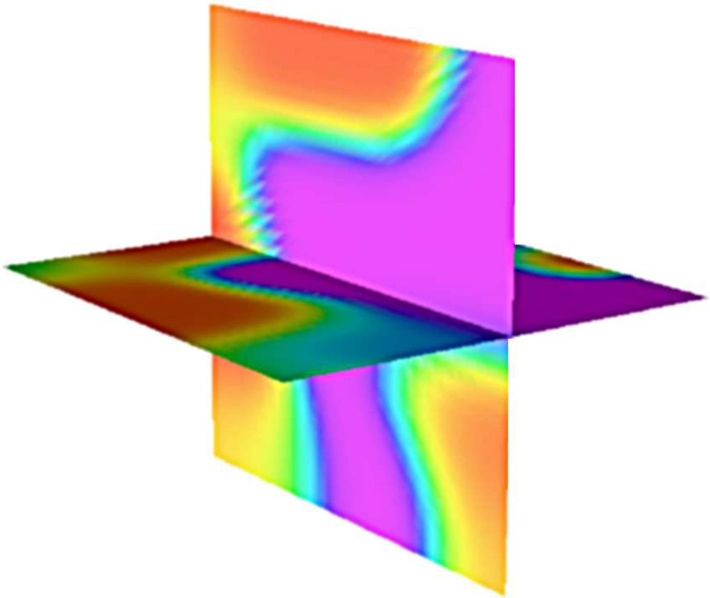
CHALMERS

FINITE ELEMENT CENTER



PREPRINT 2004–13

Algorithms for multi-adaptive time-stepping

Johan Jansson and Anders Logg

Chalmers Finite Element Center
CHALMERS UNIVERSITY OF TECHNOLOGY
Göteborg Sweden 2004

CHALMERS FINITE ELEMENT CENTER

Preprint 2004–13

Algorithms for multi-adaptive time-stepping

Johan Jansson and Anders Logg

CHALMERS

Chalmers Finite Element Center
Chalmers University of Technology
SE–412 96 Göteborg Sweden
Göteborg, April 2004

Algorithms for multi-adaptive time-stepping
Johan Jansson and Anders Logg
NO 2004–13
ISSN 1404–4382

Chalmers Finite Element Center
Chalmers University of Technology
SE–412 96 Göteborg
Sweden
Telephone: +46 (0)31 772 1000
Fax: +46 (0)31 772 3595
www.phi.chalmers.se

Printed in Sweden
Chalmers University of Technology
Göteborg, Sweden 2004

ALGORITHMS 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 time steps to be used for different components or in different regions of space. We present algorithms for multi-adaptive time-stepping, including the recursive construction of time slabs, regulation of the individual time steps, adaptive fixed point iteration on time slabs, and the automatic generation of dual problems. An example is given for the solution of a nonlinear partial differential equation in three dimensions.

1. Introduction

We have earlier in a sequence of papers [14, 15, 16, 17, 13] 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.

In the current paper, we discuss important aspects of the implementation of multi-adaptive Galerkin methods. Some of these aspects are discussed in [15] and [13], but with technical details left out. We now provide these details.

1.1. Implementation.

The algorithms presented in this paper are implemented in the multi-adaptive ODE-solver of **DOLFIN** [11, 12], which is the C++ implementation of the new open-source software project **FEniCS** [3] for the automation of Computational Mathematical Modeling (CMM). The multi-adaptive solver in **DOLFIN** is based on the original implementation *Tanganyika*, presented in [15], but has been completely rewritten for **DOLFIN**.

Date: April 14, 2004.

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

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

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

1.2. **Obtaining the software.** **DOLFIN** is licensed under the GNU General Public License [8], which means that anyone is free to use or modify the software, provided these rights are preserved.

The source code of **DOLFIN**, including numerous example programs, is available at the **DOLFIN** web page, `http://www.phi.chalmers.se/dolfin/`, and each new release is announced on freshmeat.net. Alternatively, the source code can be obtained through anonymous CVS as explained on the web page. Comments and contributions are welcome.

1.3. **Notation.** For a detailed description of the multi-adaptive Galerkin methods, we refer the reader to [14, 15, 16, 17, 13]. In particular, we refer to [14] or [16] for the definition of the methods.

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 subintervals. Subinterval j for component i is denoted by $I_{ij} = (t_{i,j-1}, t_{ij}]$, and the length of the subinterval is given by the local *time step* $k_{ij} = t_{ij} - t_{i,j-1}$. This is illustrated in Figure 1. On each subinterval I_{ij}, $U_i|_{I_{ij}}$ is a polynomial of degree q_{ij} and we refer to $(I_{ij}, U_i|_{I_{ij}})$ as an *element*.

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 present the user interface of the multi-adaptive solver in Section 2, before we discuss the algorithms of multi-adaptive time-stepping in Section 3. In Section 4, we present numerical results for the bistable equation as an example of multi-adaptive time-stepping for a partial differential equation.

2. User interface

Potential usage of the multi-adaptive solver ranges from a student or teacher wanting to solve a fixed ODE in an educational setting, to a PDE package using it as an internal solver module. The user interface of the multi-adaptive solver is specified in terms of an ODE base class consisting of a right hand side f, a time interval $[0, T]$, and initial value u_0, as shown in Figure 2.

To solve an ODE, the user implements a subclass which inherits from the ODE base class. As an example, we present in Figure 2 and Table 1 the implementation of the

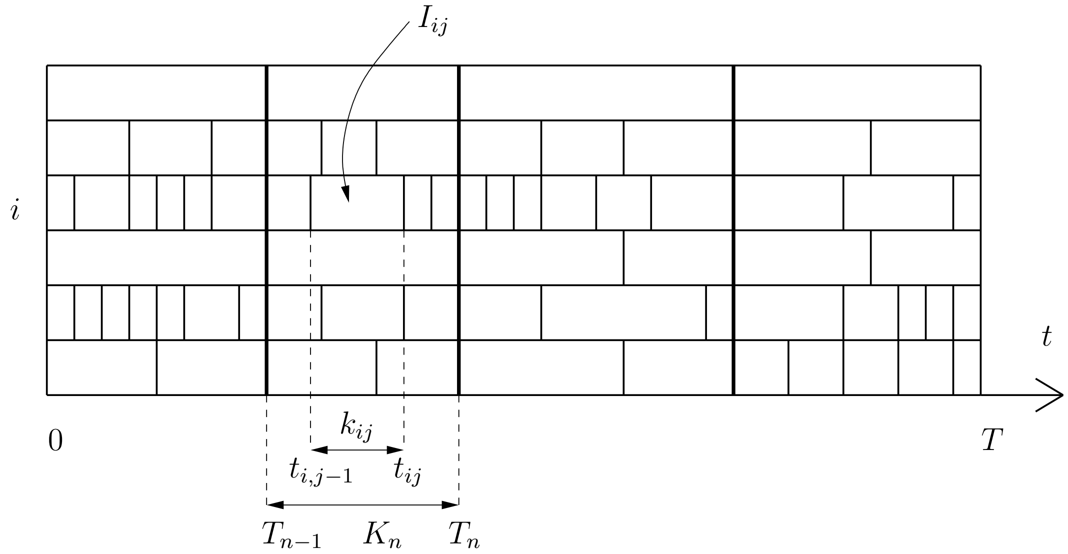


FIGURE 1. Individual partitions of the interval $(0, T]$ for different components. Elements between common synchronized time levels are organized in time slabs. In this example, we have $N = 6$ and $M = 4$.

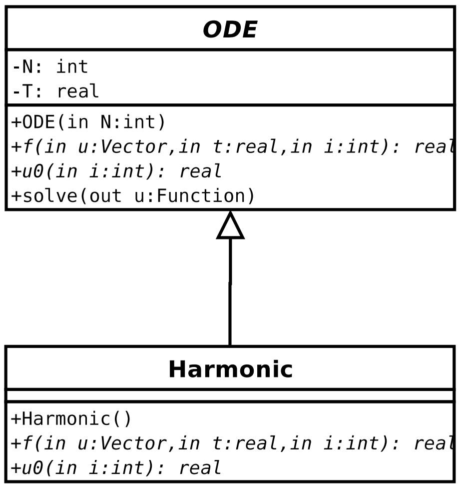


FIGURE 2. UML class diagram showing the base class interface of the multi-adaptive ODE-solver together with a subclass representing the ODE (2.1).

harmonic oscillator

$$\begin{aligned} \dot{u}_1 &= u_2, \\ \dot{u}_2 &= -u_1, \end{aligned} \tag{2.1}$$

on $[0, 10]$ with initial value $u(0) = (0, 1)$.

```
Harmonic::Harmonic : ODE(2)
{
  T = 10.0;
}

real Harmonic::u0(int i)
{
  if (i == 0)
    return 0;
  if (i == 1)
    return 1;
}

real Harmonic::f(Vector u, real t, int i)
{
  if (i == 0)
    return u(1);
  if (i == 1)
    return -u(0);
}
```

TABLE 1. Sketch of the C++ implementation of the harmonic oscillator (2.1). Note that C++ indexing starts at 0.

3. Multi-adaptive time-stepping

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 using C++ in **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 (Algorithm 1) is based on the algorithm CreateTimeSlab (Algorithm 3) and the algorithms for adaptive fixed point iteration on time slabs discussed below in Section 3.3. Starting at $t = 0$, the algorithm creates a sequence of time slabs until the given end time T is reached. The end time T is given as an argument to CreateTimeSlab, which creates a time slab covering an interval $[T_{n-1}, T_n]$ such that $T_n \leq T$. CreateTimeSlab returns the end time T_n of the created time slab and the integration continues until $T_n = T$. For each time slab, adaptive fixed point iteration is performed on the time slab until the discrete equations given by the mcG(q) or mdG(q) method have converged.

In **DOLFIN**, Algorithm 1 is implemented by the class `TimeStepper`, which can be used in two different ways (see Table 2). In the standard case, the function `TimeStepper::solve()` is called to integrate the given ODE on the interval $[0, T]$. The class `TimeStepper` also provides the alternative interface `TimeStepper::step()`, that can be used to integrate

Algorithm 1 U = Integrate(ODE)

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

```
class TimeStepper
{
  TimeStepper(ODE ode, Function u);

  static void solve(ODE ode, Function u);
  real step();
}
```

TABLE 2. Sketch of the C++ interface of the multi-adaptive time-stepper, Algorithm 1.

the solution one time slab at a time. This is useful in interactive applications, where the right-hand side f of (1.1) needs to be modified in each step of the integration.

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 (Algorithm 2). This algorithm first estimates the individual *stability factors* $\{S_i(T)\}_{i=1}^N$, which together with the local residuals determine the multi-adaptive time steps. (See [4, 5, 2] or [14] for a discussion on duality-based a posteriori error estimation.) The preliminary estimates for the stability factors can be based on previous experience, i.e., if we have solved a similar problem before, but usually we take $S_i(T) = 1$ for $i = 1, \ldots, N$.

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, as discussed below in Section 3.7, 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. This is illustrated in Figure 3.

To create a time slab, we first compute the desired time steps for all components as discussed below in Section 3.5. A threshold θK is then computed based on the maximum time step K and a fixed parameter θ controlling the density of the time slab. The

Algorithm 2 U = Solve(ODE, TOL)

estimate stability factors
repeat
  U = Integrate(ODE)
  create dual problem ODE*
  Φ = Integrate(ODE*)
  compute stability factors
  compute error estimate E
until $E \leq$ TOL

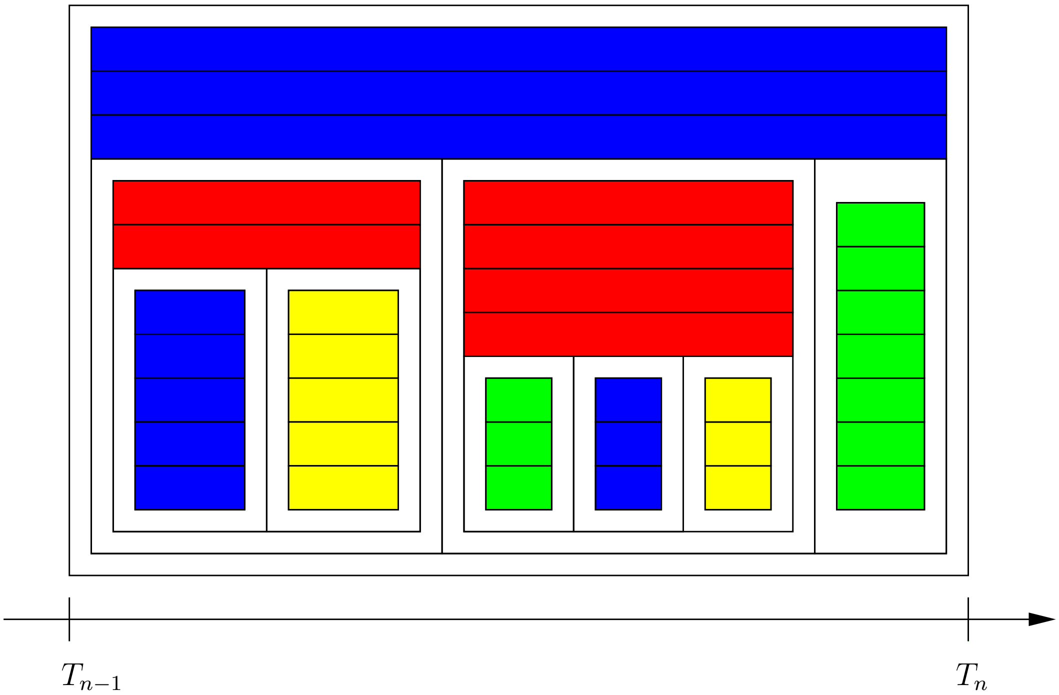


FIGURE 3. The recursive organization of the time slab. Each time slab contains an element group and a list of recursively nested time slabs. The root time slab in the figure contains one element group of three elements and three time slabs. The first of these sub slabs contains an element group of two elements and two nested time slabs, and so on. The root time slab recursively contains a total of nine element groups and 35 elements.

components are then partitioned into two sets based on the threshold, see Figure 4. 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. For a more detailed discussion of the construction, see [13].

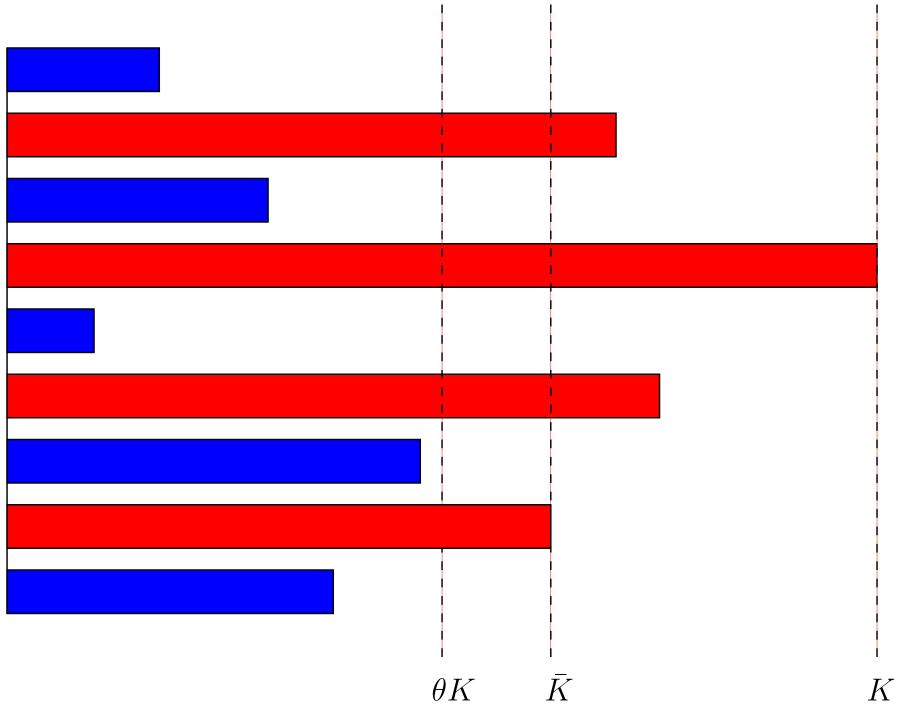


FIGURE 4. The partition of components into groups of small and large time steps for $\theta = 1/2$.

We organize the recursive construction of time slabs as described by Algorithms 3, 4, 5, and 6. The recursive construction simplifies the implementation; each recursively nested time slab can be considered as a sub system of the ODE. Note that the group of recursively nested time slabs for components in group I_0 is created before the element group containing elements for components in group I_1. The tree of time slabs is thus created recursively *depth-first*, which means in particular that the element for the component with the smallest time step is created first.

Algorithm 4 for the partition of components can be implemented efficiently using the function `std::partition()`, which is part of the Standard C++ Library.

3.3. **Adaptive fixed point iteration on time slabs.** As discussed in [13], the discrete equations given by the mcG(q) or mdG(q) method on each time slab are solved using adaptive fixed point iteration. For the fixed point iteration, each time slab is viewed as a discrete system of equations the form

$$F(x) = 0 \tag{3.1}$$

for the degrees of freedom x of the solution U on the time slab. This system is partitioned into coupled sub systems which each take the form (3.1), one for each element group. Similarly, each element group is naturally partitioned into sub systems, one for each element within the element group.

Algorithm 3 $\{$time slab, $T_n\}$ = CreateTimeSlab(components, T_{n-1}, T)

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

Algorithm 4 $\{I_0, I_1, K\}$ = Partition(components)

```
I_0 ← ∅
I_1 ← ∅
K ← maximum time step within components
for each component do
  k ← time step of component
  if k < θK then
    I_0 ← I_0 ∪ {component}
  else
    I_1 ← I_1 ∪ {component}
  end if
end for
K̄ ← minimum time step within I_1
K ← K̄
```

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

```
time slabs ← ∅
t ← T_{n-1}
while t < T do
  {time slab, t} ← CreateTimeSlab(components, t, T_n)
  time slabs ← time slabs ∪ time slab
end while
```

Algorithm 6 elements = CreateElements(components, T_{n-1}, T_n)

```
elements ← ∅
for each component do
  create element for component on [T_{n-1}, T_n]
  elements ← elements ∪ element
end for
```

Iterate(time slab)
while *condition 3* **do**
  Update(time slab)
end while
	Update(time slab)
	for each element group **do**
	  Iterate(element group)
	end for

Iterate(element group)
while *condition 2* **do**
  Update(element group)
end while
	Update(element group)
	for each element **do**
	  Iterate(element)
	end for

Iterate(element)
while *condition 1* **do**
  Update(element)
end while
	Update(element)
	for each degree of freedom **do**
	  Update(degree of freedom)
	end for

TABLE 3. Nested fixed point iteration on the time slab.

The general algorithm for nested fixed point iteration on the sub systems of a time slab is based on the principle that each iteration on a given system consists of fixed point iteration on each sub system, as outlined in Table 3. By modifying the condition (*1*, *2*, or *3*) for fixed point iteration on each level of iteration, different versions of the overall iterative algorithm are obtained. In [13], four different strategies for the adaptive fixed point iteration are discussed: *non-stiff iteration*, *adaptive level 1 iteration*, *adaptive level 2 iteration*, and *adaptive level 3 iteration*. By monitoring the convergence at the different levels of iteration, the appropriate version of the adaptive fixed point iteration is chosen, depending on the stiffness of the problem.

Table 3 shows a simplified version of the fixed point iteration. The full algorithm also needs to monitor the convergence rate, stabilize the iterations, and (possibly) change strategy, as shown in Algorithm 7 (Iterate) and Algorithm 8 (Update). Both algorithms return the increment d for the degrees of freedom of the current (sub) system. In the case of Algorithm 7, the increment is computed as the maximum increment over the iterations for the system, and in Algorithm 8, the increment is computed as the l_2-norm of the increments for the set of sub systems.

Since we sometimes need to change the strategy for the fixed point iteration depending on the stiffness of the problem, the fixed point iteration is naturally implemented as a *state machine*, where each state has different versions of the algorithms Converged, Diverged, Stabilize, and Update.

In **DOLFIN**, the state machine is implemented as shown in Figure 5, following the design pattern for a state machine suggested in [9]. The class `FixedPointIteration` implements Algorithm 7 and the class `Iteration` serves as a base class (interface) for the subclasses `NonStiffIteration`, `AdaptiveIterationLevel1`, `AdaptiveIterationLevel2`,

Algorithm 7 $d =$ Iterate(system)

$d \leftarrow 0$
loop
  for $n = 1, \dots, n_{\max}$ **do**
    if Converged(system) **then**
      return
    end if
    if Diverged(system) **then**
      ChangeState()
      break
    end if
    Stabilize(system)
    $d \leftarrow \max(d,$ Update(system)$)$
  end for
end loop

Algorithm 8 $d =$ Update(system)

$d \leftarrow 0$
for each sub system **do**
  $d_i \leftarrow$ Iterate(sub system)
  $d \leftarrow d + d_i^2$
end for
$d \leftarrow \sqrt{d}$

and `AdaptiveIterationLevel3`. Each of these subclasses implement the interface specified by the base class `Iteration`, in particular the functions `Iteration::converged()`, `Iteration::diverged()`, `Iteration::stabilize()`, and `Iteration::update()` for each level of iteration (element level, element group level, and time slab level). To change the state, the object pointed to by the pointer `state` is deleted and a new object is allocated to the pointer, with the type of the new object determined by the new state. This simplifies the implementation of the state machine and makes it possible to separate the implementation of the different states.

3.4. **Cumulative power iteration.** In each iteration of Algorithm 7, the amount of damping for the fixed point iteration is determined by the algorithm Stabilize. If the convergence rate of the iterations is not satisfactory, then the appropriate damping factor α and the number of stabilizing iterations m are determined by *cumulative power iteration*, as discussed in [13]; the divergence rate ρ is first determined by Algorithm 9, and then α and m are determined according to

$$\alpha = \frac{1/\sqrt{2}}{1+\rho} \tag{3.2}$$

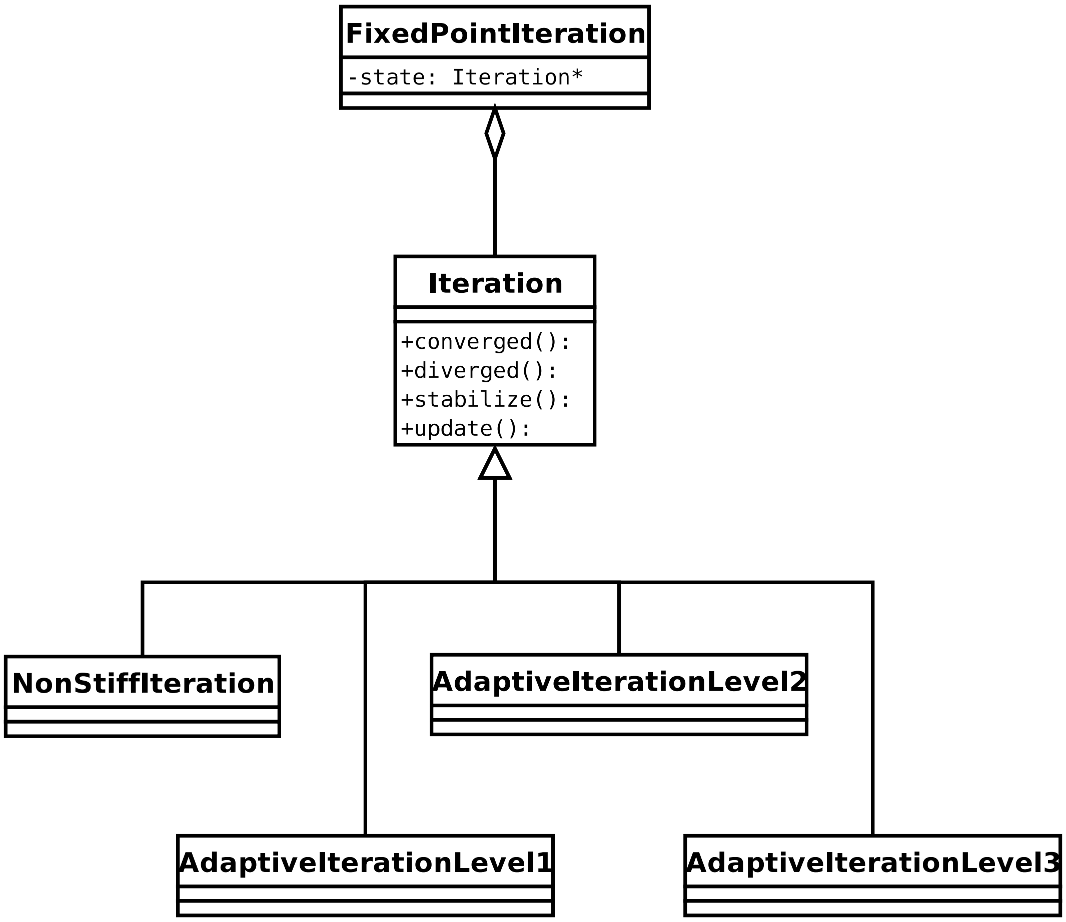


FIGURE 5. UML class diagram showing the implementation of the state machine for adaptive fixed point iteration in **DOLFIN**.

and

$$m = \log \rho. \tag{3.3}$$

The iteration continues until the divergence rate ρ has converged to within some tolerance tol, typically $\mathrm{tol} = 10\%$.

3.5. **Controlling the individual time steps.** The individual and adaptive time steps k_{ij} are determined during the recursive construction of time slabs based on an a posteriori error estimate for the global error $\|e(T)\|_{l_2}$ at final time, as described in [14, 15]. The a posteriori error estimate is expressed in terms of the individual stability factors $\{S_i(T)\}_{i=1}^N$, the local time steps $\{k_{ij}\}_{j=1,i=1}^{M_i,N}$, and the local residuals $\{r_{ij}\}_{j=1,i=1}^{M_i,N}$. The a posteriori error estimate takes the form

$$\|e(T)\|_{l_2} \leq \sum_{i=1}^{N} S_i(T) \max_{j=1,\ldots,M_i} k_{ij}^{p_{ij}} r_{ij}, \tag{3.4}$$

Algorithm 9 $\rho = \text{ComputeDivergence(system)}$

$d_1 \leftarrow \text{Update(system)}$
$d_2 \leftarrow \text{Update(system)}$
$\rho_2 \leftarrow d_2/d_1$
$n \leftarrow 2$
repeat
  $d_1 \leftarrow d_2$
  $d_2 \leftarrow \text{Update(system)}$
  $\rho_1 \leftarrow \rho_2$
  $\rho_2 \leftarrow \rho_2^{(n-1)/n} \cdot (d_2/d_1)^{1/n}$
  $n \leftarrow n+1$
until $|\rho_2 - \rho_1| < \text{tol} \cdot \rho_1$
$\rho \leftarrow \rho_2$

with $p_{ij} = q_{ij}$ for the mcG(q) method and $p_{ij} = q_{ij} + 1$ for the mdG(q) method.

Basing the time step for interval I_{ij} on the residual of the previous interval $I_{i,j-1}$, since the residual of interval I_{ij} is unknown until the solution on that interval has been computed, we should thus choose the local time step k_{ij} according to

$$k_{ij} = \left(\frac{\text{TOL}}{N\,S_i(T) r_{i,j-1}}\right)^{1/p_{ij}}, \quad j = 2,\ldots,M_i, \quad i = 1,\ldots,N, \tag{3.5}$$

where TOL is a given tolerance.

However, the time steps can not be based directly on (3.5), since that leads to oscillations in the time steps. If $r_{i,j-1}$ 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 $k_{\text{new}} = k_{ij}$ according to Algorithm 10, which determines the new time step as the harmonic mean value of the previous time step and the time step determined by (3.5).

Alternatively, the time steps can be determined using control theory, as suggested in [10, 18]. Typically, a standard PID controller is used to determine the time steps with the goal of satisfying $k_{ij}^{p_{ij}} r_{ij} = \text{TOL}/(N\,S_i(T))$ on $[0,T]$ for $i = 1,\ldots,N$. However, since multi-adaptive time-stepping requires an individual controller for each component, the current implementation of **DOLFIN** determines the time steps according to Algorithm 10.

Algorithm 10 $k = \text{Controller}(k_{\text{new}},\, k_{\text{old}},\, k_{\max})$

$k \leftarrow 2k_{\text{old}}k_{\text{new}}/(k_{\text{old}} + k_{\text{new}})$
$k \leftarrow \min(k, k_{\max})$

The initial time steps $k_{11} = \cdots = k_{N1} = K_1$ 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.

3.6. **Implementation of general elements.** As described in [15], the system of equations to be solved for the degrees of freedom $\{\xi_{ijm}\}$ on each element takes the form

$$\xi_{ijm} = \xi_{ij0} + \int_{I_{ij}} w_m^{[q_{ij}]}(\tau_{ij}(t)) f_i(U(t), t)\, dt, \quad m = 1, \ldots, q_{ij}, \tag{3.6}$$

for the mcG(q) method, with $\tau_{ij}(t) = (t - t_{i,j-1})/(t_{ij} - t_{i,j-1})$ and where $\{w_m^{[q_{ij}]}\}_{m=1}^{q_{ij}} \subset \mathcal{P}^{[q_{ij}-1]}([0,1])$ are polynomial weight functions. For the mdG(q) method, the system of equations on each element has a similar form, with $m = 0, \ldots, q_{ij}$.

The integral in (3.6) is computed using quadrature, and thus the weight functions $\{w_m^{[q_{ij}]}\}_{m=1}^{q_{ij}}$ need to be evaluated at a set of quadrature points $\{s_n\} \subset [0,1]$. In **DOLFIN**, these values are computed and tabulated each time a new type of element is created. If the same method is used for all components throughout the computation, then this computation is carried out only once.

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

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

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

After the quadrature points $\{s_n\}_{n=0}^{q_{ij}}$ have been determined, the polynomial weight functions $\{w_m^{[q_{ij}]}\}_{m=1}^{q_{ij}}$ are computed as described in [14] (again by solving a linear system) and then evaluated at the quadrature points. Multiplying these values with the quadrature weights, we rewrite (3.6) in the form

$$\xi_{ijm} = \xi_{ij0} + k_{ij} \sum_{n=0}^{q_{ij}} w_{mn}^{[q_{ij}]} f_i(U(t_{i,j-1} + s_n k_{ij}), t_{i,j-1} + s_n k_{ij}), \quad m = 1, \ldots, q_{ij}. \tag{3.7}$$

General order mcG(q) and mdG(q) have been implemented in **DOLFIN**. The two methods are implemented by the two classes `cGqElement` and `dGqElement`, implementing the interface specified by the common base class `Element`. Both classes take the order q as an argument to its constructor and implement the appropriate version of (3.7).

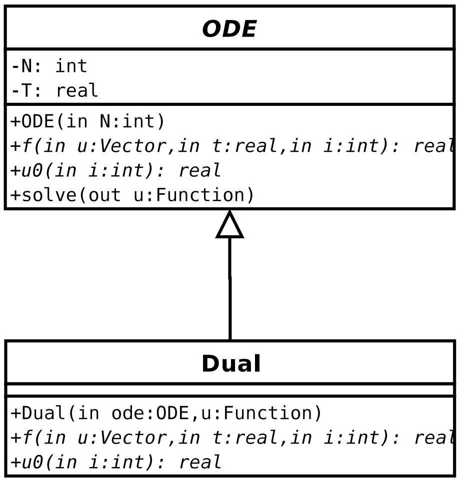


FIGURE 6. The dual problem is implemented as a subclass of the common base class for ODEs of the form (1.1).

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

$$\begin{aligned} -\dot{\phi}(t) &= J(U,t)^\top \phi(t), \quad t \in [0,T), \\ \phi(T) &= \psi, \end{aligned} \tag{3.8}$$

where $J(U,t)$ denotes the Jacobian of the right-hand side f of (1.1) at time t. 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 (3.8) as a forward problem. With $w(t) = \phi(T-t)$, we have $\dot{w} = -\dot{\phi}(T-t) = J^\top(U,T-t)w(t)$, and so (3.8) can be written as a forward problem for w in the form

$$\begin{aligned} \dot{w}(t) &= f^*(w(t),t) \equiv J(U,T-t)^\top w(t), \quad t \in (0,T], \\ w(0) &= \psi. \end{aligned} \tag{3.9}$$

In **DOLFIN**, the initial value problem for w is implemented as a subclass of the ODE base class, as shown in Figure 6, which makes it possible to solve the dual problem using the same algorithm as the primal problem.

The constructor of the dual problem takes as arguments the primal problem (1.1) and the computed solution U of the primal problem. The right-hand side f^* of the dual problem is automatically generated by numerical differentiation of the right-hand side of the primal problem.

3.8. **Interpolation of the solution.** To update the degrees of freedom for an element according to (3.7), 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_j 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, i.e., the set of components on which f_i depend,

$$\mathcal{S}_i = \{j \in \{1,\ldots,N\} : \partial f_i/\partial u_j \neq 0\}. \tag{3.10}$$

Thus, to evaluate f_i at a given time t, only the components U_j for $j \in S_i$ need to be evaluated at t, see Algorithm 11. This is of particular importance for problems of sparse structure and makes it possible to use the multi-adaptive solver for the integration of time-dependent PDEs, see Section 4. The sparsity pattern $\mathcal{S}_i$ is automatically detected by the solver. Alternatively, the sparsity pattern can be specified by a (sparse) matrix.

Algorithm 11 $y =$ EvaluateRightHandSide(i,t)

```
for j ∈ S_i do
  x(j) ← U_j(t)
end for
y ← f_i(x, t)
```

The key part of Algorithm 11 is the evaluation of a given component U_i at a given time t. To evaluate $U_i(t)$, the solver first needs to find the element $(I_{ij}, U_i|_{I_{ij}})$ satisfying $t \in I_{ij}$. The local polynomial $U_i|_{I_{ij}}$ is then evaluated (interpolated) at the given time t. During the construction of a time slab, an element such that $t \in I_{ij}$ might not exists, in which case the last element of component U_i is used and extrapolated to the given time t.

To find the element $(I_{ij}, U_i|_{I_{ij}})$ such that $t \in I_{ij}$, the function `std::upper_bound()` is used. This function is part of the Standard C++ Library and uses binary search to find the appropriate element from the ordered sequence of elements for component U_i, which means that the complexity of finding the element is logarithmic. In addition, the speed of the evaluation is increased by caching the latest used element and each time checking this element before the binary search is performed.

3.9. **Storing the solution.** Since the computed solution U of the primal problem (1.1) is needed for the computation of the discrete solution Φ of the dual problem (3.8), the solution needs to be stored to allow U to be evaluated at any given $t \in [0,T]$.

The solution is stored on disk in a temporary file created by the function `tmpfile()`, which is part of the C standard I/O library. The solution is written in blocks to the file in binary format. During the computation, a cache of blocks is kept in main memory to allow efficient evaluation of the solution. The number of blocks kept in memory depends on the amount of memory available. For a sufficiently large problem, only one block will be kept in memory. Each time a value is requested which is not in one of the available blocks, one of these blocks is dropped and the appropriate block is fetched from file.

4. Solving the bistable equation

As an example of multi-adaptive time-stepping, we solve the bistable equation on the unit cube,

$$\begin{aligned} \dot{u} - \epsilon \Delta u &= u(1-u^2) \quad \text{in } \Omega \times (0,T], \\ \partial_n u &= 0 \quad \text{on } \partial\Omega \times (0,T], \\ u(\cdot,0) &= u_0 \quad \text{in } \Omega, \end{aligned} \tag{4.1}$$

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

The bistable equation has been studied extensively before [7, 6] and has interesting stability properties. In particular, it has two stable steady-state solutions, $u = 1$ and $u = -1$, and one unstable steady-state solution, $u = 0$. From (4.1), it is clear that the solution increases in regions where it is positive and decreases in regions where it is negative. Because of the diffusion, neighboring regions will compete until finally the solution has reached one of the two stable steady states. Since this action is local on the interface between positive and negative regions, the bistable equation is an ideal example for multi-adaptive time-stepping.

To solve the bistable equation (4.1) using multi-adaptive time-stepping, we discretize in space using the standard cG(1) method [5]. Lumping the mass matrix, we obtain an ODE initial value problem of the form (1.1), which we solve using the multi-adaptive mcG(1) method. We refer to the overall method thus obtained as the cG(1)mcG(1) method.

The solution was computed on a uniformly refined tetrahedral mesh with mesh size $h = 1/64$. This mesh consists of $1,572,864$ tetrahedrons and has $N = 274,625$ nodes. In Figure 7, we plot the initial value used for the computation, and in Figure 8 the solution at final time $T = 100$. We also plot the solution and the multi-adaptive time steps at time $t = 10$ in Figure 9 and Figure 10, and note that the time steps are small in regions where there is strong competition between the two stable steady-state solutions, in particular in regions with where the curvature of the interface is small.

The computation was carried out using **DOLFIN** version 0.4.9, which includes the bistable equation as one of numerous multi-adaptive test problems. The visualization of the solution was made using OpenDX [1] version 4.3.2. Animations of the solution and the multi-adaptive time steps are available in the gallery on the **DOLFIN** web page [11].

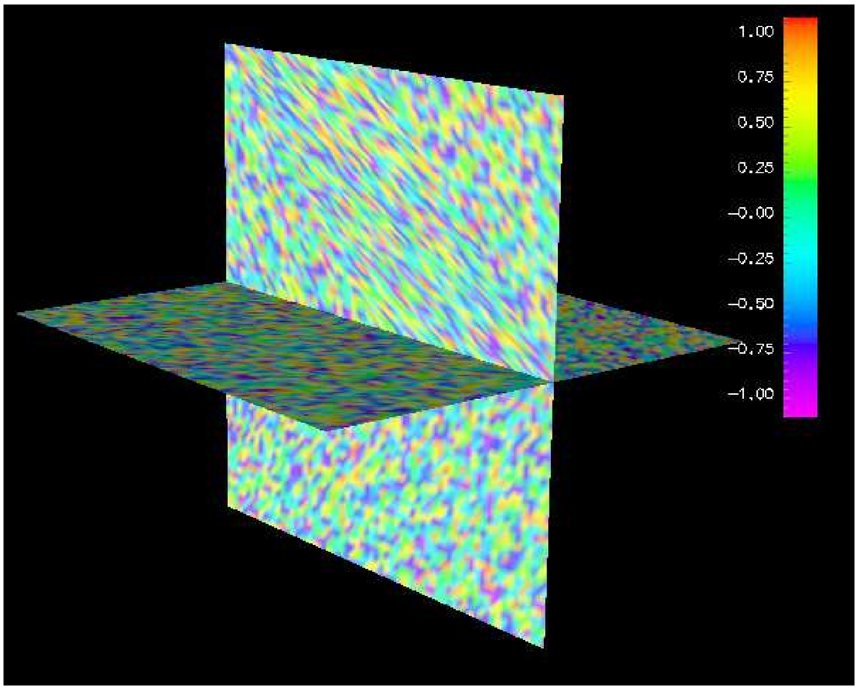


FIGURE 7. Initial data for the solution of the bistable equation (4.1).

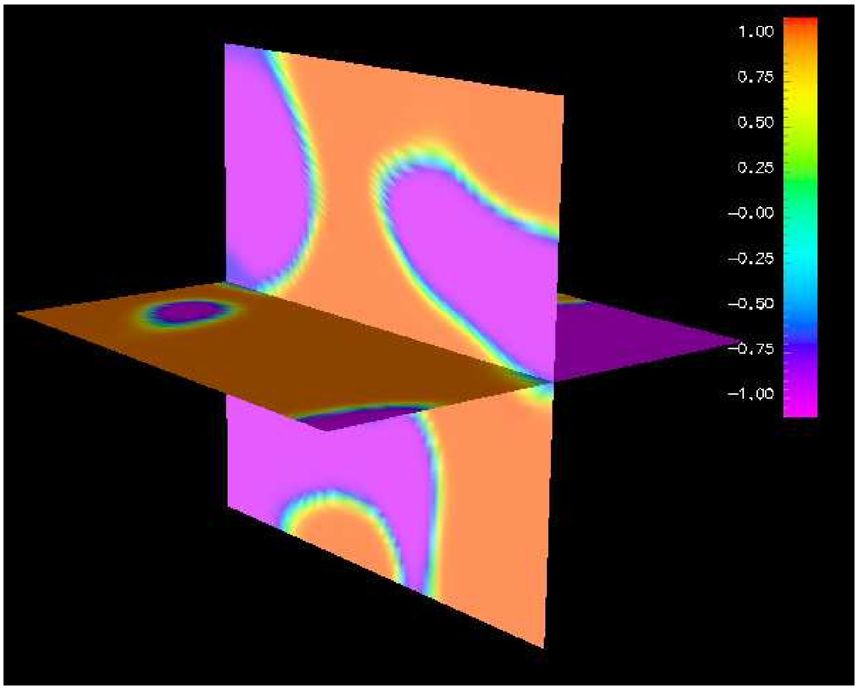


FIGURE 8. Solution of the bistable equation (4.1) at final time $T = 100$.

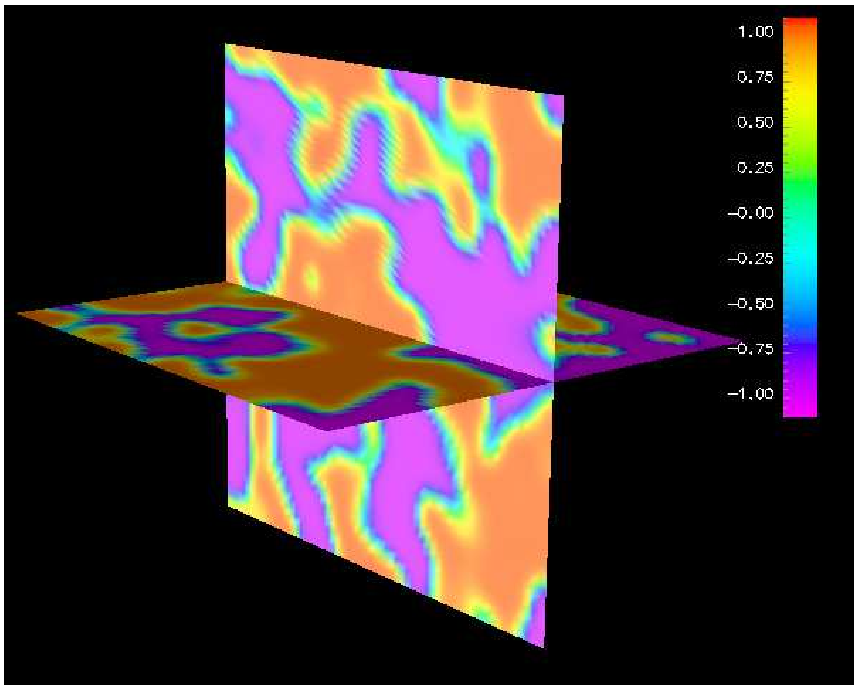


FIGURE 9. Solution of the bistable equation (4.1) at time $t = 10$.

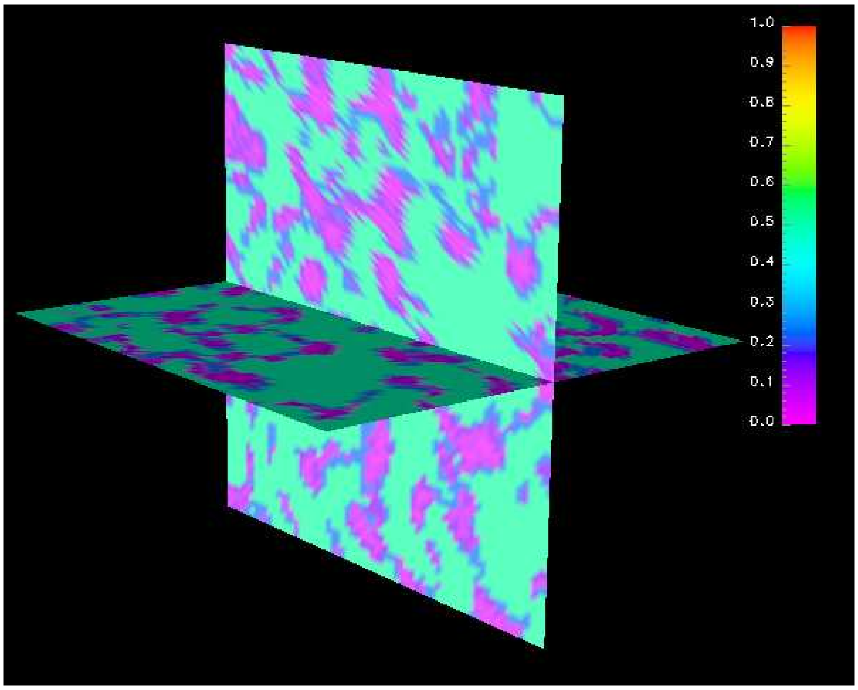


FIGURE 10. Multi-adaptive time steps at time $t = 10$ for the solution of the bistable equation (4.1).

References

[1] *OpenDX*, http://www.opendx.org/.

[2] R. Becker and R. Rannacher, *An optimal control approach to a posteriori error estimation in finite element methods*, Acta Numerica, 10 (2001).

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

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

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

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

[7] 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.

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

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley, 1995.

[10] 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.

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

[12] 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.

[13] J. Jansson and A. Logg, *Multi-adaptive Galerkin methods for ODEs V: Stiff problems*, Tech. Rep. 2004–12, Chalmers Finite Element Center Preprint Series, 2004.

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

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

[16] ——, *Multi-adaptive Galerkin methods for ODEs III: Existence and stability*, Submitted to SIAM J. Numer. Anal., (2004).

[17] ——, *Multi-adaptive Galerkin methods for ODEs IV: A priori error estimates*, Submitted to SIAM J. Numer. Anal., (2004).

[18] G. Söderlind, *Digital filters in adaptive time-stepping*, ACM Transactions on Mathematical Software, 29 (2003), pp. 1–26.

Chalmers Finite Element Center Preprints

2003–01 *A hybrid method for elastic waves*
Larisa Beilina

2003–02 *Application of the local nonobtuse tetrahedral refinement techniques near Fichera-like corners*
L. Beilina, S. Korotov and M. Křížek

2003–03 *Nitsche's method for coupling non-matching meshes in fluid-structure vibration problems*
Peter Hansbo and Joakim Hermansson

2003–04 *Crouzeix–Raviart and Raviart–Thomas elements for acoustic fluid–structure interaction*
Joakim Hermansson

2003–05 *Smoothing properties and approximation of time derivatives in multistep backward difference methods for linear parabolic equations*
Yubin Yan

2003–06 *Postprocessing the finite element method for semilinear parabolic problems*
Yubin Yan

2003–07 *The finite element method for a linear stochastic parabolic partial differential equation driven by additive noise*
Yubin Yan

2003–08 *A finite element method for a nonlinear stochastic parabolic equation*
Yubin Yan

2003–09 *A finite element method for the simulation of strong and weak discontinuities in elasticity*
Anita Hansbo and Peter Hansbo

2003–10 *Generalized Green's functions and the effective domain of influence*
Donald Estep, Michael Holst, and Mats G. Larson

2003–11 *Adaptive finite element/difference method for inverse elastic scattering waves*
Larisa Beilina

2003–12 *A Lagrange multiplier method for the finite element solution of elliptic domain decomposition problems using non-matching meshes*
Peter Hansbo, Carlo Lovadina, Ilaria Perugia, and Giancarlo Sangalli

2003–13 *A reduced P^1–discontinuous Galerkin method*
R. Becker, E. Burman, P. Hansbo, and M. G. Larson

2003–14 *Nitsche's method combined with space–time finite elements for ALE fluid–structure interaction problems*
Peter Hansbo, Joakim Hermansson, and Thomas Svedberg

2003–15 *Stabilized Crouzeix–Raviart element for the Darcy-Stokes problem*
Erik Burman and Peter Hansbo

2003–16 *Edge stabilization for the generalized Stokes problem: a continuous interior penalty method*
Erik Burman and Peter Hansbo

2003–17 *A conservative flux for the continuous Galerkin method based on discontinuous enrichment*
Mats G. Larson and A. Jonas Niklasson

2003–18 *CAD–to–CAE integration through automated model simplification and adaptive modelling*
K.Y. Lee, M.A. Price, C.G. Armstrong, M.G. Larson, and K. Samuelsson

2003–19 *Multi-adaptive time integration*
Anders Logg

2003–20 *Adaptive computational methods for parabolic problems*
Kenneth Eriksson, Claes Johnson, and Anders Logg

2003–21 *The FEniCS project*
T. Dupont, J. Hoffman, C. Johnson, R. Kirby, M. Larson, A. Logg, and R. Scott

2003–22 *Adaptive finite element methods for LES: Computation of the mean drag coefficient in a turbulent flow around a surface mounted cube using adaptive mesh refinement*
Johan Hoffman

2003–23 *Adaptive DNS/LES: a new agenda in CFD*
Johan Hoffman and Claes Johnson

2003–24 *Multiscale convergence and reiterated homogenization of parabolic problem*
Anders Holmbom, Nils Svanstedt, and Niklas Wellander

2003–25 *On the relationship between some weak compactnesses with different numbers of scales*
Anders Holmbom, Jeanette Silfver, Nils Svanstedt, and Niklas Wellander

2003–26 *A posteriori error estimation in computational inverse scattering*
Larisa Beilina and Claes Johnson

2004–01 *Computability and adaptivity in CFD*
Johan Hoffman och Claes Johnson

2004–02 *Interpolation estimates for piecewise smooth functions in one dimension*
Anders Logg

2004–03 *Estimates of derivatives and jumps across element boundaries for multi-adaptive Galerkin solutions of ODEs*
Anders Logg

2004–04 *Multi-adaptive Galerkin methods for ODEs III: Existence and stability*
Anders Logg

2004–05 *Multi-adaptive Galerkin methods for ODEs IV: A priori error estimates*
Anders Logg

2004–06 *A stabilized non-conforming finite element method for incompressible flow*
Erik Burman and Peter Hansbo

2004–07 *On the uniqueness of weak solutions of Navier-Stokes equations: Remarks on a Clay Institute prize problem*
Johan Hoffman and Claes Johnson

2004–08 *A new approach to computational turbulence modeling*
Johan Hoffman and Claes Johnson

2004–09 *A posteriori error analysis of the boundary penalty method*
Kenneth Eriksson, Mats G. Larson, and Axel Målqvist

2004–10 *A posteriori error analysis of stabilized finite element approximations of the helmholtz equation on unstructured grids*
Mats G. Larson and Axel Målqvist

2004–11 *Adaptive variational multiscale methods based on a posteriori error estimation*
Mats G. Larson and Axel Målqvist

2004–12 *Multi-adaptive Galerkin methods for ODEs V: Stiff problems*
Johan Jansson and Anders Logg

2004–13 *Algorithms for multi-adaptive time-stepping*
Johan Jansson and Anders Logg

These preprints can be obtained from

`www.phi.chalmers.se/preprints`