* that have previously been shown to perform well for the treatment of right hand sides in the forward mode of AD [43] and in Jacobi taping [26] as well as primal value taping [46] in the reverse mode of AD. Here, we use expression templates to convert a composite operation into a structured data type that represents the computational graph and allows for its traversal in forward and reverse direction. This way, the structure of the computation is fully exposed to the compiler and can be optimized during compilation. The *curiously recurring template pattern* is used to shift overhead due to the interface in the inheritance tree in Fig. 12 from runtime to compile time.

Figure 12 showcases the reverse mode without additional tangents using the example of a unary elementary operation `f()`. The interface `ReverseExpression` defines a routine `v()` for forward evaluation and a routine `back()` for backpropagation of derivatives. On the one hand, it is implemented as a type `ReverseBasic` that contains actual data, that is, a primal value `_v` and an adjoint value `_bv`. On the other hand, there are derived types that stand for applied elementary operations such as `ReverseOpF`. They are created by operation overloads such as

```
ReverseOpF f(const ReverseExpression &expr)
{ return ReverseOpF(expr); }
```

that do not immediately apply `f()` but store a reference to the arguments in the returned object. Types such as `ReverseOpF` implement the interface in a way that allows for the forward and reverse evaluation of the computational graph. A call to `v()` causes the forward evaluation of `_arg` and subsequent application of `f()`. A call to `back()` propagates derivative values in reverse direction where `df()` stands for the derivative of `f()` and must be implemented explicitly. The call `_arg.v()` in `ReverseOpF::back()` causes forward re-evaluations. This extends analogously to n-ary operations and additional forward and reverse evaluation of tangents for second order derivatives. Consider a code segment

```
// initialize primal components
// set derivative values to 0
ReverseBasic arg1 = ..., arg2 = ..., ...;

ReverseBasic result;
result._bv = 1.0; // seeding
result = CompositeExpression(
              arg1, arg2, ...);
```

where `CompositeExpression` stands for a composition of multiple elementary operations. Each elementary operation must be implemented according to Fig. 12. The operation overloads are used to build up the computational graph of this right hand side and in the course of the assignment to `result`, `ReverseBasic::operator=()` is used to trigger its forward and subsequent reverse evaluation. In the end, `argn._bv` carries the machine accurate derivative of `result._v` with respect to `argn._v` where `n = 1, 2, ...`.

The proposed AD tool can be implemented in C++ using C++11 features that are supported both by standard compilers such as g++ and by Nvidia's CUDA compiler driver `nvcc`. Particularly, the AD tool can be applied both inside OpenMP threads and CUDA kernels.

We improve the performance of the AD tool by some adaptions that are specific to our problem and setting.

1. During expression tree forward traversal, it is possible to evaluate primal values only once and store them in the nodes of the tree [43]. However, to consume as little memory as possible, we use recomputations instead. This is especially important for the GPU on which memory operations are costly and the number of registers used per thread can limit parallel execution.
2. Instead of a recursive ansatz for higher order derivatives, we implement second order expressions explicitly. This helps the compiler with the identification and elimination of common subexpressions, which it cannot always do automatically.
3. In the computation of the partials of ω, we are always only interested in the derivative with respect to either ε or a but never both. If we compute the derivative with respect to one, there is no need to propagate derivative values back to the other. Therefore, we provide mixed order expressions that actively avoid reverse propagation of derivative values to lower order type arguments.

The AD tool can only differentiate single expressions in reverse order and is overall limited to first and second order derivatives. The first and second order forward (vector) mode, however, are not restricted to single expressions and can be applied to general codes, like the ODE solvers in the case of AutoMat. In the presented design, automatic differentiation takes exclusively place in GPU registers (sometimes spilled but mostly actual, see Table 6).

To indicate the implementational effort of a potential-based material law with the AutoMat framework and this AD tool, Figs. 13 and 14 include example implementations of the Michel-Suquet law (4) and (5) for the automatic evaluation procedure. As needed for the solution process, AutoMat instantiates these templates with different combinations of primal and AD types.

6 Computational layout, profiling and performance limiters

The fields for the internal variables, the current strain field and the predicted strain field, that is, the material law inputs for all voxels, reside in host memory. In general, GPU memory is not large enough to hold all of them at the same time and the elasticity solver still runs on the CPU. Furthermore, data might reside in host memory in an array-of-struct layout that does not suite GPU computing and due to the heterogenity of the material, data for all voxels of a specific material law might be arranged in memory in a non-contiguous manner. Therefore, we divide the workload into multiple chunks of fixed size, in a way that GPU memory can at least hold the material law inputs and outputs of one or few chunks. In host memory, we allocate at least one staging area of chunk size and page-locked type that allows for fast CPU-GPU data exchange. On the host side, we copy the material law inputs of a chunk into the staging area in an OpenMP parallel manner. In doing so, we arrange them in a contiguous manner in a struct-of-arrays layout, and might convert from double to single precision. Then, we process the staging area with multiple CUDA streams. Each stream copies part of the inputs to the GPU and issues the corresponding material law evaluations. We use one CUDA thread per material law evaluation and a small multiple of 32 as block size for the computational grid. Once the evaluations are done, the stream copies the material law outputs back to the staging area. The purpose of multiple streams is an overlap of CPU-GPU data exchange with GPU computations. Once the entire staging area is processed, the material law outputs are collected from the staging area, transformed back to the original layout and otherwise postprocessed as required by the elasticity solver in an OpenMP parallel manner. By means of multiple staging areas, an overlap of CPU and GPU workloads can be achieved: During GPU computations, transformations of inputs and outputs involving other staging areas can already take place on the host side. We observed no benefits for more than two staging areas.

The CPU-GPU overlap becomes evident in Table 5. For material law evaluations without tangent, the time spent on material law evaluation is determined by the time it takes to stage and collect the data. CPU-GPU data exchange and GPU computations overlap almost completely with the CPU workloads. The minimum time needed for exchange of the combined data over the PCI Express bus (assuming full bandwidth and perfect overlap of both transfer directions) gives an impression of the amount of time that is at least hidden behind CPU workloads. The exemplary profilings presented in Fig. 15 show that the GPU compute time is in turn dominated by CPU-GPU data exchange, and due to overlap mostly hidden behind it.

For material law evaluations with tangent, the observations are different. Here, staging and collecting cannot hide all GPU workloads, in particular the GPU computations which are also more expensive than the CPU-GPU data exchange. This has two reasons. First, the postprocessing step for the tangent or solving the coupled ODE system, respectively, is in itself computationally more expensive. The derivative components, however, also increase the memory footprint of the GPU kernels, in particular the number of registers used per thread. This can be seen in Table 6. This limits the overall number of threads that can run in parallel, and it is important to keep that number small. To that end, all ODE solvers with adaptive step size among the GPU configurations with tangent are subject to another performance optimization. Instead of propagating all six tangent directions simultaneously through one material law evaluation with the forward vector mode, we re-evaluate each material law six

Fig. 13 Example implementation of ω from (4), double precision. For reverse mode differentiation of ω with our AD tool, the translation from inputs to outputs must take place in a single expression. To minimize the memory footprint, the implementation eliminates the redundant internal variable $\varepsilon_{vp\,3} = -\varepsilon_{vp\,1} - \varepsilon_{vp\,2}$, see Sect. 6 for details. The parameters are named as in Sect. 3 for readability and are assumed to be global variables; a generic implementation, however, will use a parameter array instead

```
template<typename eps_type, typename a_type, typename w_type>
void omega(const eps_type *eps, const a_type *a, w_type *w)
{
  const double mu = E * nu / (1.0 + nu) / (1.0 - 2.0 * nu);
  const double lam = 0.5 * E / (1.0 + nu);

  *w = 0.5 * (
         (eps[0] - a[0]) * (
           (2.0 * mu + lam) * (eps[0] - a[0]) +
                       lam  * (eps[1] - a[1]) +
                       lam  * (eps[2] + (a[0]+a[1]))) +
         (eps[1] - a[1]) * (
                       lam  * (eps[0] - a[0]) +
           (2.0 * mu + lam) * (eps[1] - a[1]) +
                       lam  * (eps[2] + (a[0]+a[1]))) +
         (eps[2] - (-a[0]-a[1])) * (
                       lam  * (eps[0] - a[0]) +
                       lam  * (eps[1] - a[1]) +
           (2.0 * mu + lam) * (eps[2] + (a[0]+a[1]))) +
         (eps[3] - a[2]) * mu * (eps[3] - a[2]) +
         (eps[4] - a[3]) * mu * (eps[4] - a[3]) +
         (eps[5] - a[4]) * mu * (eps[5] - a[4]))
       + H / 3.0 * (a[0] * a[0] + a[1] * a[1] + (a[0]+a[1]) * (a[0]+a[1]))
       + H / 6.0 * (a[2] * a[2] + a[3] * a[3] + a[4] * a[4])
       + sig_Y * a[5];
}
```

Fig. 14 Example implementation of Ψ from (5), double precision. The test for $\|\mathrm{dev}\, A_\nu\|_{\mathrm{eq}} + A_\alpha \leq 0$ requires a precomputation of the primal expression as indicated by the `v()` calls. In the **`if`**-case, this expression is then repeated to enable differentiation. Analogously to a, one component of A is eliminated

```
template<typename A_type>
void Psi(const A_type *A, A_type *psi)
{
  const double dev_A_nu_eq_plus_A_alpha = std::sqrt(
      ((A[0] - 0.5*A[1]) * (A[0] - 0.5*A[1]) +
       (-0.5*A[0] + A[1]) * (-0.5*A[0] + A[1]) +
       (-0.5*A[0] - 0.5*A[1]) * (-0.5*A[0] - 0.5*A[1])
      + 3.0 * (A[2] * A[2] + A[3] * A[3] + A[4] * A[4])).v() + A[5].v();

  if (dev_A_nu_eq_plus_A_alpha > 0.0)
    *psi = sig_d * dot_eps_0 / (n + 1.0) * std::pow(
      (std::sqrt(((A[0] - 0.5*A[1]) * (A[0] - 0.5*A[1]) +
                  (-0.5*A[0] + A[1]) * (-0.5*A[0] + A[1]) +
                  (-0.5*A[0] - 0.5*A[1]) * (-0.5*A[0] - 0.5*A[1]))
                + 3.0 * (A[2] * A[2] + A[3] * A[3] + A[4] * A[4]))
       + A[5]) / sig_d, n + 1.0);
  else
    *psi = 0.0;
}
```

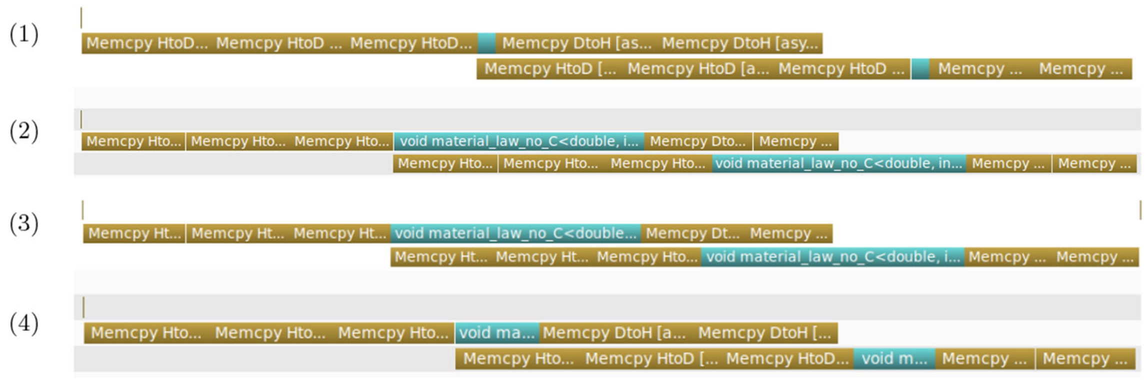


Fig. 15 Profilings performed for material law evaluations without tangent and implicit Euler — conventional (1), semi-automatic (2), automatic (3) — and automatic ode23 (4). Indicates overlap and relative duration within configurations, time scales vary between (1)–(4). Staging area processed with two CUDA streams. CPU-GPU data exchange brown, computations blue (Generated with Nvidia Visual Profiler, https://developer.nvidia.com/nvidia-visual-profiler.) (Color figure online)

Table 5 Refinement of timings for GPU configurations from Tables 2 and 3 into CPU workloads and non-overlapped GPU workloads

ODE solver	Eval. strategy	Tangent	Staging [s]	Wait for GPU [s]	Collecting [s]	PCIe bound [s]
impl. Euler	Conventional	No	236.99	0.60	215.87	44.05
impl. Euler	Semi-automatic	No	236.24	0.65	212.24	44.05
impl. Euler	Automatic	No	237.14	0.64	213.25	44.05
ode12	Automatic	No	225.14	0.63	198.87	41.16
ode12	Semi-automatic	No	225.15	0.63	197.37	41.16
ode23	Automatic	No	220.07	0.63	193.51	40.04
ode23	Semi-automatic	No	219.48	0.62	191.49	40.04
ode23s	Semi-automatic	No	221.21	0.65	197.31	40.72
impl. Euler	Conventional	Yes	10.51	0.03	11.37	2.37
impl. Euler	Semi-automatic	Yes	10.66	5.03	11.79	2.37
impl. Euler	Automatic	Yes	10.53	0.07	11.62	2.37
ode12	Automatic	Yes	10.65	90.27	16.81	2.37
ode12	Semi-automatic	Yes	10.52	70.56	16.62	2.37
ode23	Automatic	Yes	10.52	19.14	16.72	2.37
ode23	Semi-automatic	Yes	10.49	15.05	16.62	2.37
ode23s	Semi-automatic	Yes	10.50	466.05	18.14	2.37

Each row displays accumulated timings spent on material law evaluations either with or without tangent during one run of the simulation. Includes also lower time bound for the PCIe data exchange. Some variations between material law evaluations without tangent are due to differences in the number of elasticity solver iterations. The variations in the PCIe bounds indicate this extent

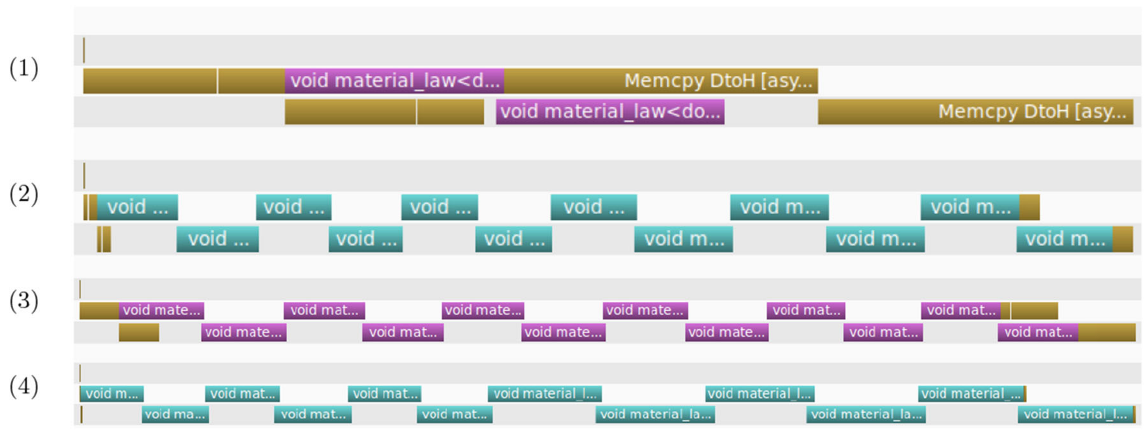


Fig. 16 Profilings performed for automatic material law evaluations with tangent — implicit Euler (1), ode12 (2), ode23 (3) — and semi-automatic ode23s (4). Time scales vary between (1)–(4). Note the six evaluation steps between data transfer in (2)–(4). Staging area processed with two CUDA streams. CPU-GPU data exchange brown, computations blue/purple (Generated with Nvidia Visual Profiler, https://developer.nvidia.com/nvidia-visual-profiler). (Color figure online)

times, each with the standard forward mode and one tangent direction, i. e. we compute C_{n+1} column by column. This can also be seen in Fig. 16. Note that this has no influence on staging, collecting or the amount of CPU-GPU data exchange. We trade memory for computations on the GPU, and the performance benefits of kernels with smaller memory footprint outweigh the additional effort incurred by the re-evaluations.

Interestingly, the CPU-GPU data exchange is—due to overlap and the cost of staging and collecting—in none of the configurations discussed above a key limiting factor. Nonetheless, our implementation of the material law from Sect. 3 reduces that data. Material law evaluations with tangent, for example, copy back C_{n+1} but neither stresses nor internal variables. Specifically for the GSM given by (4), (5), we exploit $\varepsilon_{vp} \in \operatorname{range}(\operatorname{dev})$, i. e. one component of the viscoplastic strain can be eliminated and is computed on the fly in the implementations of ω and Ψ from the others instead.

The effect of using AutoMat on the total runtime of FFT-based homogenization is summarized in Table 7. On the CPU, ode23 is the best choice. It needs approximately the same

Table 6 `ptxas` info for configurations from Table 5 (double precision)

ODE solver	Eval. strategy	Tangent	Stack frame [bytes]	Spill stores [bytes]	Spill loads [bytes]	Registers
impl. Euler	Conventional	No	0	0	0	73
impl. Euler	Semi-automatic	No	656	0	0	184
impl. Euler	Automatic	No	656	0	0	186
ode12	Automatic	No	0	0	0	156
ode12	Semi-automatic	No	0	0	0	136
ode23	Automatic	No	0	0	0	206
ode23	Semi-automatic	No	0	0	0	174
ode23s	Semi-automatic	No	944	224	392	255
impl. Euler	Conventional	Yes	0	0	0	114
impl. Euler	Semi-automatic	Yes	960	0	0	173
impl. Euler	Automatic	Yes	960	0	0	198
ode12	Automatic	Yes	0	0	0	246
ode12	Semi-automatic	Yes	0	0	0	215
ode23	Automatic	Yes	368	320	424	255
ode23	Semi-automatic	Yes	168	88	136	255
ode23s	Semi-automatic	Yes	4416	4196	3952	255

Indicates resources consumed per CUDA thread

time as our conventional implementation and gives more precise results according to Sect. 4.3. On the GPU, the choice of the ODE solver does not influence the total runtime significantly with the notable exception of ode23s. For all other ODE solvers, AutoMat accelerates the FFT-based homogenization method by a factor of more than two on the GPU. For ode23s, this holds only true if the reference material is not updated. In all other cases, the ODE solver can be chosen without performance considerations on the GPU.

7 Scaling studies

To investigate the speedup of our GPU implementation for directly implemented material laws as well as for automatically evaluated material laws, we conduct two types of performance studies. The first investigates the dependency of the speedup due to using a GPU on the number of OpenMP threads on the CPU. The second investigates the influence of the size of the geometry.

For the first study, we simulate the example of Sect. 3 for different numbers of OpenMP threads with and without one GPU. Since the compute time of the graphics card is independent of the number of used OpenMP threads, the speedup decreases with increasing number of OpenMP threads as soon as the GPU time cannot be hidden behind increasingly parallel CPU computations anymore.

For our example, the speedup for automatically evaluated material laws ranges between 4.52 and 2.00 for one resp. 28 threads, see Fig. 17. The computationally more demanding reference material calculation, which needs second order derivatives, has a maximal speedup of 63.33 for sequential CPU usage and only a speedup of 28.9 for 28 threads. Consequently, in the case of the automatically evaluated material law, the total runtime speedup decreases from 7.66 to 3.69. For conventionally implemented material laws, the speedup does not depend as strongly on the number of threads. While the material law evaluation itself has a similar speedup of 3.95 for one thread and 1.84 for 28 threads, the reference material calculation exhibits an almost constant speedup of 2.61 for one thread and 2.41 for 28 threads. Therefore, we observe for the total runtime only a decrease of the speedup from 3.12 for one thread to 1.75 for 28 threads.

To understand the dependency on the number of OpenMP threads t better, let

$$\mathrm{tot}^{\mathrm{CPU}}_{\mathrm{GPU}}(t) = \underbrace{\mathrm{GPU} - \mathrm{ovl}(t)}_{\mathrm{wait}(t)} + \mathrm{CPU}(t)$$

denote the splitting of the total runtime when using both a CPU and a GPU into CPU, GPU and overlap parts. The GPU runtime is independent of t. The time $\mathrm{wait}(t)$ spent on waiting for the GPU is measured in AutoMat, see Table 5. The overlap time

$$0 \leq \mathrm{ovl}(t) \leq \mathrm{stag}(t) + \mathrm{coll}(t)$$

is furthermore limited by the sum of staging the input for the GPU and collecting its results on the CPU. Under the assumption of perfect scaling of the CPU calculations if a

Table 7 Total runtime of FFT-based homogenization with and without updated reference material for different settings of AutoMat on the CPU and GPU

Architecture	ODE solver	Evaluation strategy	Time with update [s]	Time without update [s]
CPU	Implicit Euler	Conventional	1220.83	1218.48
CPU	Implicit Euler	Automatic	5702.26	5529.76
CPU	Implicit Euler	Semi-automatic	1642.23	1596.68
CPU	ode12	Automatic	8919.11	1731.27
CPU	ode12	Semi-automatic	2346.24	724.29
CPU	ode23	Automatic	2460.59	1163.06
CPU	ode23	Semi-automatic	856.44	485.08
CPU	ode23s	Semi-automatic	4154.99	1477.61
GPU	Implicit Euler	Conventional	695.78	707.79
GPU	Implicit Euler	Automatic	743.02	746.40
GPU	Implicit Euler	Semi-automatic	695.94	706.20
GPU	ode12	Automatic	765.57	666.40
GPU	ode12	Semi-automatic	740.66	692.89
GPU	ode23	Automatic	667.43	664.80
GPU	ode23	Semi-automatic	674.70	641.23
GPU	ode23s	Semi-automatic	1157.55	642.98

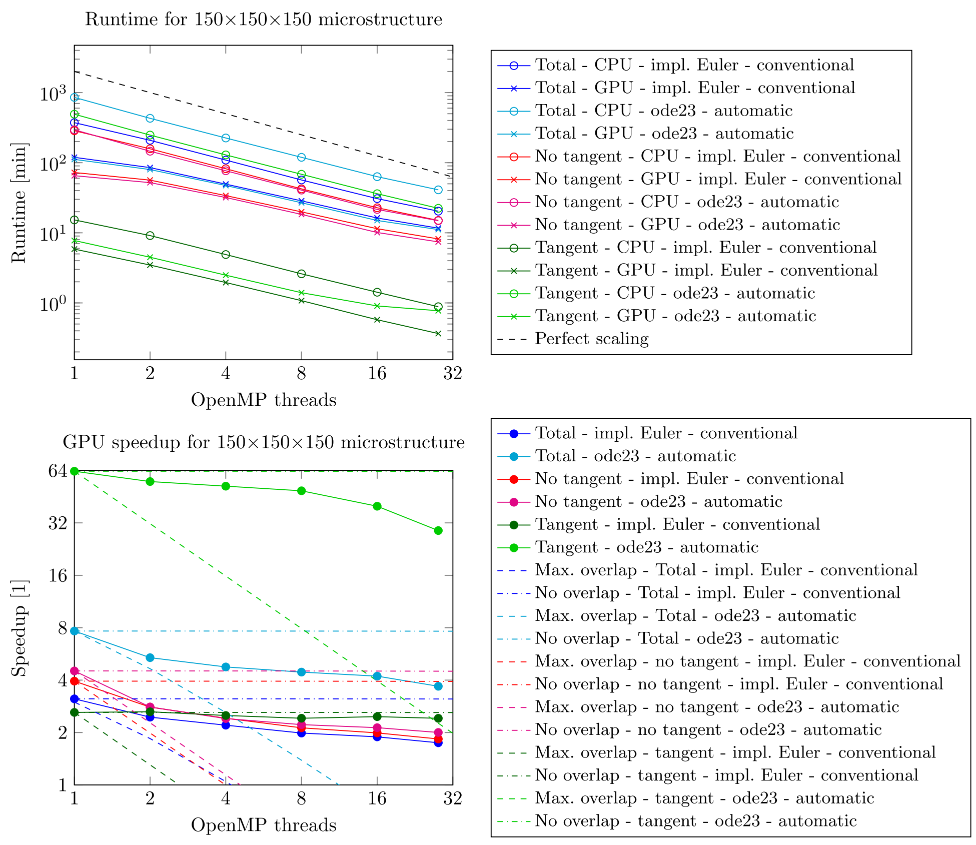


Fig. 17 Runtime and GPU speedup for varying number of OpenMP threads. (Color figure online)

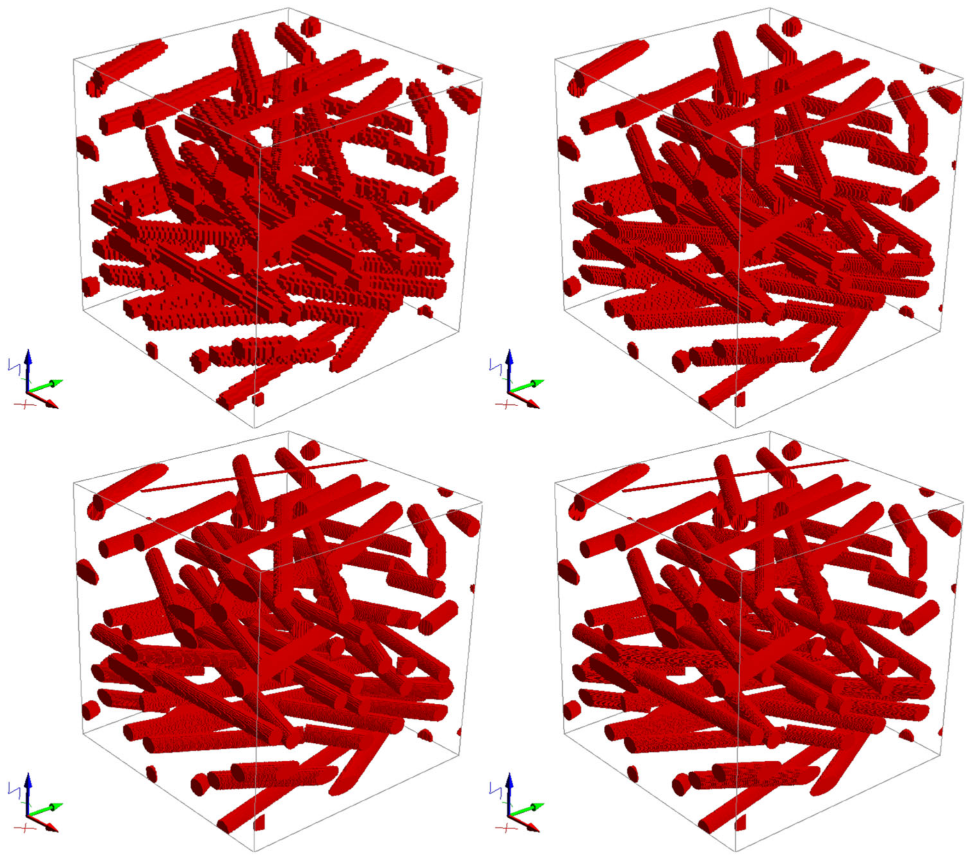


Fig. 18 Microstructure of the MMC at different resolutions

GPU is used, i. e.

$$\mathrm{CPU}(t) = \frac{\mathrm{CPU}(1)}{t},$$
$$\mathrm{stag}(t) = \frac{\mathrm{stag}(1)}{t},$$
$$\mathrm{coll}(t) = \frac{\mathrm{coll}(1)}{t},$$

as well as perfect scaling of the total time if only a CPU is used, i. e.

$$\mathrm{tot}_{\mathrm{CPU}}(t) = \frac{\mathrm{tot}_{\mathrm{CPU}}(1)}{t},$$

the speedup by using a GPU

$$\mathrm{speedup}(t) = \frac{\mathrm{tot}_{\mathrm{CPU}}(t)}{\mathrm{tot}^{\mathrm{CPU}}_{\mathrm{GPU}}(t)}$$

that is plotted in Fig. 17 can be expressed as

$$\mathrm{speedup}(t) = \frac{\mathrm{tot}_{\mathrm{CPU}}(1)}{\mathrm{GPU}\cdot t - \mathrm{ovl}(t)\cdot t + \mathrm{CPU}(1)}.$$

For either of the extremes

$$\forall t: 0 = \mathrm{ovl}(t) \implies \mathrm{GPU} = \mathrm{wait}(1),$$

i. e. no overlap, and

$$\forall t: \mathrm{ovl}(t) = \mathrm{stag}(t) + \mathrm{coll}(t)$$
$$\implies \mathrm{GPU} = \mathrm{wait}(1) + \mathrm{stag}(1) + \mathrm{coll}(1),$$

i. e. maximal overlap, the speedup formula simplifies to

$$\mathrm{speedup}(t) = \frac{\mathrm{tot}_{\mathrm{CPU}}(1)}{\mathrm{GPU}\cdot(t-1) + \mathrm{tot}^{\mathrm{CPU}}_{\mathrm{GPU}}(1)}.$$

These curves have been added for both extremes to the plots of the GPU speedup.

For the automatically evaluated material laws, the CPU runtime is increased whereas we do not observe an increased wait time for the GPU. Consequently, the speedup is higher for this case. Additionally, the decrease of the speedup is stronger because for the computationally more demanding material law, the ratio $\mathrm{GPU}/\mathrm{tot}^{\mathrm{CPU}}_{\mathrm{GPU}}(1)$ is bigger. Furthermore, an overall speedup can only be observed for material laws where $\mathrm{tot}_{\mathrm{CPU}}(1)/\mathrm{tot}^{\mathrm{CPU}}_{\mathrm{GPU}}(1) > 1$, which is the case for material laws with a high computational complexity compared to their memory footprint. To put this in perspective, we observed no speedup for linear elastic or viscoelastic material laws. For the latter, a potential-based formulation and example parameters can be found in [16].

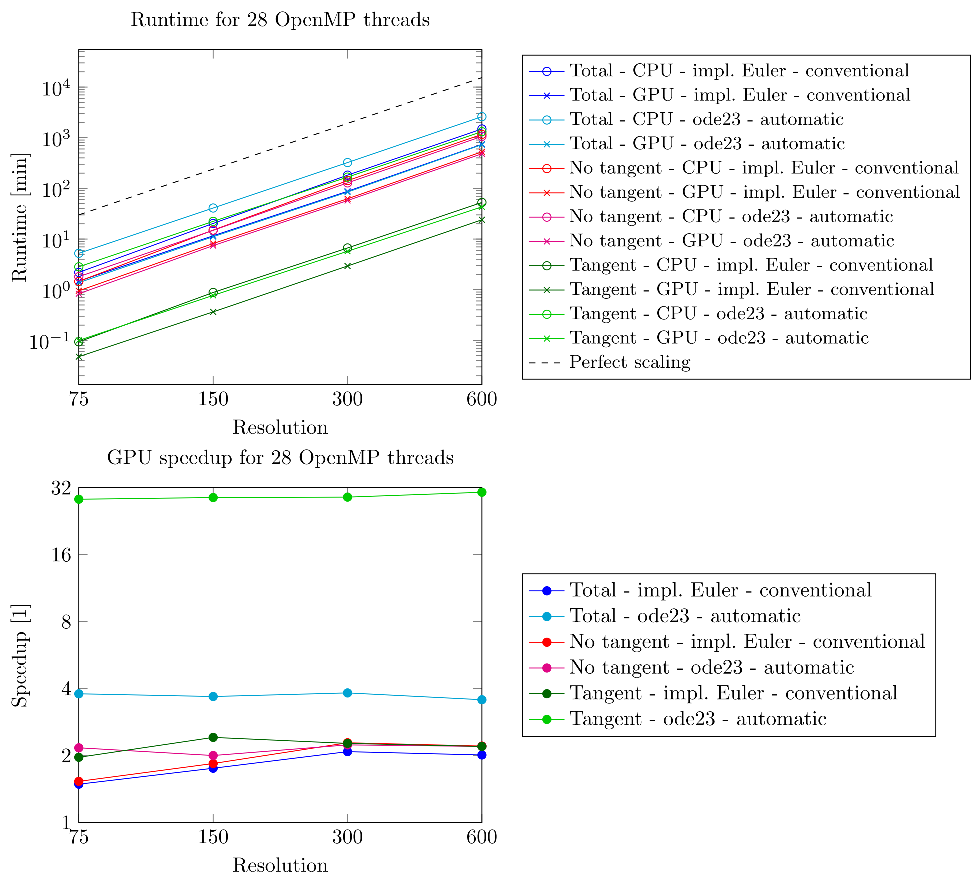


Fig. 19 Runtime and GPU speedup for varying geometry size. (Color figure online)

In our second performance study, we investigate the influence of the size of the geometry on the GPU speedup. The four different geometries shown in Fig. 18 were obtained by discretization of the analytical description of the (periodic) geometry at different resolutions $75 \times 75 \times 75$, $150 \times 150 \times 150$, $300 \times 300 \times 300$, and $600 \times 600 \times 600$. The resulting runtimes are plotted in Fig. 19. As expected, the GPU speedup is mostly independent of the size of the geometry.

When using not only one but g GPUs, the assumption of perfect GPU scaling

$$\mathrm{GPU}(g) = \frac{\mathrm{GPU}(1)}{g}$$

leads to the following similar relation

$$\frac{\mathrm{tot}_{\mathrm{CPU}}(1)}{\mathrm{GPU}(1) \cdot \left(\frac{t}{g} - 1\right) + \mathrm{tot}^{\mathrm{CPU}}_{\mathrm{GPU}}(1,\ 1)}$$

for the speedup due to using the CPU with t OpenMP threads and g GPUs compared to using only the CPU with t OpenMP threads. Our approach can be extended to multiple GPUs by means of an MPI implementation with which we hope to observe this behaviour. The details of this are beyond the scope of this paper and will be discussed in a follow-up publication.

8 Conclusion

In this article, we have introduced and studied a universal method for evaluating GSMs. With automatic differentiation, the material law setup is reduced to the implementation of two potentials. This eliminates the inconvenience of hand-computed derivatives and greatly simplifies the material law implementation process.

In a first step, we automatized the conventional implicit Euler approach and were able to reproduce the solution of the elasticity problem up to machine accuracy. However, we also demonstrated that its tangent computation is subject to general accuracy issues. As these can be resolved by an integration of the evolution equation for the state variables with adaptive time step sizes, we detailed how blackbox automatic differentiation of Rosenbrock and Runge–Kutta methods must be modified in the presence of time step size control to obtain derivatives that are as accurate as the primal solution. Material law evaluations with adaptive time steps improved the solution accuracy of the elasticity problem significantly for large loading steps, especially when stress and stiffness error measures are used for time step size control. Thus, we have a method at hand to assess the time discretization error disregarding contributions from solving the evolution equation.

To make the method applicable to CT-scale problems, we finally moved the material law evaluation to the GPU. Various kinds of overlap resulted in runtimes for the stress response that are independent of the chosen integration scheme and are moreover much faster than our conventional implementation on the CPU. Especially automatic evaluation strategies are accelerated significantly, which would not be possible without our efficient implementation of automatic differentiation on the GPU. For example, using ode23, the overall simulation with material law evaluations on the GPU was over 7 times faster for sequential CPU usage and still almost 4 times faster for 28 OpenMP threads. In our test scenario, automatic material law evaluation becomes feasible due to using the GPU. In future work, further material laws will be considered. In general, the speedups are material law specific and driven by the ratio between computational complexity and memory footprint.

We conclude that the framework for integrating GSMs into mechanical solvers presented in this article is promising due to its simultaneous flexibility, accuracy and performance. It is particularly well suited to improve and accelerate matrix-free solvers like FFT-based homogenization. The resulting user-friendly and fast method is a useful tool for the investigation of the non-linear material behavior of composites on a single workstation.

Acknowledgements The authors are deeply indebted to Heiko Andrä for numerous stimulating discussions on the topic of this article. M. Kabel thanks Quentin Horn for setting up the GPU benchmarking framework of AutoMat during his internship at Fraunhofer ITWM. The authors would like to thank the anonymous reviewers for their detailed remarks and valuable suggestions.

Funding Open Access funding enabled and organized by Projekt DEAL.

References

1. Alexe M, Sandu A (2009a) Forward and adjoint sensitivity analysis with continuous explicit Runge–Kutta schemes. Appl Math Comput 208(2):328–346. https://doi.org/10.1016/j.amc.2008.11.035
2. Alexe M, Sandu A (2009b) On the discrete adjoints of adaptive time stepping algorithms. J Comput Appl Math 233(4):1005–1020. https://doi.org/10.1016/j.cam.2009.08.109
3. Arbenz P, van Lenthe GH, Mennel U, Müller R, Sala M (2008) A scalable multi-level preconditioner for matrix-free μ-finite element analysis of human bone structures. Int J Numer Meth Eng 73(7):927–947. https://doi.org/10.1002/nme.2101
4. Arbenz P, Flaig C, Kellenberger D (2014) Bone structure analysis on multiple GPGPUs. J Parallel Distrib Comput 74:2941–2950. https://doi.org/10.1016/j.jpdc.2014.06.014
5. Arya VK, Hornberger K, Stamm H (1986). On the numerical integration of viscoplastic models. https://doi.org/10.5445/IR/270022902
6. Auroux D, Groza V (2017) Optimal parameters identification and sensitivity study for abrasive waterjet milling model. Inverse Probl Sci Eng 25(11):1560–1576. https://doi.org/10.1080/17415977.2016.1273916
7. Bogacki P, Shampine LF (1989) A 3(2) pair of Runge–Kutta formulas. Appl Math Lett 2(4):321–325. https://doi.org/10.1016/0893-9659(89)90079-7
8. Brisard S, Dormieux L (2010) FFT-based methods for the mechanics of composites: A general variational framework. Comput Mater Sci 49(3):663–671. https://doi.org/10.1016/j.commatsci.2010.06.009
9. Brisard S, Dormieux L (2012) Combining Galerkin approximation techniques with the principle of Hashin and Shtrikman to derive a new FFT-based numerical method for the homogenization of composites. Comput Methods Appl Mech Eng 217–220:197–212. https://doi.org/10.1016/j.cma.2012.01.003
10. Chen Y, Gélébart L, Chateau C, Bornert M, Sauder C, King A (2019a) Analysis of the damage initiation in a SiC/SiC composite tube from a direct comparison between large-scale numerical simulation and synchrotron X-ray micro-computed tomography. Int J Solids Struct 161:111–126. https://doi.org/10.1016/j.ijsolstr.2018.11.009
11. Chen Y, Vasiukov D, Gélébart L, Park CH (2019b) A FFT solver for variational phase-field modeling of brittle fracture. Comput Methods Appl Mech Eng 349:167–190. https://doi.org/10.1016/j.cma.2019.02.017
12. Cormeau I (1975) Numerical stability in quasi-static elasto/visco-plasticity. Int J Numer Meth Eng 9(1):109–127. https://doi.org/10.1002/nme.1620090110
13. Eberhard P, Bischof C (1999) Automatic differentiation of numerical integration algorithms. AMS Math Comput 68(226):717–731. https://doi.org/10.1090/S0025-5718-99-01027-3
14. Eisenlohr P, Diehl M, Lebensohn RA, Roters F (2013) A spectral method solution to crystal elasto-viscoplasticity at finite strains. Int J Plast 46:37–53. https://doi.org/10.1016/j.ijplas.2012.09.012
15. Enciu P, Gerbaud L, Wurtz F (2010) Automatic differentiation applied for optimization of dynamical systems. IEEE Trans Magn 46(8):2943–2946. https://doi.org/10.1109/TMAG.2010.2044770
16. Fritzen F, Böhlke T (2013) Reduced basis homogenization of viscoelastic composites. Compos Sci Technol 76:84–91. https://doi.org/10.1016/j.compscitech.2012.12.012
17. Gauger NR, Walther A, Moldenhauer C, Widhalm M (2008) Automatic Differentiation of an Entire Design Chain for Aerodynamic Shape Optimization. In: Tropea C, Jakirlic S, Heinemann HJ, Henke R, Hönlinger H (eds) New Results in Numerical and Experimental Fluid Mechanics VI, Springer Berlin Heidelberg,

Berlin, Heidelberg, pp 454–461, https://doi.org/10.1007/978-3-540-74460-3_56
18. Gélébart L, Mondon-Cancel R (2013) Non-linear extension of FFT-based methods accelerated by conjugate gradients to evaluate the mechanical behavior of composite materials. Comput Mater Sci 77:430–439. https://doi.org/10.1016/j.commatsci.2013.04.046
19. Griewank A, Walther A (2008) Evaluating derivatives: principles and techniques of algorithmic differentiation, vol 105. Siam, Philadelphia. https://doi.org/10.1137/1.9780898717761
20. Grimm-Strele H, Kabel M (2019) Runtime optimization of a memory efficient CG solver for FFT-based homogenization: implementation details and scaling results for linear elasticity. Comput Mech 64(5):1339–1345. https://doi.org/10.1007/s00466-019-01713-3
21. Günther S, Ruthotto L, Schroder JB, Cyr EC, Gauger NR (2020) Layer-parallel training of deep residual neural networks. SIAM J Math Data Sci 2(1):1–23. https://doi.org/10.1137/19M1247620
22. Hairer E, Wanner G (2010) Solving ordinary differential equations II: stiff and differential-algebraic problems, 2nd edn. Springer, Berlin. https://doi.org/10.1007/978-3-642-05221-7
23. Hairer E, Nørsett SP, Wanner G (1993) Solving ordinary differential equations I: Nonstiff Problems, 2nd edn. Springer, Berlin. https://doi.org/10.1007/978-3-540-78862-1
24. Halphen B, Nguyen QS (1975) Sur les matériaux standard généralisés. Journal de mécanique 14:39–63
25. Hiley RA, Rouainia M (2008) Explicit Runge–Kutta methods for the integration of rate-type constitutive equations. Comput Mech 42(1):53. https://doi.org/10.1007/s00466-007-0234-2
26. Hogan RJ (2014) Fast reverse-mode automatic differentiation using expression templates in C++. ACM Transactions on Mathematical Software 40(4), https://doi.org/10.1145/2560359
27. Kabel M, Böhlke T, Schneider M (2014) Efficient fixed point and Newton-Krylov solvers for FFT-based homogenization of elasticity at large deformations. Comput Mech 54(6):1497–1514. https://doi.org/10.1007/s00466-014-1071-8
28. Kabel M, Fliegener S, Schneider M (2016) Mixed boundary conditions for FFT-based homogenization at finite strains. Comput Mech 57(2):193–210. https://doi.org/10.1007/s00466-015-1227-1
29. Kabel M, Fink A, Schneider M (2017) The composite voxel technique for inelastic problems. Comput Methods Appl Mech Eng 322:396–418. https://doi.org/10.1016/j.cma.2017.04.025
30. Kiran U, Sharma D, Gautam SS (2019) GPU-warp based finite element matrices generation and assembly using coloring method. Journal of Computational Design and Engineering 6(4):705–718 https://doi.org/10.1016/j.jcde.2018.11.001
31. Leppkes K, Lotz J, Naumann U (2016) Derivative code by overloading in C++ (dco/c++): Introduction and summary of features. http://aib.informatik.rwth-aachen.de/2016/2016-08.pdf
32. Leppkes K, Lotz J, Naumann U, Du Toit J (2017) Meta Adjoint Programming in C++. http://aib.informatik.rwth-aachen.de/2017/2017-07.pdf
33. Leuschner M, Fritzen F (2018) Fourier-Accelerated Nodal Solvers (FANS) for homogenization problems. Comput Mech 62(3):359–392. https://doi.org/10.1007/s00466-017-1501-5
34. Lucarini S, Segurado J (2019) On the accuracy of spectral solvers for micromechanics based fatigue modeling. Comput Mech 63(2):365–382. https://doi.org/10.1007/s00466-018-1598-1
35. Ma R, Truster TJ (2019) FFT-based homogenization of hypoelastic plasticity at finite strains. Comput Methods Appl Mech Eng 349:499–521. https://doi.org/10.1016/j.cma.2019.02.037
36. Macioł P, Płaszewski P, Banaś K (2010) 3D finite element numerical integration on GPUs. Procedia Comput Sci 1(1):1093–1100. https://doi.org/10.1016/j.procs.2010.04.121
37. Michel JC, Suquet P (2016) A model-reduction approach in micromechanics of materials preserving the variational structure of constitutive relations. J Mech Phys Solids 90:254–285. https://doi.org/10.1016/j.jmps.2016.02.005
38. Michel JC, Moulinec H, Suquet P (2001) A computational scheme for linear and non-linear composites with arbitrary phase contrast. Int J Numer Meth Eng 52(12):139–160. https://doi.org/10.1002/nme.275
39. Miehe P, Sandu A (2006) Forward, Tangent Linear, and Adjoint Runge-Kutta Methods in KPP–2.2. In: Alexandrov VN, van Albada GD, Sloot PMA, Dongarra J (eds) Computational Science – ICCS 2006, Springer Berlin Heidelberg, Berlin, Heidelberg, pp 120–127, https://doi.org/10.1007/11758532_18
40. Moulinec H, Suquet P (1994) A fast numerical method for computing the linear and nonlinear mechanical properties of composites. Comptes rendus de l'Académie des sciences Série II, Mécanique, physique, chimie, astronomie 318(11):1417–1423
41. Moulinec H, Suquet P (1998) A numerical method for computing the overall response of nonlinear composites with complex microstructure. Comput Methods Appl Mech Eng 157(1–2):69–94. https://doi.org/10.1016/S0045-7825(97)00218-1
42. Ohshima S, Hayashi M, Katagiri T, Nakajima K (2013) Implementation and Evaluation of 3D Finite Element Method Application for CUDA. In: Daydé M, Marques O, Nakajima K (eds) High Performance Computing for Computational Science - VECPAR 2012, Lecture Notes in Computer Science, vol 7851, Springer Berlin Heidelberg, Berlin, Heidelberg, pp 140–148, https://doi.org/10.1007/978-3-642-38718-0_16
43. Phipps E, Pawlowski R (2012) Efficient Expression Templates for Operator Overloading-Based Automatic Differentiation. In: Forth S, Hovland P, Phipps E, Utke J, Walther A (eds) Recent Advances in Algorithmic Differentiation, Springer, Berlin, Heidelberg, pp 309–319, https://doi.org/10.1007/978-3-642-30023-3_28
44. Prüss JW, Wilke M (2010) Gewöhnliche Differentialgleichungen und dynamische Systeme. Springer, Berlin. https://doi.org/10.1007/978-3-0348-0002-0
45. Rothe S, Hartmann S (2014) Automatic differentiation for stress and consistent tangent computation. Arch Appl Mech 85(8):1103–1125. https://doi.org/10.1007/s00419-014-0939-6
46. Sagebaum M, Albring T, Gauger NR (2018) Expression templates for primal value taping in the reverse mode of algorithmic differentiation. Optim Methods Soft 33(4–6):1207–1231. https://doi.org/10.1080/10556788.2018.1471140
47. Sagebaum M, Albring T, Gauger NR (2019) High-performance derivative computations using CoDiPack. ACM Trans Math Soft. https://doi.org/10.1145/3356900
48. Sandu A (2006) On the Properties of Runge-Kutta Discrete Adjoints. In: Alexandrov VN, van Albada GD, Sloot PMA, Dongarra J (eds) Computational Science – ICCS 2006, Springer Berlin Heidelberg, Berlin, Heidelberg, pp 550–557, https://doi.org/10.1007/11758549_76
49. Sandu A (2008) Reverse automatic differentiation of linear multistep methods. In: Bischof CH, Bücker HM, Hovland P, Naumann U, Utke J (eds) Advances in automatic differentiation. Springer Berlin Heidelberg, Berlin, Heidelberg, pp 1–12. https://doi.org/10.1007/978-3-540-68942-3_1
50. Schlenkrich S, Walther A, Gauger NR, Heinrich R (2008) Differentiating fixed point iterations with ADOL-C: Gradient calculation for fluid dynamics. In: Modeling, Simulation and Optimization of Complex Processes, Springer, pp 499–508, https://doi.org/10.1007/978-3-540-79409-7_36
51. Schneider M (2015) Convergence of FFT-based homogenization for strongly heterogeneous media. Math Methods Appl Sci 38(13):2761–2778. https://doi.org/10.1002/mma.3259
52. Schneider M (2019) On the Barzilai-Borwein basic scheme in FFT-based computational homogenization. Int J Numer Meth Eng 118(8):482–494. https://doi.org/10.1002/nme.6023
53. Schneider M (2020) A dynamical view of nonlinear conjugate gradient methods with applications to FFT-based computational

micromechanics. Comput Mech. https://doi.org/10.1007/s00466-020-01849-7
54. Schneider M, Ospald F, Kabel M (2016) Computational homogenization of elasticity on a staggered grid. Int J Numer Meth Eng 105(9):693–720. https://doi.org/10.1002/nme.5008
55. Schneider M, Merkert D, Kabel M (2017) FFT-based homogenization for microstructures discretized by linear hexahedral elements. Int J Numer Meth Eng 109(10):1461–1489. https://doi.org/10.1002/nme.5336
56. Shampine LF, Reichelt MW (1997) The MATLAB ODE Suite. SIAM J Sci Comput 18(1):1–22. https://doi.org/10.1137/S1064827594276424
57. Shanthraj P, Eisenlohr P, Diehl M, Roters F (2015) Numerically robust spectral methods for crystal plasticity simulations of heterogeneous materials. Int J Plast 66:31–45. https://doi.org/10.1016/j.ijplas.2014.02.006
58. Simo JC, Hughes TJR (2006) Computational inelasticity, interdisciplinary applied mathematics, vol 7. Springer, Berlin. https://doi.org/10.1007/b98904
59. Simo JC, Taylor RL (1985) Consistent tangent operators for rate-independent elastoplasticity. Comput Methods Appl Mech Eng 48(1):101–118. https://doi.org/10.1016/0045-7825(85)90070-2
60. Utke J, Naumann U, Fagan M, Tallent N, Strout M, Heimbach P, Hill C, Wunsch C (2008) OpenAD/F: A Modular Open-Source Tool for Automatic Differentiation of Fortran Codes. ACM Trans Math Soft. https://doi.org/10.1145/1377596.1377598
61. Voigt W (1966) Lehrbuch der Kristallphysik. Vieweg+Teubner Verlag, Wiesbaden, https://doi.org/10.1007/978-3-663-15884-4
62. Walther A (2007) Automatic differentiation of explicit Runge–Kutta methods for optimal control. Comput Optim Appl 36(1):83–108. https://doi.org/10.1007/s10589-006-0397-3
63. Walther A (2009) Getting Started with ADOL-C. In: Uwe Naumann, Olaf Schenk, Horst D Simon, Sivan Toledo (eds) Combinatorial Scientific Computing, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany, Dagstuhl, Germany, Dagstuhl Seminar Proceedings, http://drops.dagstuhl.de/opus/volltexte/2009/2084
64. Wicht D, Schneider M, Böhlke T (2019) On Quasi-Newton methods in fast Fourier transform-based micromechanics. Int J Numer Meth Eng 318(11):1417. https://doi.org/10.1002/nme.6283
65. Willot F (2015) Fourier-based schemes for computing the mechanical response of composites with accurate local fields. Comptes Rendus Mécanique 343(3):232–245. https://doi.org/10.1016/j.crme.2014.12.005
66. Wu PY (1988) Products of positive semidefinite matrices. Linear Algebra Appl 111:53–61. https://doi.org/10.1016/0024-3795(88)90051-1
67. Zeman J, Vondřejc J, Novák J, Marek I (2010) Accelerating a FFT-based solver for numerical homogenization of periodic media by conjugate gradients. J Comput Phys 229(21):8065–8071. https://doi.org/10.1016/j.jcp.2010.07.010