FINITE ELEMENT CENTER

PREPRINT 2006–06

Algorithms and Data Structures for Multi-Adaptive Time-Stepping

Johan Jansson and Anders Logg

FINITE ELEMENT CENTER

PREPRINT 2006–06

Algorithms and Data Structures for Multi-Adaptive Time-Stepping

Johan Jansson and Anders Logg

Finite Element Center
`http://www.femcenter.org/`

Algorithms and Data Structures for
Multi-Adaptive Time-Stepping
Johan Jansson and Anders Logg

Finite Element Center Preprint
NO 2006–06
ISSN 1653–574X

ALGORITHMS AND DATA STRUCTURES FOR MULTI-ADAPTIVE TIME-STEPPING

JOHAN JANSSON AND ANDERS LOGG

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

1. Introduction

We have earlier in a sequence of papers [33, 34, 37] introduced the multi-adaptive Galerkin methods mcG(q) and mdG(q) for the approximate (numerical) solution of ODEs of the form

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

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

The multi-adaptive Galerkin methods mcG(q) and mdG(q) extend the standard mono-adaptive continuous and discontinuous Galerkin methods cG(q) and dG(q), studied earlier in detail in [27, 26, 28, 4, 12, 29, 7, 8, 9, 10, 11, 6, 13, 14, 15, 17, 16], by allowing individual time step sequences $k_i = k_i(t)$ for the different components $U_i = U_i(t)$, $i = 1, 2, \ldots, N$, of the approximate solution $U \approx u$ of the initial value problem (1.1). For related work on local time-stepping, see also [24, 25, 38, 2, 1, 39, 18, 3, 32, 40].

Date: August 18, 2006.

Key words and phrases. Multi-adaptivity, individual time steps, local time steps, ODE, continuous Galerkin, discontinuous Galerkin, mcgq, mdgq, C++, implementation, algorithms.

Johan Jansson, Department of Computational Technology, Chalmers University of Technology, SE–412 96 Göteborg, Sweden. *Email*: `johanjan@math.chalmers.se`.

Anders Logg, Simula Research Laboratory, PO Box 134 NO–1325 Lysaker, Norway. *Email:* `logg@simula.no`.

In the current paper, we discuss important aspects of the implementation of multi-adaptive Galerkin methods. While earlier results on multi-adaptive time-stepping presented in [33, 34, 37] include the basic formulation of the methods, a priori and a posteriori error estimates, together with a proof-of-concept implementation and results for a number of model problems, the current paper addresses the important issue of efficiently implementing the multi-adaptive methods with minimal overhead as compared to standard mono-adaptive solvers. For many problems, in particular when the propagation of the solution is local in space and time, the potential speedup of multi-adaptivity is large, but the actual speedup may be far from the ideal speedup if the overhead of the more complex implementation is significant.

1.1. **Implementation.**

The algorithms presented in this paper are implemented by the multi-adaptive ODE-solver available in DOLFIN [22, 23], Dynamic Object-oriented Library for FINite element computation, which is the C++ interface of the new open-source software project FEniCS [21, 5] for the automation of Computational Mathematical Modeling (CMM). The multi-adaptive solver in DOLFIN is based on the original implementation Tanganyika, presented in [34], but has been completely rewritten for DOLFIN and is actively developed by the authors.

1.2. **Obtaining the software.**

DOLFIN is licensed under the GNU General Public License [19], which means that anyone is free to use or modify the software, provided these rights are preserved. The complete source code of DOLFIN, including numerous example programs, is available at the DOLFIN web page [22].

1.3. **Notation.**

The following notation is used throughout this paper: Each component $U_i(t)$, $i = 1, \ldots, N$, of the approximate m(c/d)G(q) solution $U(t)$ of (1.1) is a piecewise polynomial on a partition of $(0, T]$ into M_i sub intervals. Sub interval j for component i is denoted by $I_{ij} = (t_{i,j-1}, t_{ij}]$, and the length of the sub interval is given by the local *time step* $k_{ij} = t_{ij} - t_{i,j-1}$. We shall sometimes refer to I_{ij} as an *element*. This is illustrated in Figure 1. On each sub interval I_{ij}, $U_i|_{I_{ij}}$ is a polynomial of degree q_{ij}.

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

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

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

1.4. **Outline of the paper.**

We first give an introduction to multi-adaptive time-stepping in Section 2. We then present the key algorithms used by the multi-adaptive ODE solver of DOLFIN in Section 3, followed by a discussion of data structures for efficient representation and interpolation of multi-adaptive solutions in Section 4. In Section 5, we discuss the efficiency of multi-adaptive time-stepping and in Section 6, we present a number of numerical examples that demonstrate the efficiency of the proposed algorithms and data structures. Finally, we give some concluding remarks in Section 7.

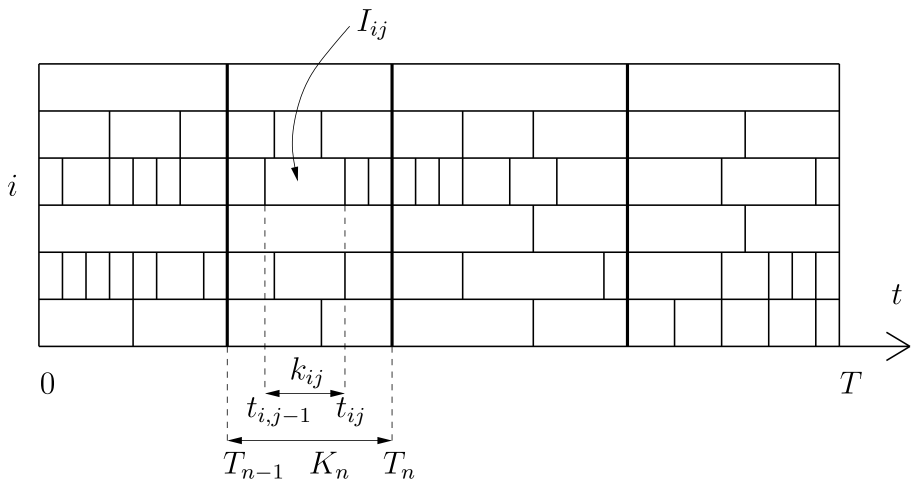

FIGURE 1. Individual partitions of the interval $(0, T]$ for different components. Elements between common synchronized time levels are organized in time slabs. In this example, we have $N = 6$ and $M = 4$.

2. Multi-adaptive time-stepping

In this section, we give a quick introduction to multi-adaptive time-stepping, including the formulation of the methods, error estimates and adaptivity. For a more detailed account, we refer the reader to [33, 34, 37].

2.1. **Formulation of the methods.** Just as the standard cG(q) and dG(q) methods, the multi-adaptive mcG(q) and mdG(q) methods are obtained by multiplying the system of equations (1.1) with a suitable test function v, to obtain a variational problem of the form: Find $U \in V$ with $U(0) = u_0$, such that

$$\int_0^T (v, \dot{U})\,\mathrm{d}t = \int_0^T (v, f(U, \cdot))\,\mathrm{d}t \quad \forall v \in \hat{V}, \tag{2.1}$$

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

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

For the standard dG(q) method, the test and trial spaces are equal and consist of the space of (possibly discontinuous) piecewise polynomial vector-valued functions of degree $q = q(t)$ on a partition $0 = t_0 < t_1 < \cdots < t_M = T$, which extends naturally to the multi-adaptive mdG(q) method by allowing each component of the test and trial functions to be piecewise polynomial on individual partitions of the time interval as above. Note that for both the dG(q) method and the mdG(q) method, the integral $\int_{0,T}(v, \dot{U})\,\mathrm{d}t$ in (2.1) must be treated appropriately at the points of discontinuity, see [33].

Both in the case of the mcG(q) and mdG(q) methods, the variational problem (2.1) gives rise to a system of discrete equations by expanding the solution U in a suitable basis on each local interval I_{ij},

$$U_i|_{I_{ij}} = \sum_{m=0}^{q_{ij}} \xi_{ijm}\phi_{ijm}, \tag{2.2}$$

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

$$\xi_{ijm} = \xi_{ij0}^- + k_{ij} \sum_{n=0}^{q_{ij}} w_{mn}^{[q_{ij}]}\ f_i(U(\tau_{ij}^{-1}(s_n^{[q_{ij}]})), \tau_{ij}^{-1}(s_n^{[q_{ij}]})), \tag{2.3}$$

for $m = 0, \ldots, q_{ij}$, where $\{w_{mn}^{[q_{ij}]}\}_{m=0,n=0}^{q_{ij}}$ are weights, τ_{ij} maps I_{ij} to $(0,1]$: $\tau_{ij}(t) = (t - t_{i,j-1})/(t_{ij} - t_{i,j-1})$, and $\{s_n^{[q_{ij}]}\}_{n=0}^{q_{ij}}$ are quadrature points defined on $[0,1]$. Note that we have here assumed that the number of quadrature points is equal to the number of nodal points. See [33] for a discussion of suitable quadrature rules and basis functions.

2.2. **Error estimates and adaptivity.** In [33], a posteriori error estimates are proved for the multi-adaptive mcG(q) and mdG(q) methods. For the mcG(q) method, the estimate takes the form

$$|M(e)| \leq E \equiv \sum_{i=1}^{N} S_i(T) \max_{[0,T]} \{C_i k_i^{q_i} |R_i|\}, \tag{2.4}$$

with a similar estimate for the mdG(q) method. Here, $M : \mathbb{R}^N \to \mathbb{R}$ denotes some given functional of the global error $e = U - u$ to be estimated, $R = \dot{U} - f(U, \cdot)$ denotes the *residual* of the computed solution, $C_i = C_i(t)$ denotes an interpolation constant (which may be different for each local interval) and $S_i(T)$ denotes a *stability factor* that measures the rate of propagation of errors for component U_i (the influence of a nonzero residual in component U_i on the size of the error in the given functional). Comparing to standard Runge–Kutta methods for the solution of initial value problems, the stability factor is the missing link between the "local error" and the global error. Note that alternatively, the stability information may be kept as a local time-dependent *stability weight* for more fine-grained control of the contributions to the global error. The stability factors are obtained from solving the *dual problem* of (1.1) for the given functional M, see [6, 33].

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

$$C_{ij}k_{ij}^{q_{ij}} \max_{I_{ij}} |R_i| = \mathrm{TOL}/(NS_i(T)), \tag{2.5}$$

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

(0) Assume $S_i(T) = 1$ for $i = 1, 2, \ldots, N$;
(i) Solve the primal problem with time steps based on (2.5);
(ii) Solve the dual problem and compute the stability factors;
(iii) Compute an error bound E based on (2.4);
(iv) If $E \leq \mathrm{TOL}$ then stop; if not go back to (i).

3. Algorithms

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

3.1. **General algorithm.** The general multi-adaptive time-stepping algorithm is Algorithm 1. Starting at $t = 0$, the algorithm creates a sequence of time slabs until the given end time T is reached. The end time T is given as an argument to CreateTimeSlab (Algorithm 2), which creates a time slab covering an interval $[T_{n-1}, T_n]$ such that $T_n \leq T$. CreateTimeSlab returns the end time T_n of the created time slab and the integration continues until $T_n = T$. For each time slab, the system of discrete equations is solved iteratively, using direct fixed-point iteration or a preconditioned Newton's method, until the discrete equations given by the mcG(q) or mdG(q) method have converged.

Algorithm 1 U = Integrate(ODE)

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

The basic forward integrator, Algorithm 1, can be used as the main component of an adaptive algorithm with automated error control of the computed solution as outlined in Section 2. In each iteration, the *primal problem* (1.1) is solved using Algorithm 1. An ODE of the form (1.1) representing the *dual problem* is then created and solved using Algorithm 1. It is important to note that both the primal and the dual problems are solved using the same algorithm, but with different time steps and, possibly, different tolerances, methods, and orders. When the solution of the dual problem has been computed, the stability factors $\{S_i(T)\}_{i=1}^N$ and the error estimate can be computed.

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

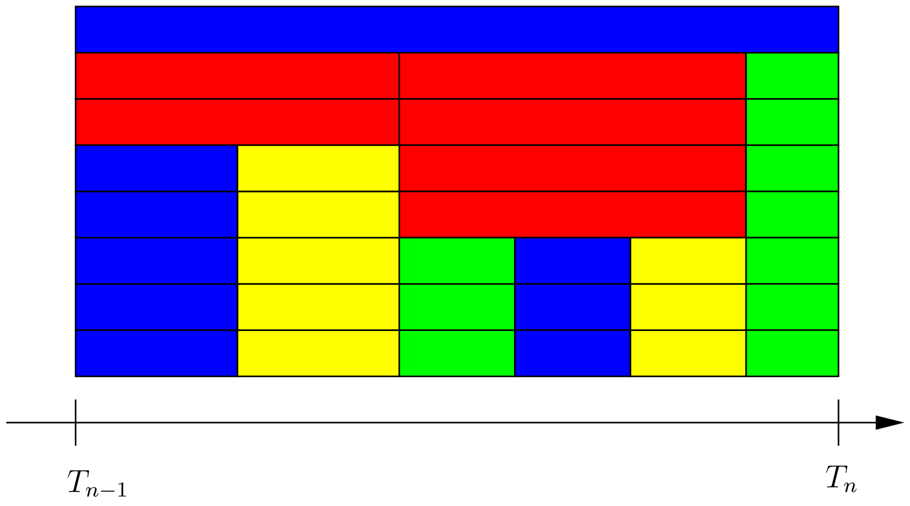

FIGURE 2. The recursive organization of the time slab. Each time slab contains an element group and a list of recursively nested time slabs. The root time slab in the figure contains one element group of one element and three sub slabs. The first of these sub slabs contains an element group of two elements and two nested sub slabs, and so on. The root time slab recursively contains a total of nine element groups and 33 elements.

To create a time slab, we first compute the desired time steps for all components as given by the a posteriori error estimate (2.4). We discuss in detail the time step selection below in Section 3.4. A threshold θK is then computed based on the maximum time step K and a fixed parameter $\theta \in (0, 1)$ controlling the density of the time slab. The components are partitioned into two sets based on the threshold and a suitable large time step $\underline{K}$ is selected as in Figure 3. For each component in the group with large time steps, an element is created and added to the element group of the time slab. The remaining components with small time steps are processed by a recursive application of this algorithm for the construction of time slabs.

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

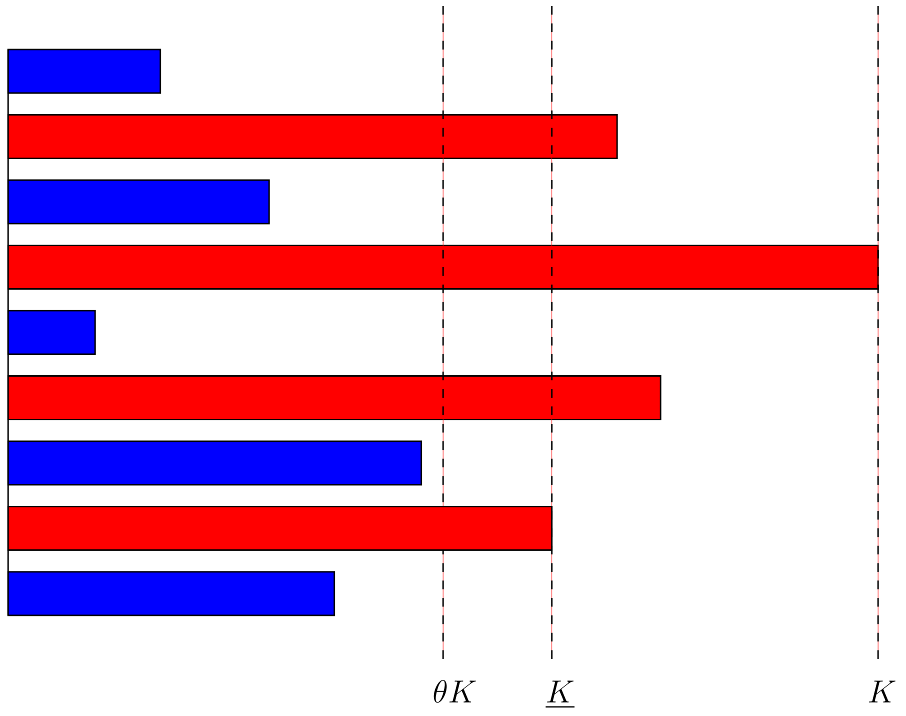

FIGURE 3. The partition of components into groups of small and large time steps for $\theta = 1/2$.

breadth-first, which means in particular that the element for the component with the largest time step is created first.

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

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

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

3.3. **Solving the system of discrete equations.** On each time slab $\mathcal{T}_n$, $n = 1, 2, \ldots, M$, we need to solve a system of equations for the degrees of freedom on the time slab. On each local interval $I_{ij} \in \mathcal{T}_n$, these equations are given by (2.3). Depending on the properties of

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

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

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

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

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

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

the given system (1.1), different solution strategies for the time slab system (2.3) may be appropriate as outlined below.

3.3.1. *Direct fixed-point iteration.* In the simplest case, the time slab system is solved by direct fixed-point iteration on (2.3) for each element in the time slab. The fixed-point iteration is performed in a forward fashion, sweeping over the elements in the time slab in the same order as they are created by Algorithm 2. In particular, this means that for each component in the time slab system, the end-time value on each element is updated before the degrees of freedom for the following element. Thus, for each element $I_{ij} \in \mathcal{T}_n$, we compute the degrees of freedom $\{\xi_{ijm}\}_{j=0}^{q_{ij}}$ according to

$$\xi_{ijm} = \xi_{ij0}^{-} + k_{ij} \sum_{n=0}^{q_{ij}} w_{mn}^{[q_{ij}]} \, f_i(U(\tau_{ij}^{-1}(s_n^{[q_{ij}]})), \tau_{ij}^{-1}(s_n^{[q_{ij}]})), \quad m = 0, 1, \ldots, q_{ij}. \tag{3.1}$$

Direct fixed-point iteration converges if the system is non-stiff and typically only a few iterations are needed. In fact, one may define a system as being stiff if direct fixed-point iteration does not converge.

3.3.2. *Damped fixed-point iteration.* If the system is stiff, that is, direct fixed-point iteration does not converge, one may introduce a suitable amount of damping to adaptively stabilize the fixed-point iteration. The fixed-point iteration (3.1) may be written in the form

$$\xi_{ijm} = g_{ijm}(\xi), \tag{3.2}$$

where ξ is the vector (or tensor) of degrees of freedom for the solution on the time slab. We modify the fixed-point iteration by introducing a damping parameter α:

$$\xi_{ijm} = (1 - \alpha_{ijm})\xi_{ijm} + \alpha_{ijm} g_{ijm}(\xi). \tag{3.3}$$

In [35], a number of different strategies for the selection of the damping parameter α are discussed. We mention two of these strategies here. The first strategy chooses α based on the diagonal derivatives $\partial f_i / \partial u_i$, $i = 1, 2, \ldots, N$, corresponding to a Newton's method with a diagonal Jacobian. This strategy works well for systems with a diagonally dominant Jacobian, including many systems of chemical reactions. The second strategy adaptively chooses a scalar α based on the convergence of the fixed-point iterations. With a suitable sequence combining large and small values of α, one may be improve the convergence rate well beyond the limit given by a constant scalar $\alpha \sim 1/||g'||$.

3.3.3. *Newton's method.* Alternatively, one may apply Newton's method directly to the full system of equations on the time slab, given on each element by 3.1. The linear system in each Newton iteration may then be solved either by a direct method or an iterative method such as a Krylov subspace method in combination with a suitable preconditioner, depending on the characteristics of the underlying system (1.1). In addition, one may also apply a special preconditioner that improves the convergence by propagating values forward in time within the time slab. Note that if the multi-adaptive efficiency index is large (see Section 5 below), then the time slab system is not significantly larger than the corresponding time slab system for a mono-adaptive method.

3.3.4. *Choosing a solution strategy.* Ultimately, an intelligent solver should automatically choose a suitable algorithm for the solution of the time slab system. Thus, the solver may initially try direct fixed-point iteration. If the system is stiff, the solver switches to adaptive fixed-point iteration (as outlined in [35]). Finally, if the adaptive fixed-point iteration converges slowly, the solver may switch to Newton's method.

3.3.5. *Interpolation of the solution.* To update the degrees of freedom on an element according to (3.1), the appropriate component f_i of the right-hand side of (1.1) needs to be evaluated at the set of quadrature points. In order for f_i to be evaluated, each component $U_{i'}$ of the computed solution U on which f_i depends has to be evaluated at the quadrature points. We let $\mathcal{S}_i \subseteq \{1, \ldots, N\}$ denote the *sparsity pattern* of component U_i, that is, the set of components on which f_i depends,

$$\mathcal{S}_i = \{i' \in \{1, \ldots, N\} : \partial f_i / \partial u_{i'} \neq 0\}. \tag{3.4}$$

Thus, to evaluate f_i at a given quadrature point t, only the components $\{U_{i'}\}_{i' \in \mathcal{S}_i}$ need to be evaluated at t, as in Algorithm 6. This is of particular importance for problems of sparse structure and enables efficient multi-adaptive integration of time-dependent PDEs, as demonstrated below in Section 6. The sparsity pattern $\mathcal{S}_i$ is automatically detected by the solver. Alternatively, the sparsity pattern can be specified by a (sparse) matrix.

Algorithm 6 y = EvaluateRightHandSide(i, t)

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

In Algorithm 6, the key step is the evaluation of a component $U_{i'}$ at a given point t. For a standard mono-adaptive method, this is straightforward since all components use the same time steps. In particular, if the quadrature points are chosen to be the same as the nodal points, the value of $U_{i'}(t)$ is already known. For a multi-adaptive method, a quadrature point t for the evaluation of f_i is not necessarily a nodal point for $U_{i'}$. To evaluate $U_{i'}(t)$, one thus needs to find the local interval $I_{i'j'}$ such that $t \in I_{i'j'}$ and then evaluate $U_{i'}(t)$ by interpolation on that interval. In Section 4 below, we discuss data structures that allow efficient storage and interpolation of the multi-adaptive solution. In particular, these data structures give $\mathcal{O}(1)$ access to the value of any component $U_{i'}$ in the sparsity pattern $\mathcal{S}_i$ at any quadrature point t for f_i.

3.4. **Multi-adaptive time step selection.** The individual and adaptive time steps k_{ij} are determined during the recursive construction of time slabs based on an a posteriori error estimate as discussed in Section 2. Thus, according to (2.5), each local time step k_{ij} should be chosen to satisfy

$$k_{ij} = \left(\frac{\mathrm{TOL}}{C_{ij} N S_i(T) \max_{I_{ij}} |R_i|} \right)^{1/q_{ij}}. \tag{3.5}$$

where TOL is a given tolerance.

However, the time steps can not be based directly on (3.5), since that leads to unwanted oscillations in the size of the time steps. If $r_{i,j-1} = \max_{I_{i,j-1}} |R_i|$ is small, then k_{ij} will be large, and as a result r_{ij} will also be large. Consequently, $k_{i,j+1}$ and $r_{i,j+1}$ will be small, and so on. To avoid these oscillations, we adjust the time step k_{ij} according to

Algorithm 7, which determines the new time step as a weighted harmonic mean value of the previous time step and the time step given by (3.5). Alternatively, DOLFIN provides time step control based on the (PID) controllers presented in [20, 41], including H0211 and H211PI. However, the simple controller of Algorithm 7 performs well compared to the more sophisticated controllers in [20, 41]. A suitable value for the weight w in Algorithm 7 is $w = 5$.

Algorithm 7 $k = \text{Controller}(k_{\text{new}},\, k_{\text{old}},\, k_{\text{max}})$

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

The initial time steps $k_{11} = k_{21} = \cdots = k_{N1} = K_1$ are chosen equal for all components and are determined iteratively for the first time slab. The size K_1 of the first time slab is first initialized to some default value, possibly based on the length T of the time interval, and then adjusted until the local residuals are sufficiently small for all components.

3.5. **Automatic generation of the dual problem.** The dual problem of (1.1) for $\varphi = \varphi(t)$ that is solved to obtain stability factors and error estimates is given by

$$\begin{aligned} -\dot{\varphi}(t) &= J(U,t)^\top \varphi(t), \quad t \in [0,T), \\ \varphi(T) &= \psi, \end{aligned} \tag{3.6}$$

where $J(U,t)$ denotes the Jacobian of the right-hand side f of (1.1) at time t and $\psi = M'$ is initial data for the dual problem corresponding to the given functional M to be estimated. Note that we need to linearize around the computed solution U, since the exact solution u of (1.1) is not known. To solve this backward problem over $[0,T)$ using the forward integrator Algorithm 1, we rewrite (3.6) as a forward problem. With $w(t) = \varphi(T-t)$, we have $\dot{w} = -\dot{\varphi}(T-t) = J^\top(U,T-t)w(t)$, and so (3.6) can be written as a forward problem for w in the form

$$\begin{aligned} \dot{w}(t) &= f^*(w(t),t) \equiv J(U,T-t)^\top w(t), \quad t \in (0,T], \\ w(0) &= \psi. \end{aligned} \tag{3.7}$$

4. Data structures

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

4.1. **Representing the solution.** The multi-adaptive solution on a time-slab can be efficiently represented using a data structure consisting of eight arrays as shown in Table 1. For simplicity, we assume that all elements in a time slab are constructed for the same choice of method, mcG(q) or mdG(q), for a given fixed q.

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

$$\begin{aligned} a_s &= \texttt{sa[}s\texttt{]}, \\ b_s &= \texttt{sb[}s\texttt{]}. \end{aligned} \tag{4.1}$$

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

For each element, we store the corresponding component index i in the array `ei` in order to be able to evaluate the correct component f_i of the right-hand side f of (1.1) when iterating over all elements in the time slab to update the degrees of freedom. When updating the values on an element according to (2.3), it is also necessary to know the left and right end-points of the elements. Thus, we store an array `es` that maps the number e of a given element to the number s of the corresponding sub slab containing the element. As a consequence, the left end-point a_e and right end-point b_e for a given element e are given by

$$\begin{aligned} a_e &= \texttt{sa}[\texttt{es}[e]], \\ b_e &= \texttt{sb}[\texttt{es}[e]]. \end{aligned} \tag{4.2}$$

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

TABLE 1. Data structures for efficient representation of a multi-adaptive time slab.

4.2. **Interpolating the solution at quadrature points.** Updating the values on an element according to (2.3) also requires knowledge of the value at the left end-point, which is given as the end-time value on the previous element in the time slab for the same component (or the end-time value from the previous time slab). This information is available in the array `ee`, which stores for each element the number of the previous element (or -1 if there is no previous element).

As discussed above in Section 3.3, the system of discrete equations on each time slab is solved by iterating over the elements in the time slab and updating the values on each element, either in a direct fixed-point iteration or a Newton's method. We must then for any given element e corresponding to some component $i = \texttt{ei}[e]$ evaluate the right-hand side f_i at each quadrature point t within the element. This requires the values of the solution U at t for all components contained in the sparsity pattern $\mathcal{S}_i$ for component i according to Algorithm 6. As a consequence of Algorithm 2 for the recursive construction of time slabs, elements for components that use large time steps are constructed before elements for components that use small time steps. Since all elements of the time slab are traversed in the same order during the iterative solution of the system of discrete equations, elements corresponding to large time steps have recently been visited and cover any element that corresponds to a smaller time step. The last visited element for each component is stored in an auxiliary array `elast` of size N. Thus, if $i' \in \mathcal{S}_i$ and component i' has recently been visited, then it is straight-forward to find the latest element $e' = \texttt{elast}[i']$ for component i' that covers the current element for component i and interpolate $U_{i'}$ at time t. It is also straight-forward to interpolate the values for any components that are present in the same element group as the current element.

However, when updating the values on an element e corresponding to some component $i = \texttt{ei}[e]$ depending on some other component $i' \in \mathcal{S}_i$ which uses smaller time steps, one must find for each quadrature point t on the element e the element e' for component i' containing t, which is non-trivial. The element e' can be found by searching through all elements for component i' in the time slab, but this quickly becomes inefficient. Instead, we store for each element e a list of dependencies to elements with smaller time steps in the two arrays `ed` and `de`. These two arrays store a sparse integer matrix of dependencies to elements with smaller time steps for all elements in the time slab. Thus, for any given element e, the number of dependencies to elements with smaller time steps is given by

$$\texttt{ed}[e+1] - \texttt{ed}[e], \tag{4.3}$$

and the elements with smaller time steps that need to be interpolated at the quadrature points for element e are given by

$$\{\texttt{de}[\texttt{ed}[e]], \texttt{de}[\texttt{ed}[e]+1], \ldots, \texttt{de}[\texttt{ed}[e+1]-1]\}. \tag{4.4}$$

5. Performance

The efficiency of multi-adaptive time-stepping compared to standard mono-adaptive time-stepping depends on the system being integrated, the tolerance and the efficiency of the implementation. For many systems, the potential speedup is large, but the actual speedup depends also on the overhead needed to handle the additional complications of a multi-adaptive implementation: the recursive construction of time slabs and the interpolation of values within a time slab.

To study the performance of multi-adaptive time-stepping, we consider a system with N components and time steps given by $\{k_{ij} = |I_{ij}| : I_{ij} \in \mathcal{T}_n\}$ on some time slab $\mathcal{T}_n$. We define the *multi-adaptive efficiency index* μ by

$$\mu = \frac{N/k_{\min}}{|\mathcal{T}_n|/k_{\max}} = \frac{k_{\max}}{k_{\min}} \frac{N}{|\mathcal{T}_n|}, \tag{5.1}$$

where $k_{\min} = \min_{I_{ij} \in \mathcal{T}_n} k_{ij}$, $k_{\max} = \max_{I_{ij} \in \mathcal{T}_n} k_{ij}$ and $|\mathcal{T}_n|$ is the number of local intervals in the time slab $\mathcal{T}_n$. Thus, to obtain the multi-adaptive efficiency index, we divide the number of local intervals per unit time for a mono-adaptive discretization with the actual number of local intervals per unit time for a multi-adaptive discretization. This is the potential speedup when compared to a mono-adaptive method that is forced to use the same time step $k_{\min}$ for all components. However, the actual speedup is always smaller than μ. This has two main reasons. The first is the overhead of the multi-adaptive implementation and the second is that the system of discrete equations on each time slab may sometimes be more expensive to solve than the corresponding mono-adaptive system(s).

Consider a model problem consisting of $N = N_K + N_k$ components, where N_K components vary on a slow time scale K and N_k components vary on a fast time scale k as in Figure 4. The potential speedup is given by the multi-adaptive efficiency index,

$$\mu = \frac{K}{k} \frac{N}{N_K + N_k K/k} = \frac{K}{k} \frac{N/K}{N_K/K + N_k/k} \sim \frac{K}{k} \gg 1, \tag{5.2}$$

if $N_K/K \gg N_k/k$ and $K \gg k$, that is the number of large elements dominates the number of small elements. Thus, the potential speedup can be very large for a system where a large part of the system varies on a large time scale and a small part of the system varying on a small time scale.

If, on the other hand, $K \sim k$ or $N_K \sim N_k$, then the multi-adaptive efficiency index may be of moderate size and as a consequence the actual speedup small if the overhead of the multi-adaptive implementation is significant. In the next section, we indicate the multi-adaptive efficiency index and compare this to the actual speedup for a number of benchmark problems.

6. Numerical examples and benchmark results

In this section, we present two benchmark problems to demonstrate the efficiency of multi-adaptive time-stepping. Both examples are time-dependent PDEs that we discretize in space using the cG(1) finite element method to obtain a system of ODEs, sometimes referred to as the method of lines approach. In each case, we lump and invert the mass matrix so as to obtain a system of the form (1.1).

In the first of the two benchmark problems, the individual time steps are chosen automatically based on an a posteriori error estimate as discussed above in Section 3.4 and

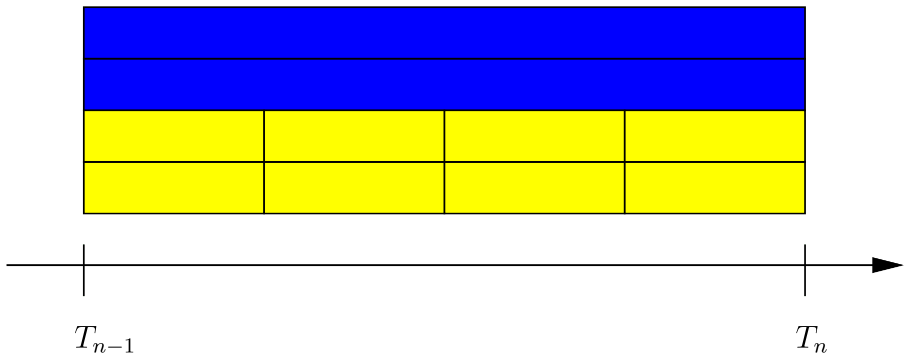

FIGURE 4. A time slab with $N_K = N_k = 2$ and multi-adaptive efficiency index $\mu = 16/10 = 1.6$

.

in the second, the time steps are fixed in time and vary in space according to the CFL condition so that $k \sim h$ locally. The results were obtained with DOLFIN version 0.6.2.[1]

6.1. **A nonlinear reaction-diffusion equation.** As a first example, we solve the following nonlinear reaction-diffusion equation, taken from [40]:

$$\begin{aligned} \dot{u} - \epsilon u'' &= \gamma u^2(1-u) \quad \text{in } \Omega \times (0,T], \\ \partial_n u &= 0 \quad \text{on } \partial\Omega \times (0,T], \\ u(\cdot,0) &= u_0 \quad \text{in } \Omega, \end{aligned} \tag{6.1}$$

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

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

$$u_0(x) = \frac{1}{1 + \exp(\lambda(x-1))}. \tag{6.2}$$

The resulting solution is a reaction front, sweeping across the domain from left to right, as demonstrated in Figure 5. The multi-adaptive time steps are automatically selected to be small in and around the reaction front and sweep the domain at the same velocity as the reaction front, as demonstrated in Figure 6.

To study the performance of the multi-adaptive solver, we compute the solution for a range of tolerances with $L = 5$ and compare the resulting error and CPU time with a standard mono-adaptive solver that uses equal (adaptive) time steps for all components. To make the comparison fair, we compare the multi-adaptive mcG(q) method with the mono-adaptive cG(q) method for $q = 1$. Both methods are implemented for general order q in the same programming language (C++) within a common framework (DOLFIN), but the mono-adaptive method takes full advantage of the fact that the time steps are

[1]DOLFIN 0.6.2 will be released shortly. In the meantime, the benchmark problems can be found in the publicly available development version of DOLFIN which can be downloaded from `http://www.fenics.org/dolfin/`.

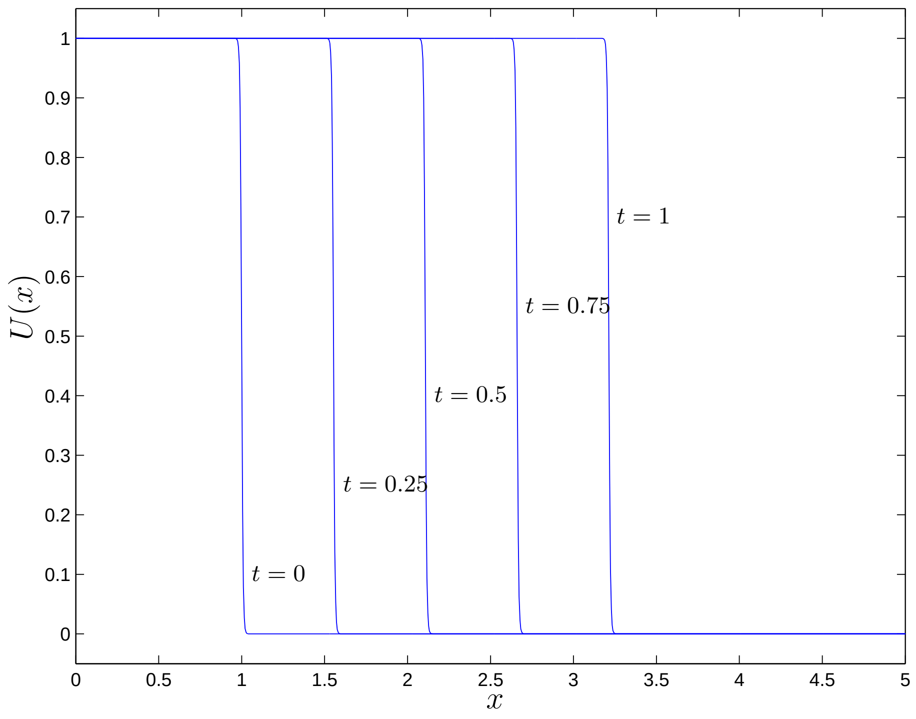

FIGURE 5. Propagation of the solution of the reaction–diffusion problem (6.1).

equal for all components. In particular, the mono-adaptive solver may use much simpler data structures (a plain C array) to store the solution on each time slab and there is no overhead for interpolation of the solution. This is a more difficult benchmark than only comparing the number degrees of freedom (local steps) as in [40] or comparing the CPU time against the same multi-adaptive solver when it is forced to use identical time steps for all components as in [33], since it also takes into account the overhead of the more complicated algorithms and data structures necessary for the implementation of multi-adaptive time-stepping.

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

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

In Figure 7, we plot the CPU time as function of the tolerance and number of components (size of domain) for the mcG(1) and cG(1) methods. We also summarize the results in Table 2 and Table 3. As expected, the speedup expressed as the multi-adaptive efficiency

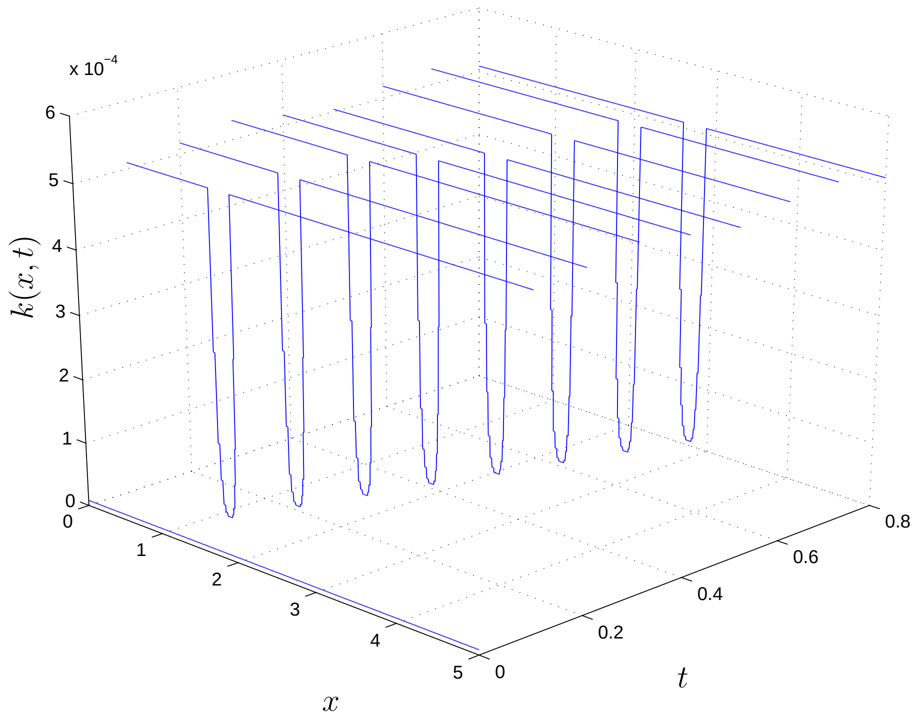

FIGURE 6. The multi-adaptive time steps as function of space at a sequence of points in time for the test problem (6.1).

index μ, that is, the ideal speedup if the cost per degree of freedom were the same for the multi- and mono-adaptive methods, is large in all test cases, around a factor 100. The speedup in terms of the total number of time slabs is also large. Note that in Table 2, the total number of time slabs M remains practically constant as the tolerance and the error are decreased. The decreased tolerance instead results in finer local resolution of the reaction front, which is evident from the increasing multi-adaptive efficiency index. At the same time, the mono-adaptive method needs to decrease the time step for all components and so the relative efficiency of the multi-adaptive method increases as the tolerance decreases. See also Figure 8 for a comparison of the multi-adaptive time steps at two different tolerances.

The situation is slightly different in Table 3, where the tolerance is kept constant but the size of the domain and number of components vary. Here, the number of time slabs remains practically constant for both methods, but the multi-adaptive efficiency index increases as the size of the domain increases, since the reaction front then becomes more and more localized relative to the size of the domain. As a result, the efficiency index of the multi-adaptive method increases as the size of the domain is increased.

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

There are mainly two reasons that make it difficult to attain full speedup. The first reason is that as the size of the time slab increases, the number of iterations n needed to solve the system of discrete equations increases. In Table 3, the number of iterations, including local iterations on individual elements as part of a global iteration on the time slab, is about a factor 1.5 larger for the multi-adaptive method. However, the main overhead lies in the more straightforward implementation of the mono-adaptive method compared to the more complicated data structures needed to store and interpolate the multi-adaptive solution. For constant time step and equal time step for all components, this overhead is roughly a factor 5 for the test problem, but the overhead increases to about a factor 100 when the time slab is locally refined. It thus remains important to further reduce the overhead of the implementation in order to increase the range of problems where the multi-adaptive methods give a positive speedup.

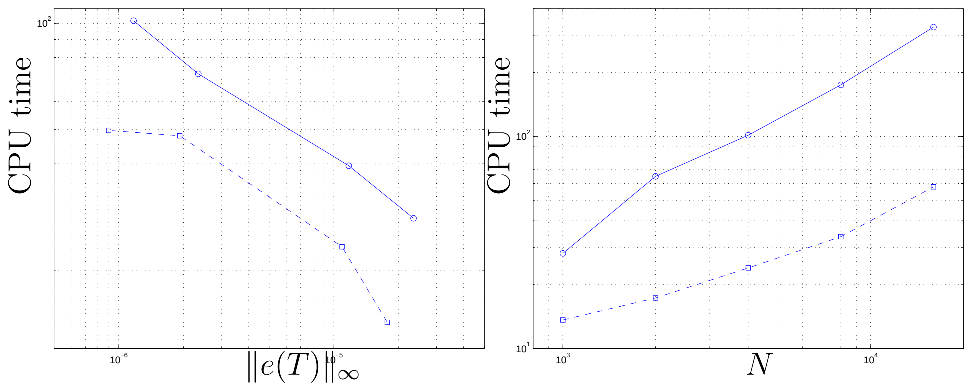

FIGURE 7. CPU time as function of the error (left) and number of components N (right) for mcG(1) (dashed line) and cG(1) (solid line).

6.2. **The wave equation.** Next, we consider the wave equation,

$$\begin{aligned} \ddot{u} - \Delta u &= 0 \quad \text{in } \Omega \times (0,T], \\ \partial_n u &= 0 \quad \text{on } \partial\Omega \times (0,T], \\ u(\cdot,0) &= u_0 \quad \text{in } \Omega, \end{aligned} \tag{6.3}$$

on a two-dimensional domain Ω consisting of two square sub domains of side length 0.5 separated by a thin wall with a narrow slit of size 0.0001×0.0001 at its center. The initial condition is chosen as a plane wave traversing the domain from right to left. In Figure 9, we plot the initial data together with the (fixed) multi-adaptive time steps. The resulting solution is shown in Figure 10.

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

Table 2. Benchmark results for mcG(1) (above) and cG(1) below for varying tolerance and fixed number of components $N = 1000$. The table shows the tolerance tol used for the computation, the error $\|e(T)\|_\infty$ in the maximum norm at the final time, the time used to compute the solution, the number of time slabs M (with the number of rejected time slabs in paranthesis), the average number of iterations n on the time slab system (with the number of local iterations on sub slabs in paranthesis), and the multi-adaptive efficiency index μ.

N	$\|e(T)\|_\infty$	time	M	n	μ
1000	$1.8 \cdot 10^{-5}$	13.6 s	1922 (5)	4.0 (1.5)	95.3
2000	$1.7 \cdot 10^{-5}$	17.3 s	1923 (5)	4.0 (1.2)	140.5
4000	$1.6 \cdot 10^{-5}$	24.0 s	1920 (6)	4.0 (1.0)	185.0
8000	$1.7 \cdot 10^{-5}$	33.7 s	1918 (5)	4.0 (1.0)	218.8
16000	$1.7 \cdot 10^{-5}$	57.9 s	1919 (5)	4.0 (1.0)	240.0
N	$\|e(T)\|_\infty$	time	M	n	μ
1000	$2.3 \cdot 10^{-5}$	28.1 s	117089 (1)	4.0	1.0
2000	$2.2 \cdot 10^{-5}$	64.8 s	117091 (1)	4.0	1.0
4000	$2.2 \cdot 10^{-5}$	101.3 s	117090 (1)	4.0	1.0
8000	$2.2 \cdot 10^{-5}$	175.1 s	117089 (1)	4.0	1.0
16000	$2.2 \cdot 10^{-5}$	327.7 s	117089 (1)	4.0	1.0

Table 3. Benchmark results for mcG(1) (above) and cG(1) below for fixed tolerance tol $= 1.0 \cdot 10^{-6}$ and varying number of components (and size of domain). (See Table 2 for an explanation of table legends.)

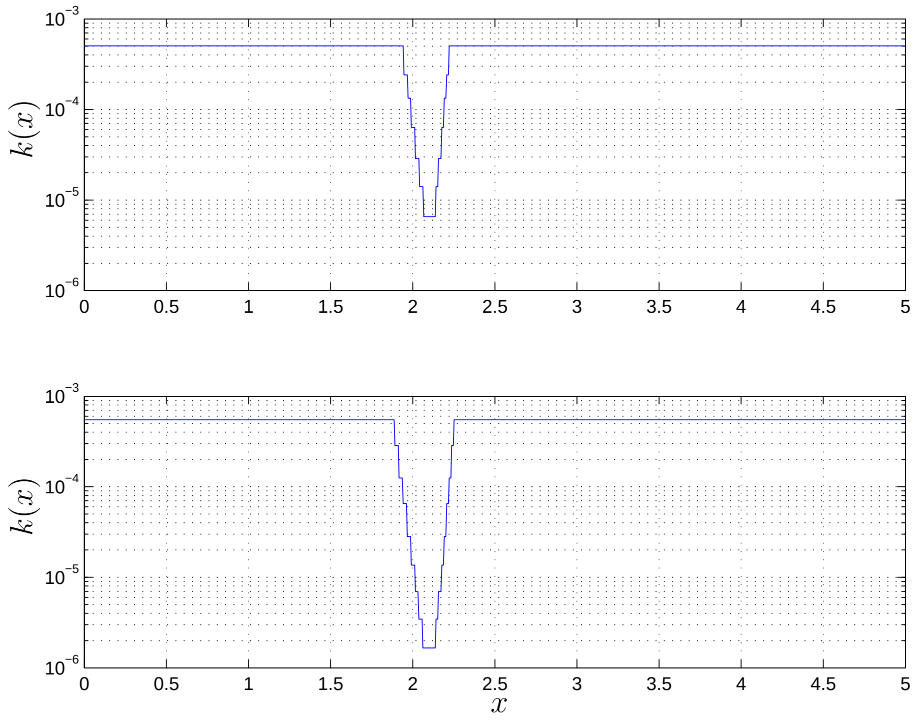

FIGURE 8. Multi-adaptive time steps at $t = 0.5$ for two different tolerances.

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

$$k \leq h_{\min} = \min_{x\in\Omega} h(x), \tag{6.4}$$

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

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

$$k(x) \leq h(x), \quad x \in \Omega, \tag{6.5}$$

and as a result, the number of local steps may decrease significantly (depending on the properties of the mesh) and as a result the total work may decrease (depending on the size of the overhead for the multi-adaptive method). In this case, with $k = 0.1h$, the speedup for the multi-adaptive mcG(1) method was a factor 4.2.

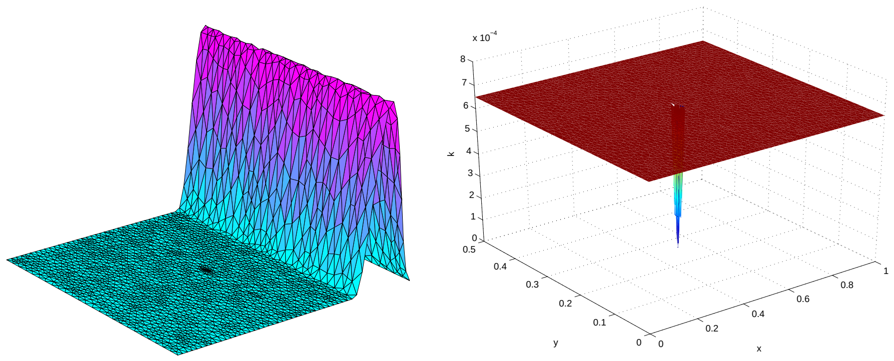

FIGURE 9. Initial data (left) and multi-adaptive time steps (right) for the solution of the wave equation.

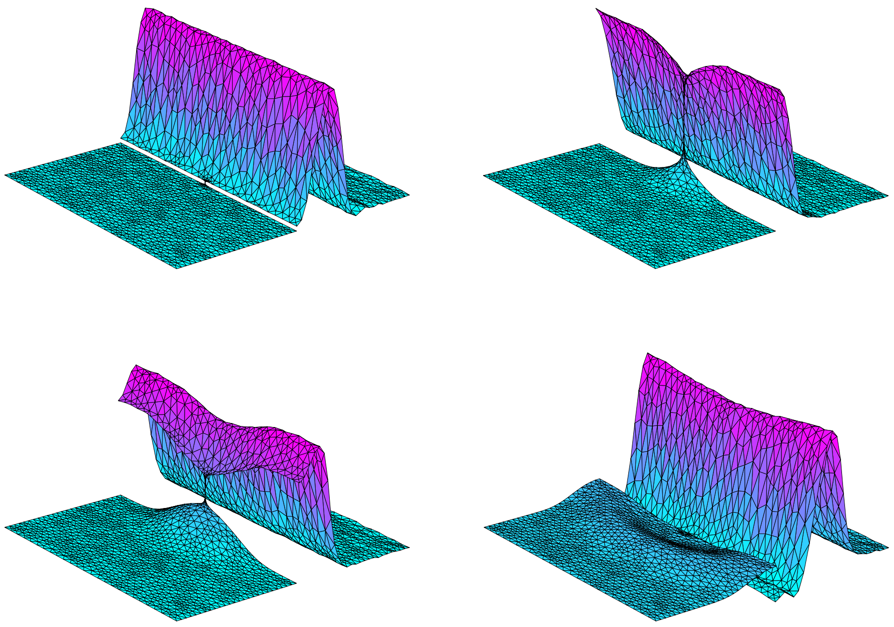

FIGURE 10. The solution of the wave equation at times $t = 0.25$, $t = 0.4$, $t = 0.45$ and $t = 0.6$.

7. CONCLUSIONS

We have discussed the basic algorithms for multi-adaptive time-stepping, including the recursive construction of time slabs and efficient interpolation of multi-adaptive solutions. The efficiency of the multi-adaptive methods was demonstrated for a pair of benchmark problems.

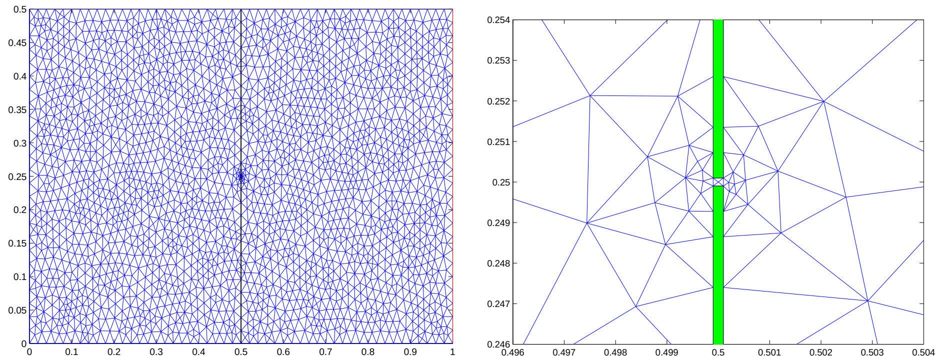

FIGURE 11. The mesh used for the solution of the wave equation on a domain intersected by a thin wall with a narrow slit (left) and details of the mesh close to the slit (right).

The multi-adaptive methods mcG(q) and mdG(q) are available as components of DOLFIN [22, 23], together with implementations of the standard mono-adaptive cG(q) and dG(q) methods. The ODE solvers of DOLFIN are currently being integrated with other components of the FEniCS project [21, 5], in particular the FEniCS Form Compiler (FFC) [36, 30, 31] in order to provide reliable, efficient and automatic integration of time dependent PDEs.

REFERENCES

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

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

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

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

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

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

[7] K. Eriksson and C. Johnson, *Adaptive finite element methods for parabolic problems I: A linear model problem*, SIAM J. Numer. Anal., 28, No. 1 (1991), pp. 43–77.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[35] ——, *Automation of Computational Mathematical Modeling*, PhD thesis, Chalmers University of Technology, Sweden, 2004.

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

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

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

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

[40] V. Savcenco, W. Hundsdorfer, and J. Verwer, *A multirate time stepping strategy for parabolic PDEs*, Tech. Rep. MAS–E0516, Centrum voor Wiskunde en Informatica (CWI), 2005.

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